Not Your Father's
Von Neumann Machine:

A Crash Course in Modern Hardware

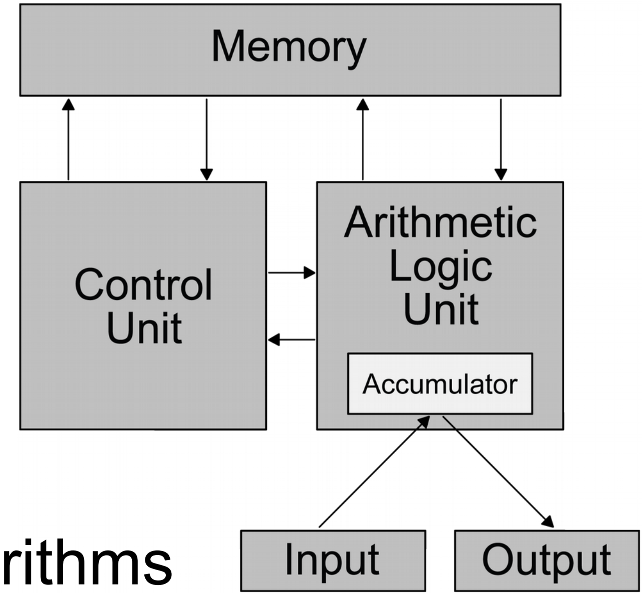
Dr. Cliff Click
cliffc@acm.org
cliffc.org/blog

Kraków, 9-11 May 2018

Agenda

- **Introduction**
- The Quest for ILP
- Memory Subsystem Performance & Data Races
- Specter and Meltdown
- New Performance Models for a New Era

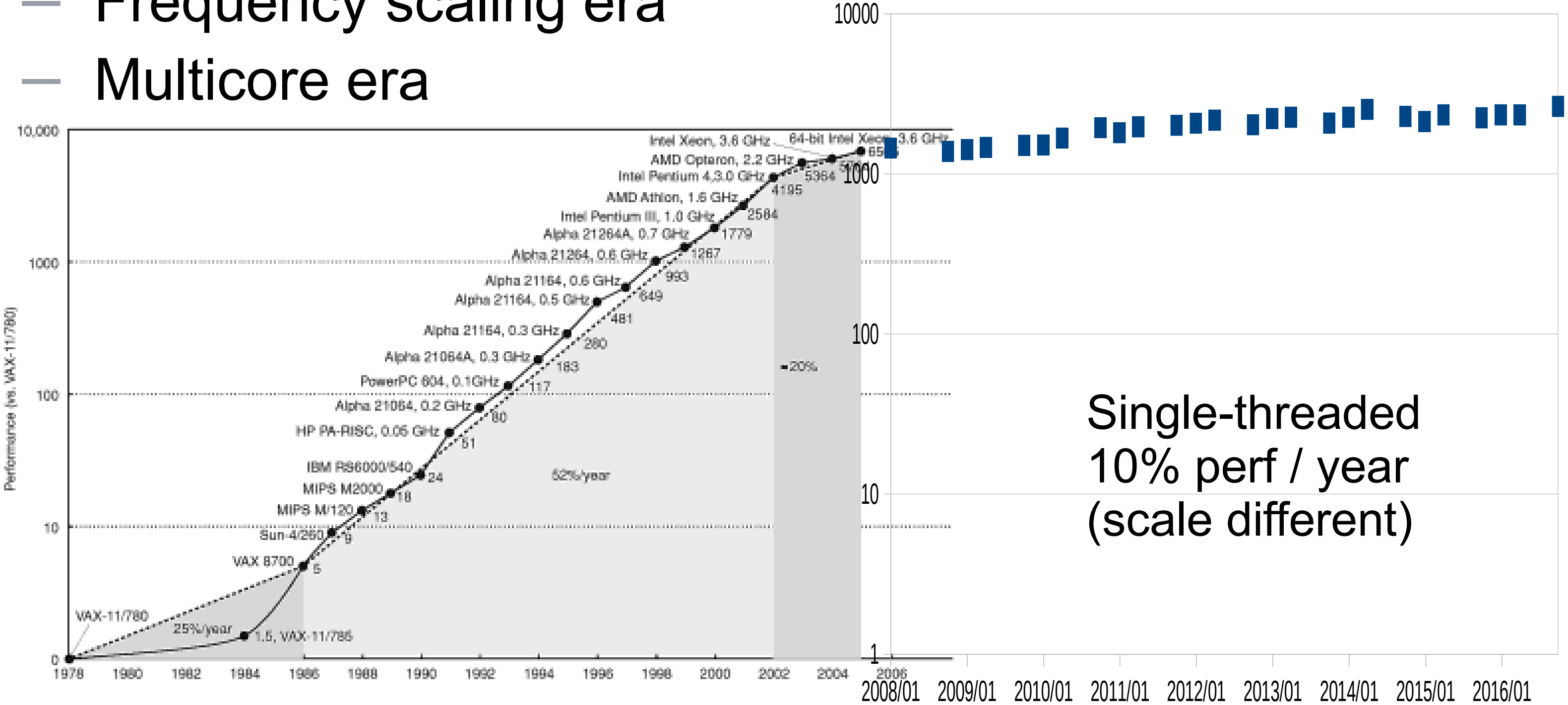
The Von Neumann Machine Architecture



- 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

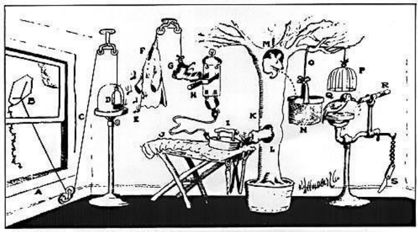
CPU Performance

- 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



Single-threaded
10% perf / year
(scale different)

CISC systems, pre 1988



Pencil Sharpener RUBE GOLDBERG (tm) RGI 038

- CISC ISAs were designed to be *used by humans*
- VAX exemplified the CISC approach
  - Orthogonal instruction set
    - Any instruction, any data type, any addressing mode
  - Exotic hardware primitives for library stuff:
    - Packed character arithmetic, string pattern matching, polynomial evaluation
  - Lots of addressing modes
    - Multiple levels of indirection in a single instruction
      - Convenient to program, hard to pipeline!
    - Example: ADDL3 4(R1)[R2], @8(R1), R3

CISC systems, pre 1988

- CPI (cycles per instruction) for CISC chips varied
  - 4-10 was typical (but highly predictable!)
  - Program performance was basically:
    N*page faults + instructions executed
  - Or, basically, page faults (for typical systems)
    - And just instruction count for embedded systems
  - Page fault count very easy to measure
    - Managing code and data locality key to good performance
- Complex architecture == harder to scale

The era of cheap frequency scaling, 1988-2002

- For about 15 years, we were able to scale CPU performance at ~50% / year
  - Enabled by development of RISC processors
    - Simpler processors → easier to scale
    - Simpler instructions → fewer CPI, better pipelining
  - ISA not practical for programming by hand
    - Example: delay slots
      - Instruction after branch is always executed
      - Some ISAs had data delay slots, which means result of a computation isn't necessarily available to the next instruction
    - Required more sophisticated compilers
  - Memory got cheaper → fewer page faults

Hitting the wall

- Serial performance has hit the wall
  - 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 likely to increase rapidly for some time
- Challenges for programmers
  - How are we going to use those cores?
  - Adjusting our mental performance models?

Agenda

- Introduction
- **The Quest for ILP**
- Memory Subsystem Performance and Data Races
- Specter and Meltdown
- New Performance Models for a New Era

The Quest for ILP

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

The Quest for ILP: Pipelining

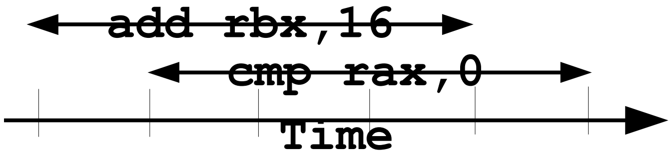
- Internally, each instruction has multiple stages
  - Many of which must be done sequentially
    - Fetching the instruction from memory
      - Also identifying the end of the instruction (update PC)
    - Decoding the instruction
    - Fetching needed operands (memory or register)
    - Performing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

```
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

- 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 really low now
  - But a miss costs 100-1000 instruction issue slots
  - So a ~5% miss rate dominates performance

How did this turn out?

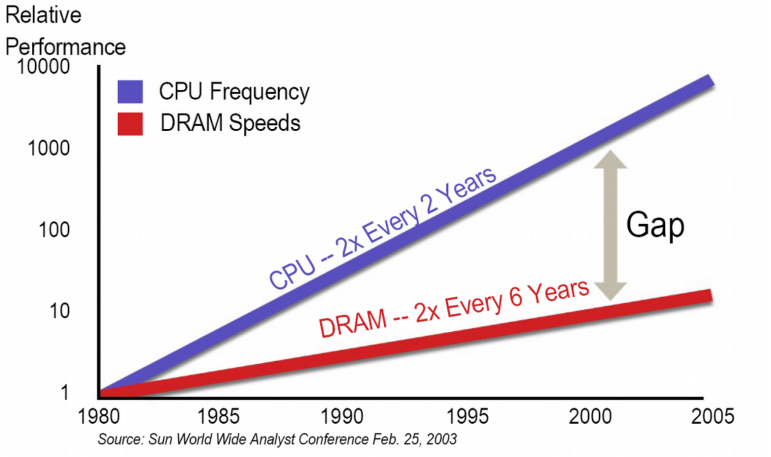
- ILP is mined out
  - As CPUs get more complicated, more transistors are thrown at dealing with the hazards of ILP
    - Like speculative execution
    - Instead of providing more computational power
  - Moore's law gives us a growing transistor budget
  - But we spend more and more on ILP hazards
-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

Agenda

- Introduction
- The Quest for ILP
- **Memory Subsystem Performance and Data Races**
- Specter and Meltdown
- New Performance Models for a New Era

Memory subsystem performance

- Chart shows speedup in CPU vs memory
  - Exponentially widening gap



Source: Sun World Wide Analyst Conference Feb. 25, 2003

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

Types of memory

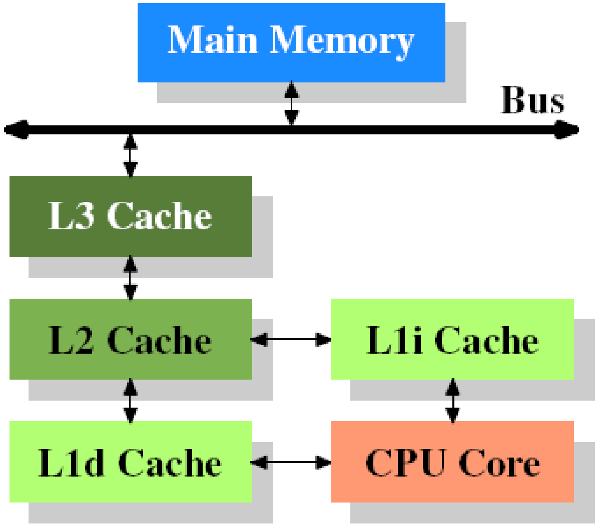
- Static RAM (SRAM) – fast but expensive
  - Six transistors per bit
- Dynamic RAM (DRAM) – cheap but slow
  - One transistor + one capacitor per bit
  - Improvements in DRAM (DDR, DDR2, DDR4, etc) improve bandwidth but latency not so much
  - More improvements in power & density than speed

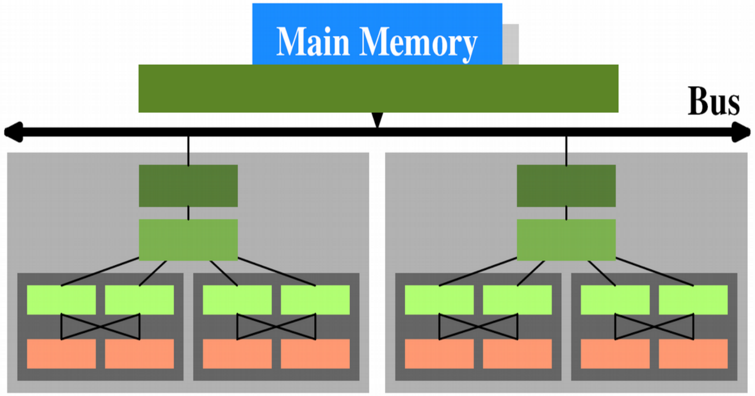
Caching

- 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





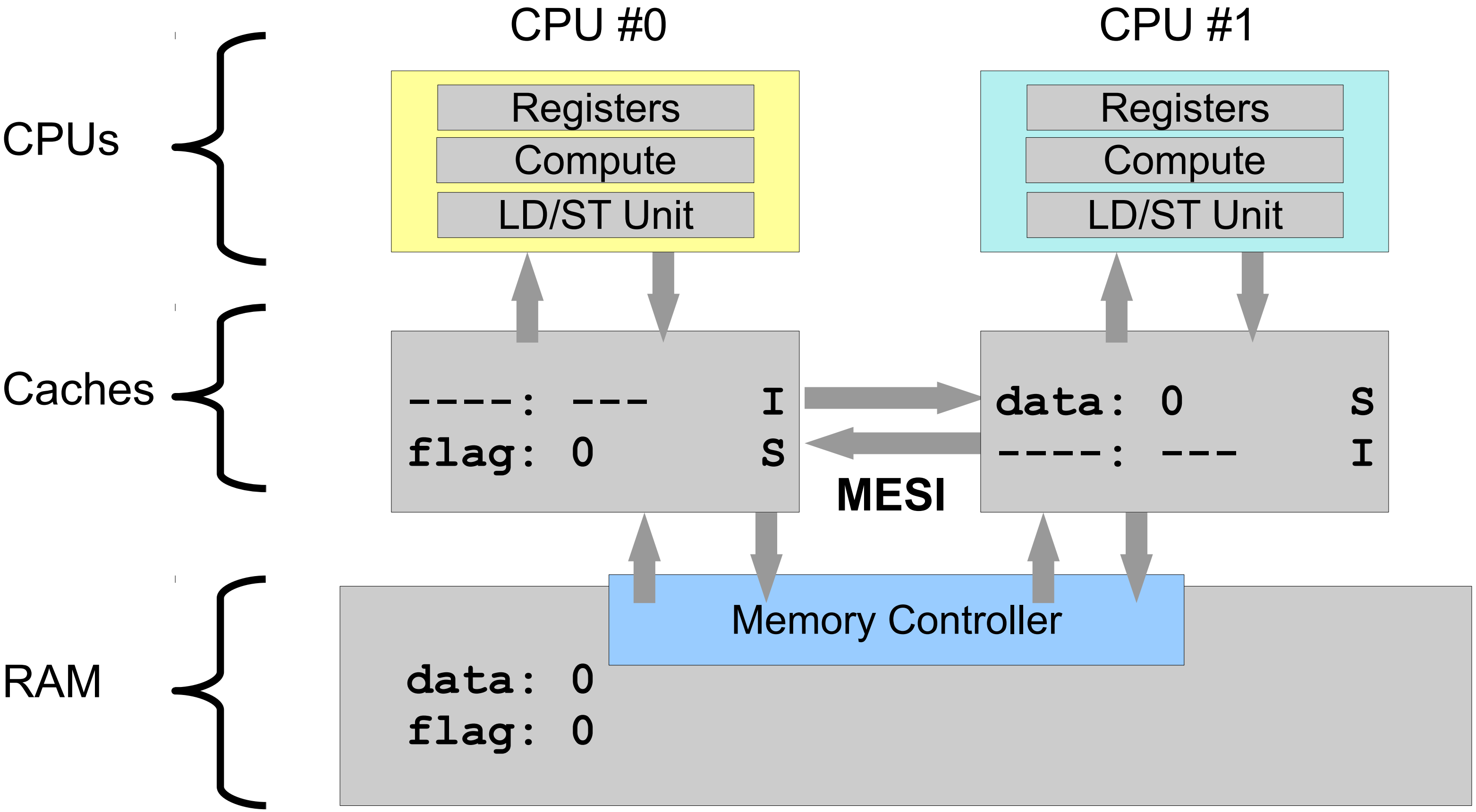
Caching

- 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)

Real Chips Reorder Stuff



CPU #0

CPU #1

CPUs

Registers

Compute

LD/ST Unit

Registers

Compute

LD/ST Unit

Caches

```
----: ---      I
flag:  0       S
```

```
data:  0       S
----: ---      I
```

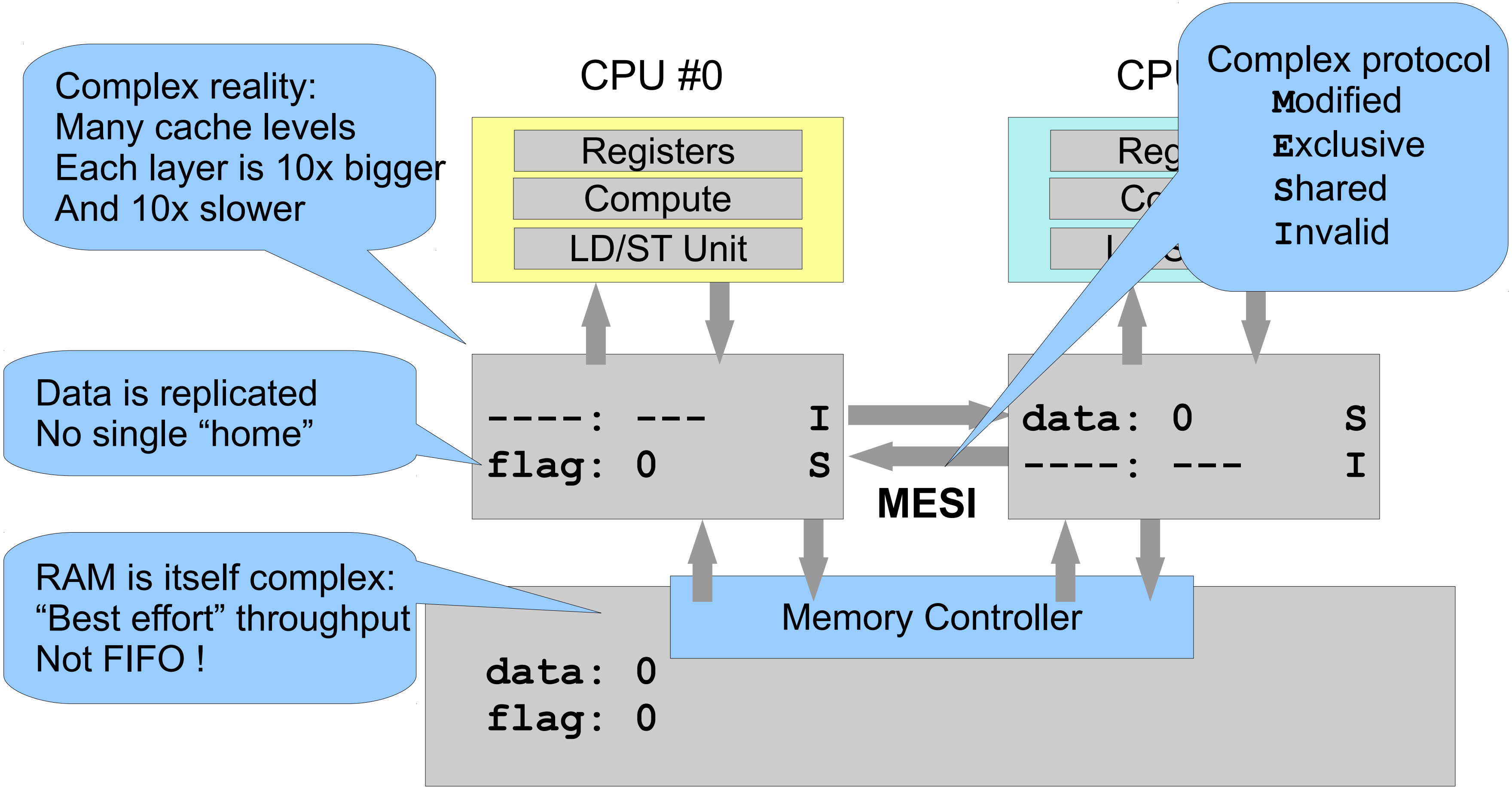
MESI

RAM

Memory Controller

```
data:  0
flag:  0
```

Real Chips Reorder Stuff



Complex reality:
Many cache levels
Each layer is 10x bigger
And 10x slower

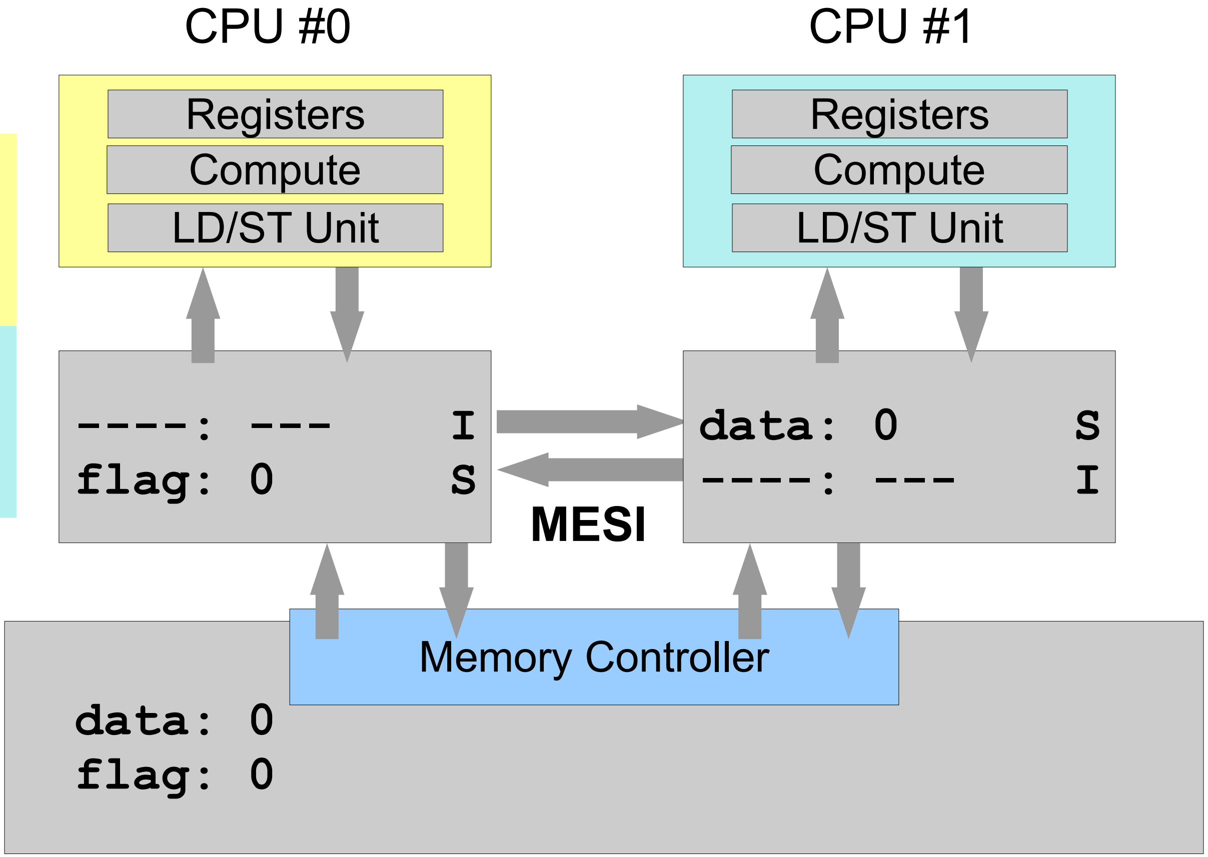
Data is replicated
No single “home”

RAM is itself complex:
“Best effort” throughput
Not FIFO !

Complex protocol
Modified
Exclusive
Shared
Invalid

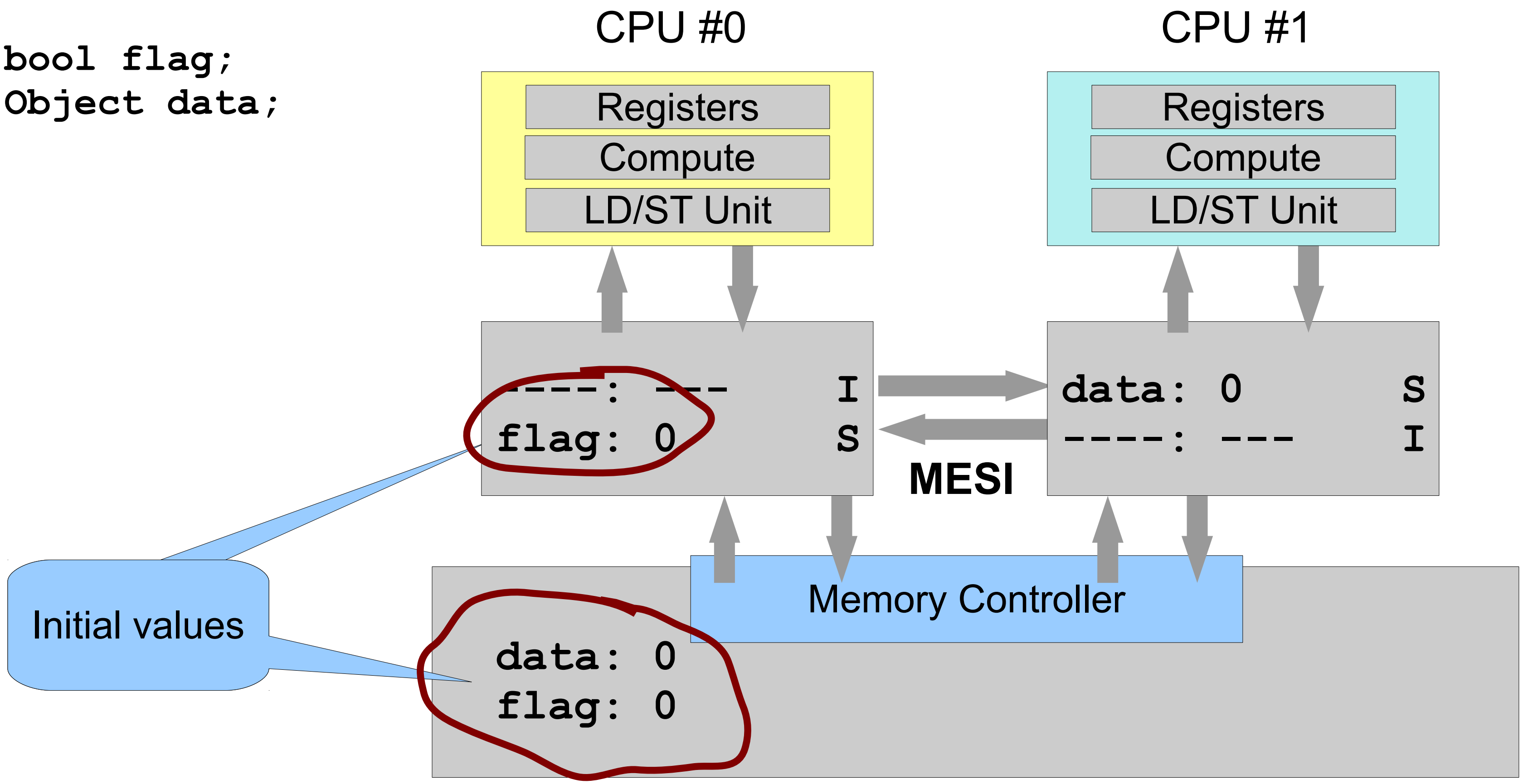
Real Chips Reorder Stuff

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



Real Chips Reorder Stuff

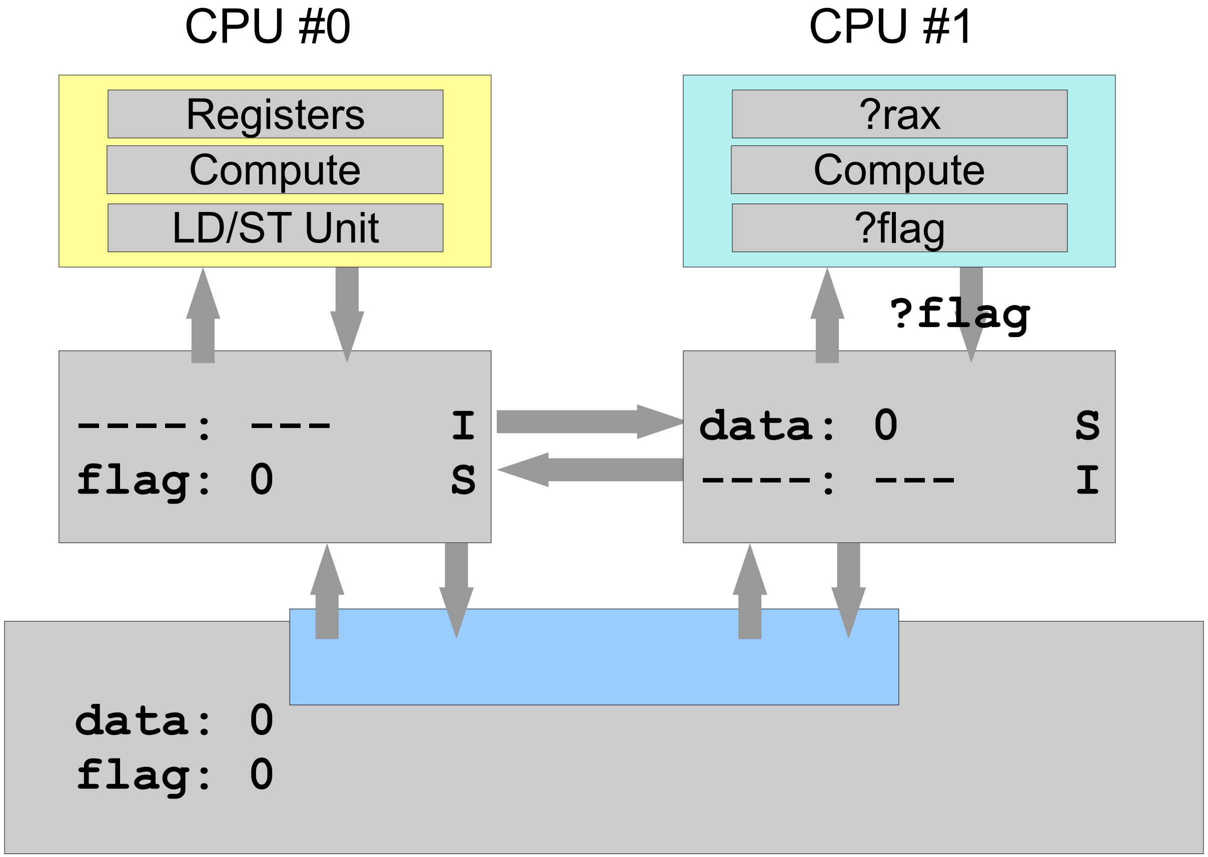
```
bool flag;
Object data;
```



Real Chips Reorder Stuff

if(!flag) ...

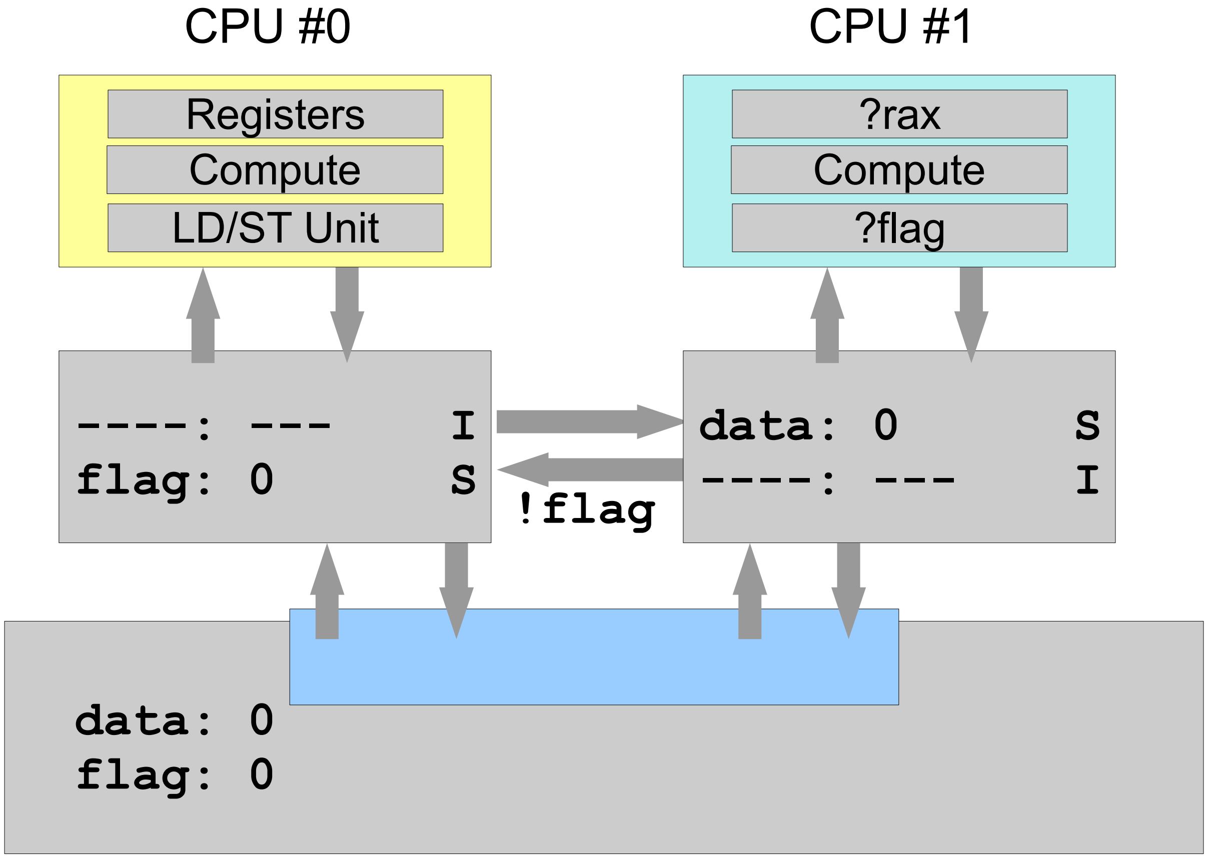
```
ld  rax,[&flag]
```



Real Chips Reorder Stuff

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

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



CPU #0

Registers
Compute
LD/ST Unit

CPU #1

?rax
Compute
?flag

----: --- I
flag: 0 S

data: 0 S
----: --- I

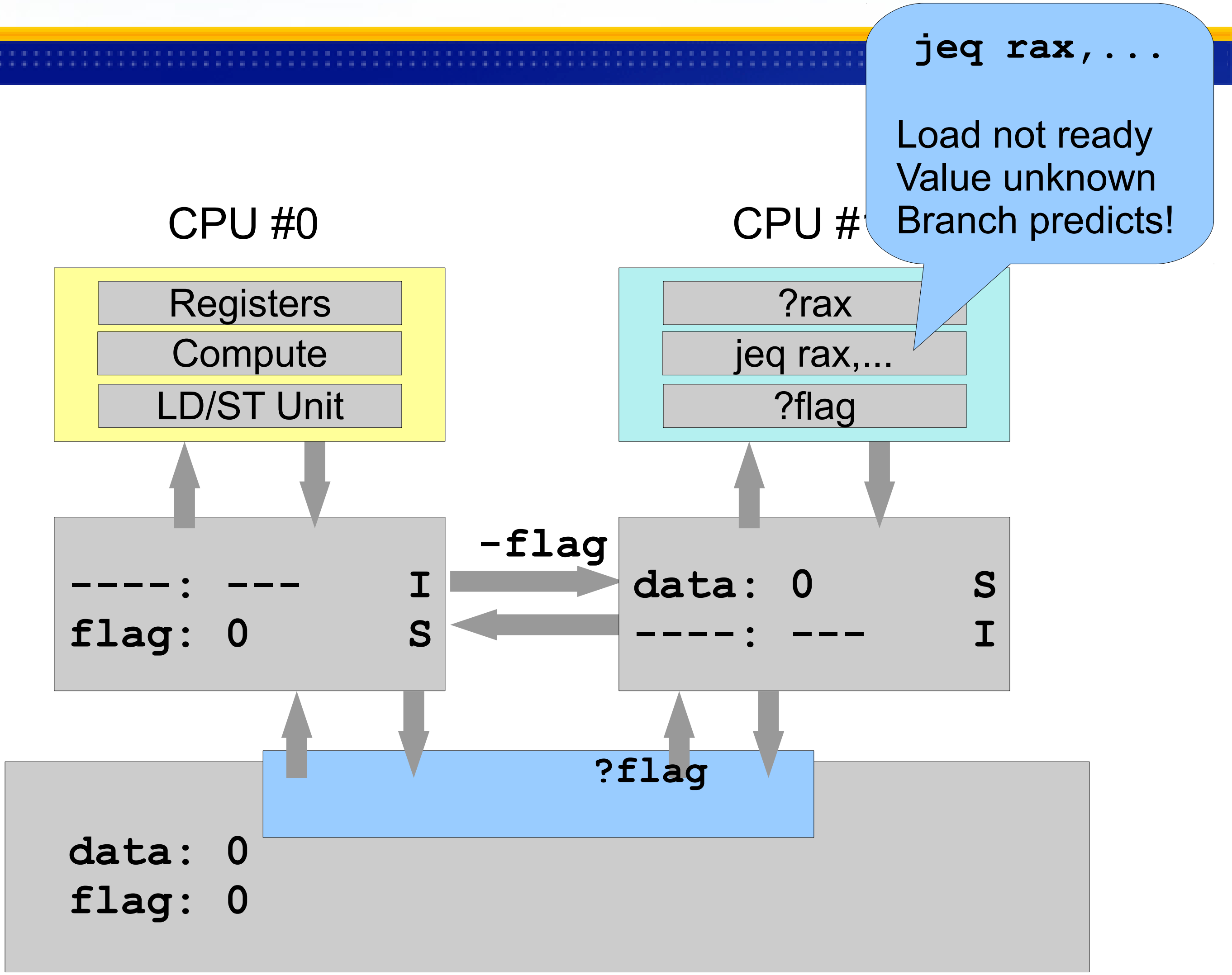
!flag

data: 0
flag: 0

Real Chips Reorder Stuff

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

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



`jeq rax,...`

Load not ready
Value unknown
Branch predicts!

CPU #0

Registers
Compute
LD/ST Unit

CPU #

?rax
jeq rax,...
?flag

`----: ---` I
`flag: 0` S

`-flag`

`data: 0` S
`----: ---` I

`?flag`

`data: 0`
`flag: 0`

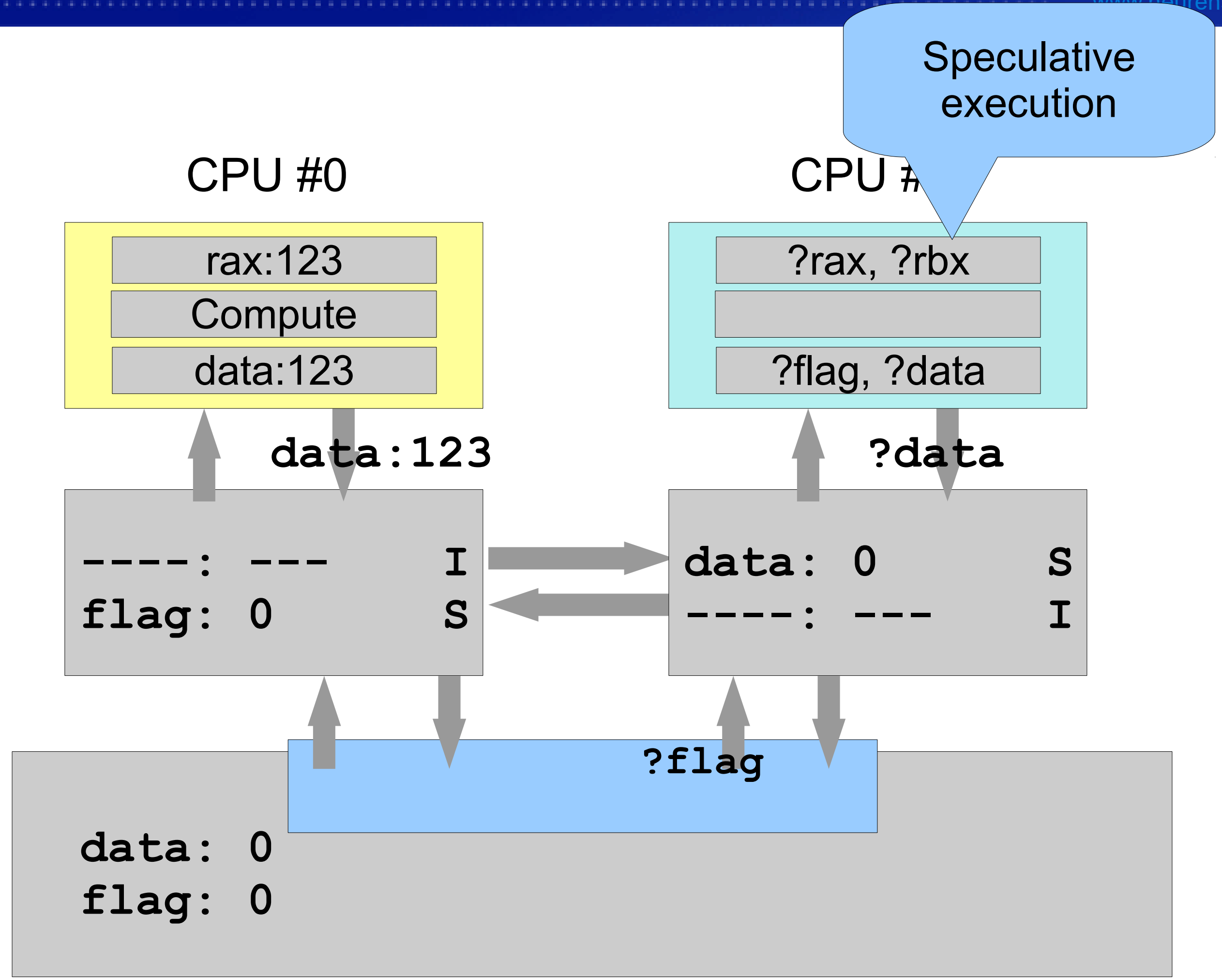
Real Chips Reorder Stuff

data = ...;

if(!~~flag~~) …
return data;

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

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



Speculative execution

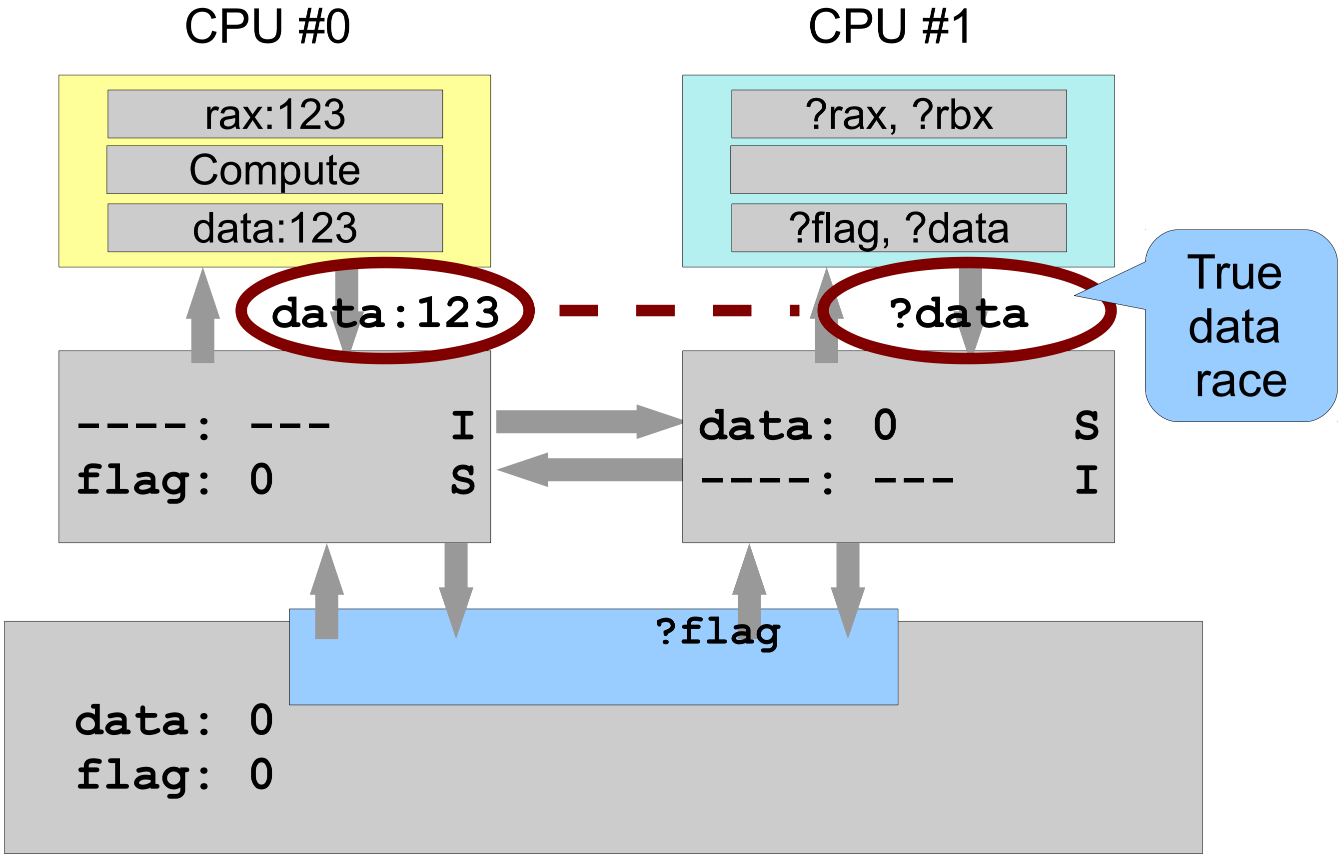
Real Chips Reorder Stuff

data = ...;

if(!~~flag~~) …
return data;

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

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



CPU #0

rax:123
Compute
data:123

data:123

----: ---   I
flag: 0   S

CPU #1

?rax, ?rbx

?flag, ?data

?data

data: 0   S
----: ---   I

True data race

?flag

data: 0
flag: 0

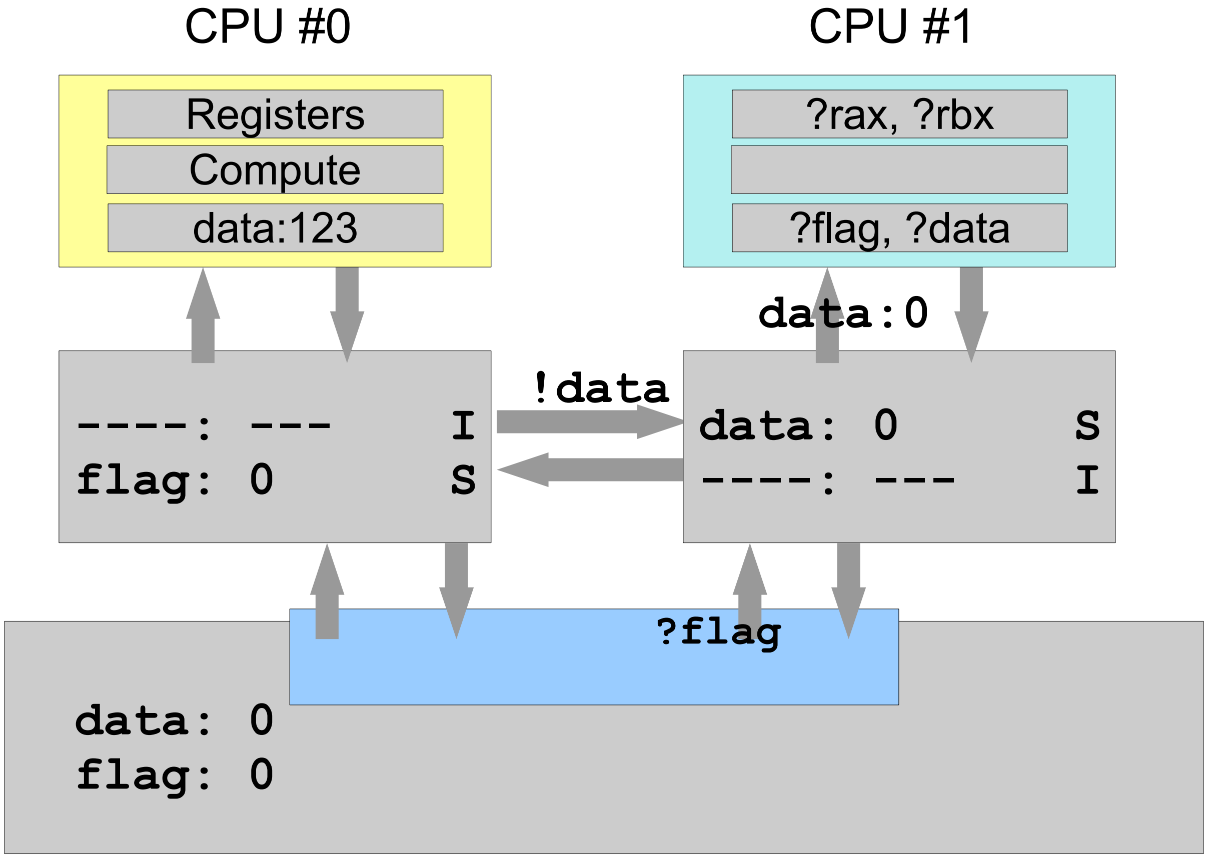
Real Chips Reorder Stuff

~~data = ...;~~

if(!~~flag~~) …
return data;

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

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



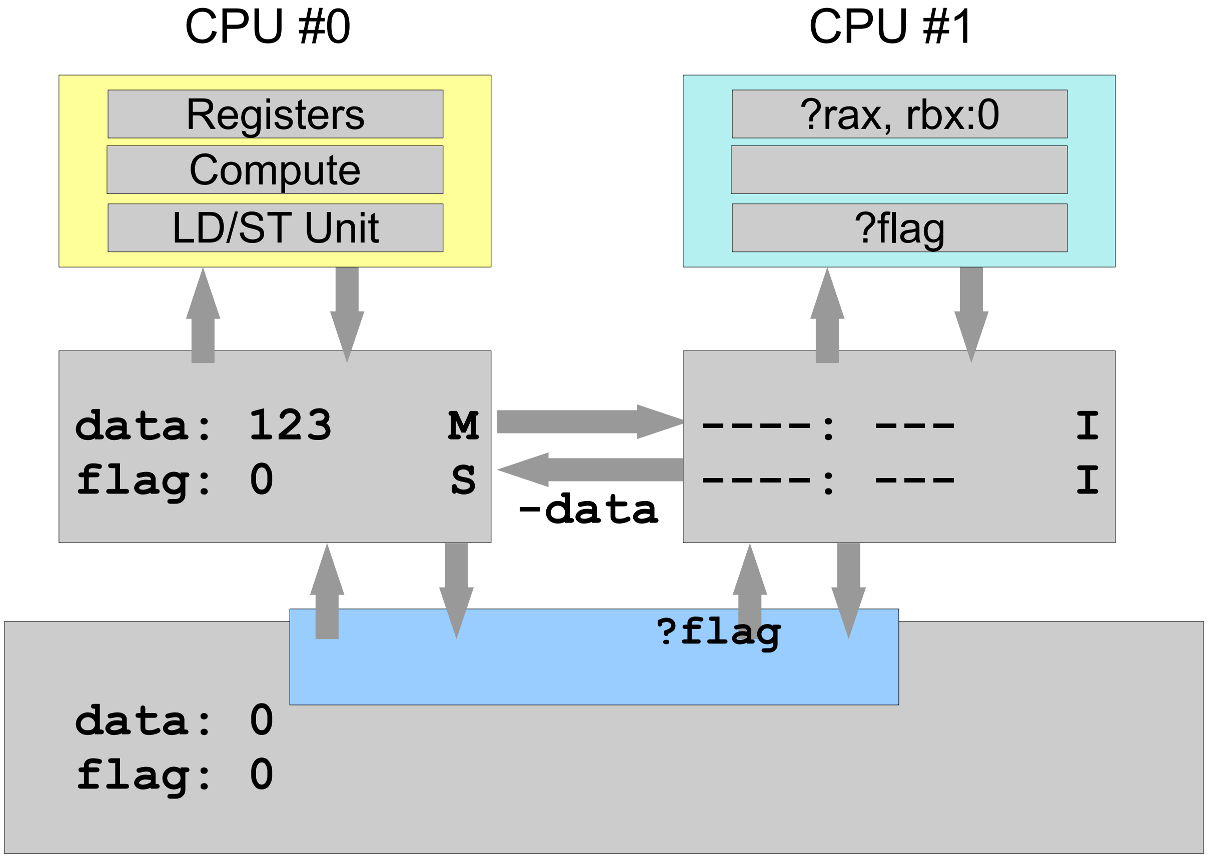
Real Chips Reorder Stuff

~~data = ...;~~

if(!~~flag~~) …
return data;

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

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



Real Chips Reorder Stuff

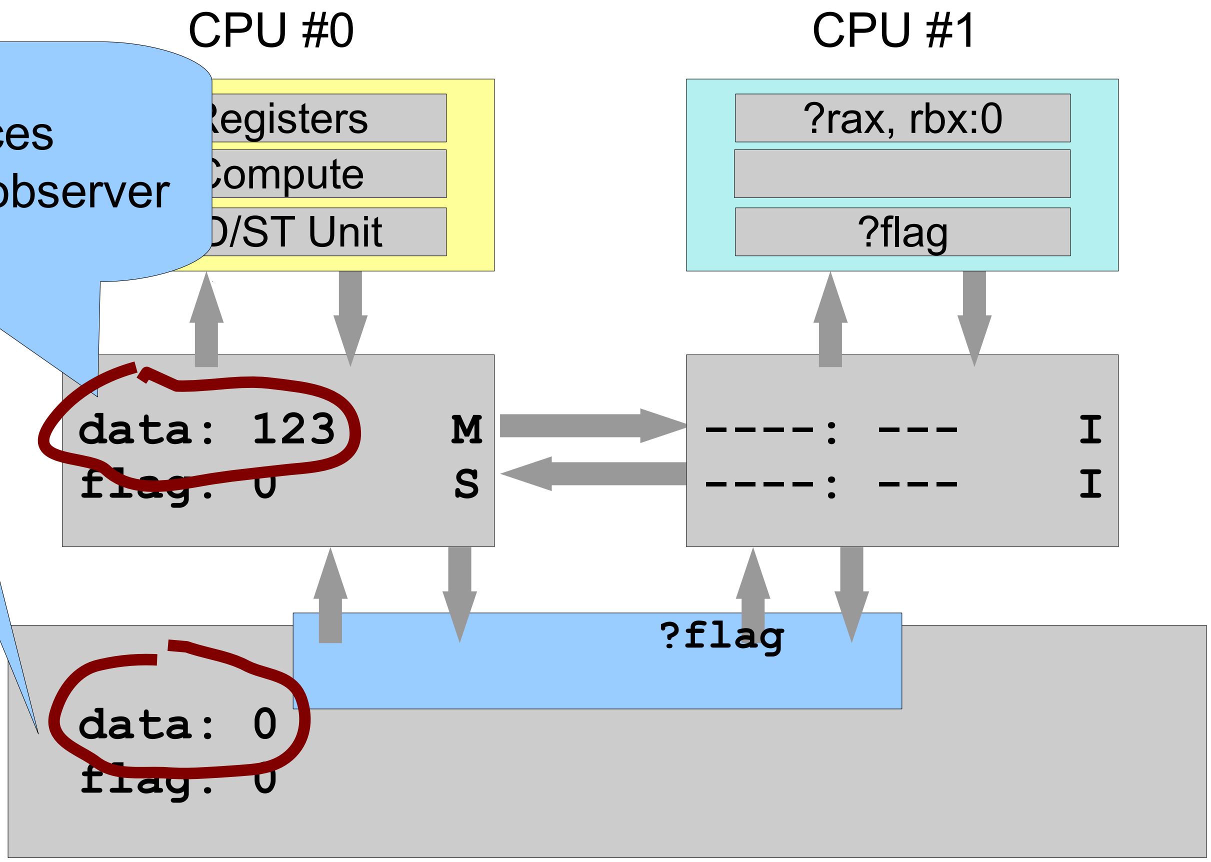
d

return data;

`data` in 2 places
Value is relative to observer

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

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



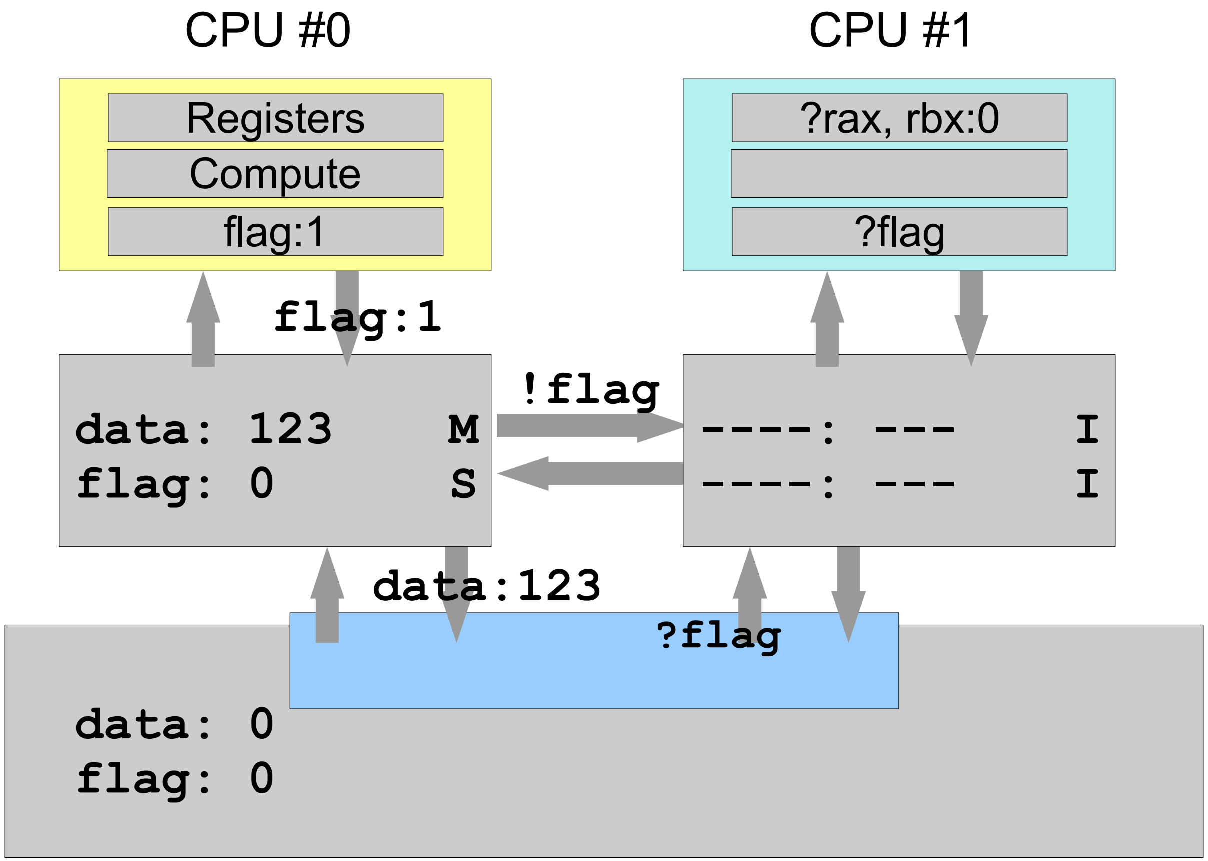
Real Chips Reorder Stuff

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

if(!~~flag~~) …
return data;

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

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



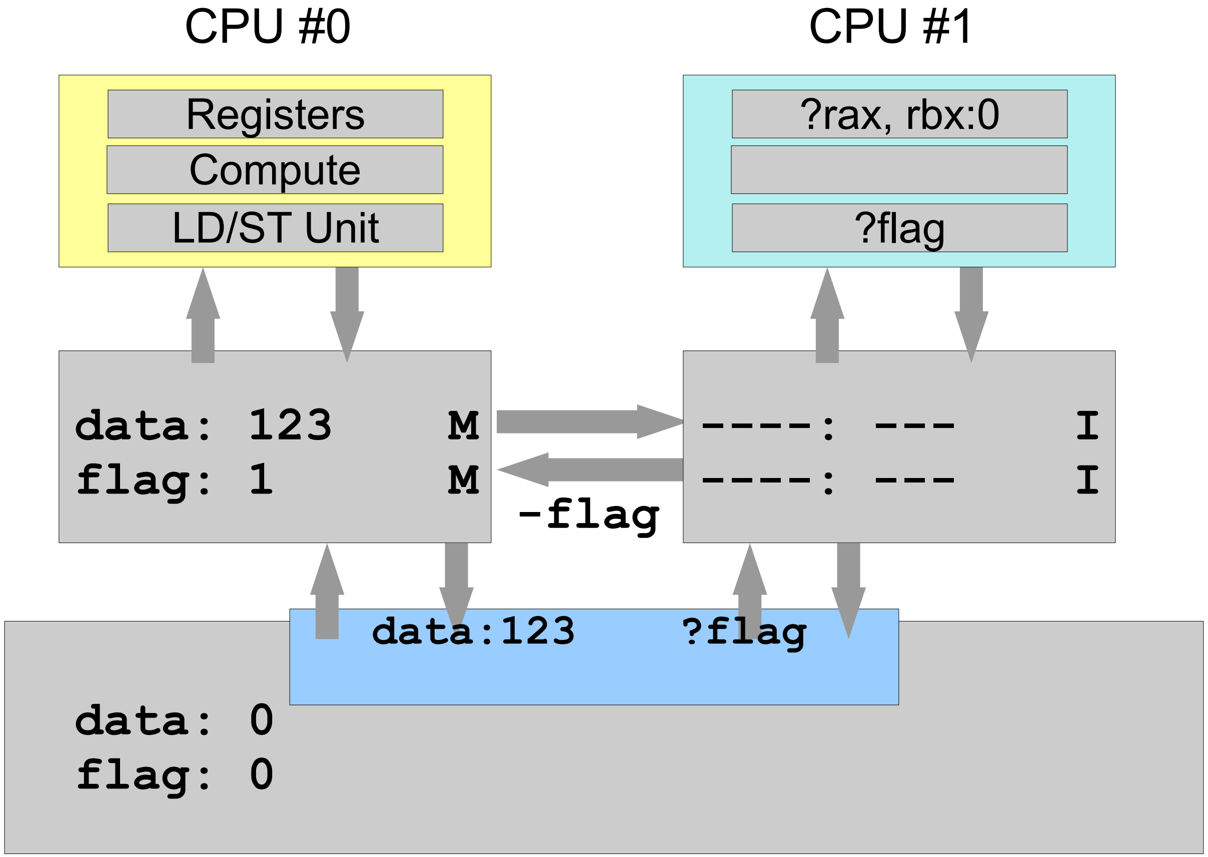
Real Chips Reorder Stuff

~~data = …;~~
~~flag=true;~~

if(!~~flag~~) …
return data;

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

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



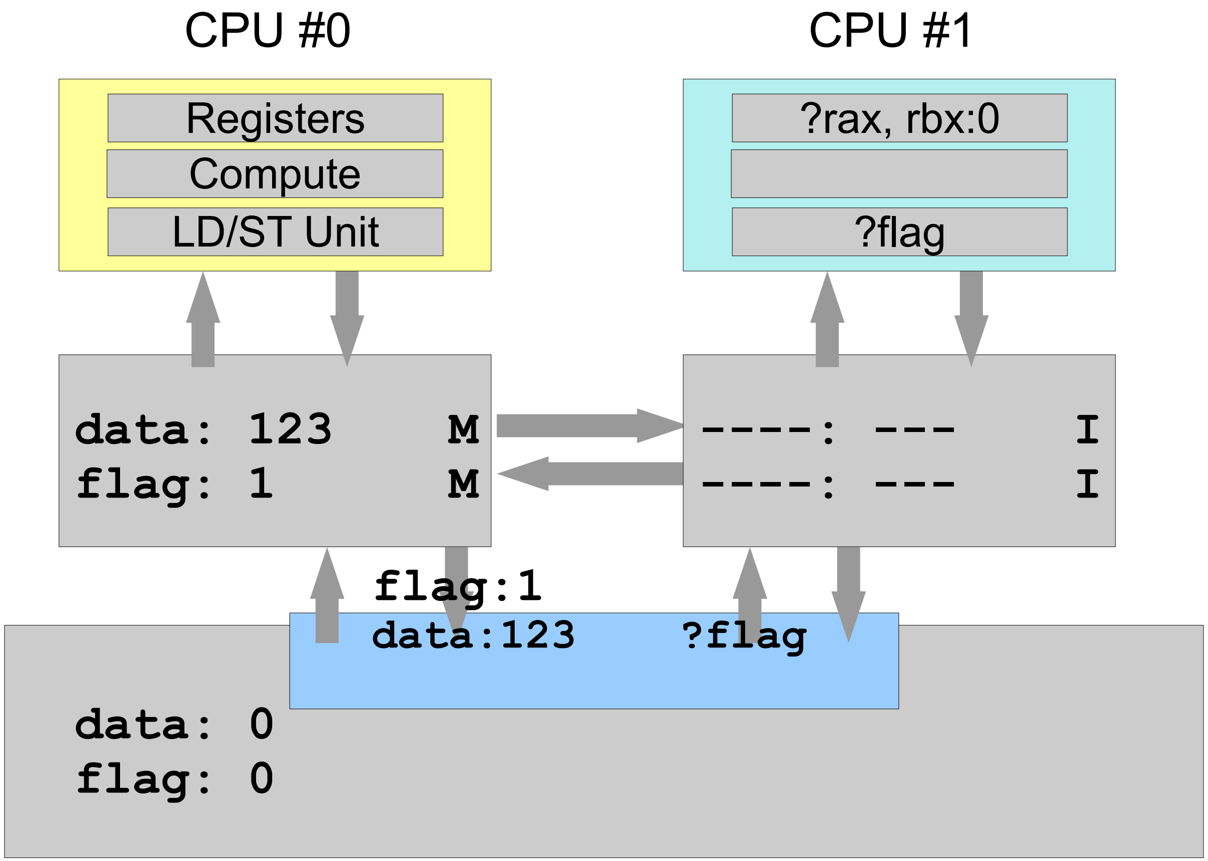
Real Chips Reorder Stuff

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

if(!~~flag~~) …
return data;

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

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



CPU #0

Registers

Compute

LD/ST Unit

CPU #1

?rax, rbx:0

?flag

```
data:  123     M
flag:  1       M
```

```
----:  ---     I
----:  ---     I
```

```
flag:1
data:123     ?flag
```

```
data:  0
flag:  0
```

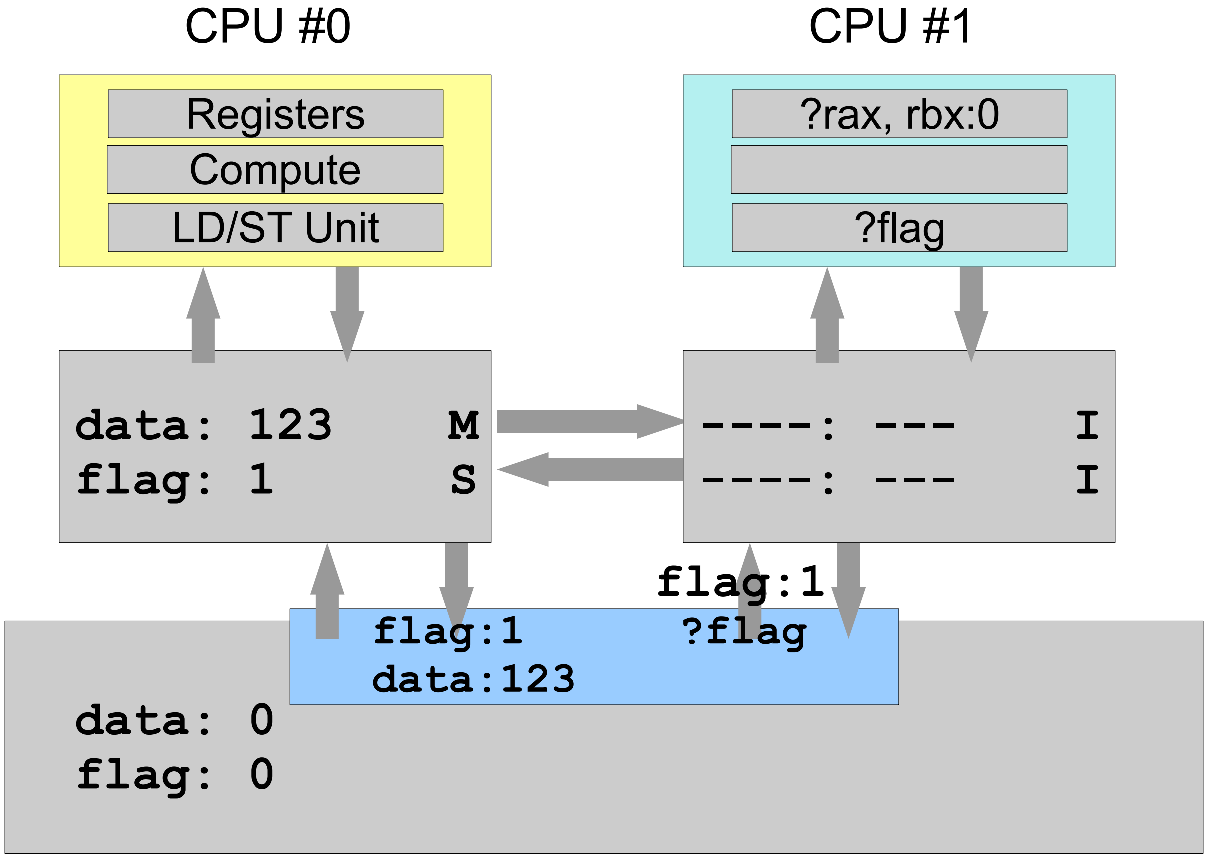
Real Chips Reorder Stuff

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

if(!~~flag~~) …
return data;

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

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



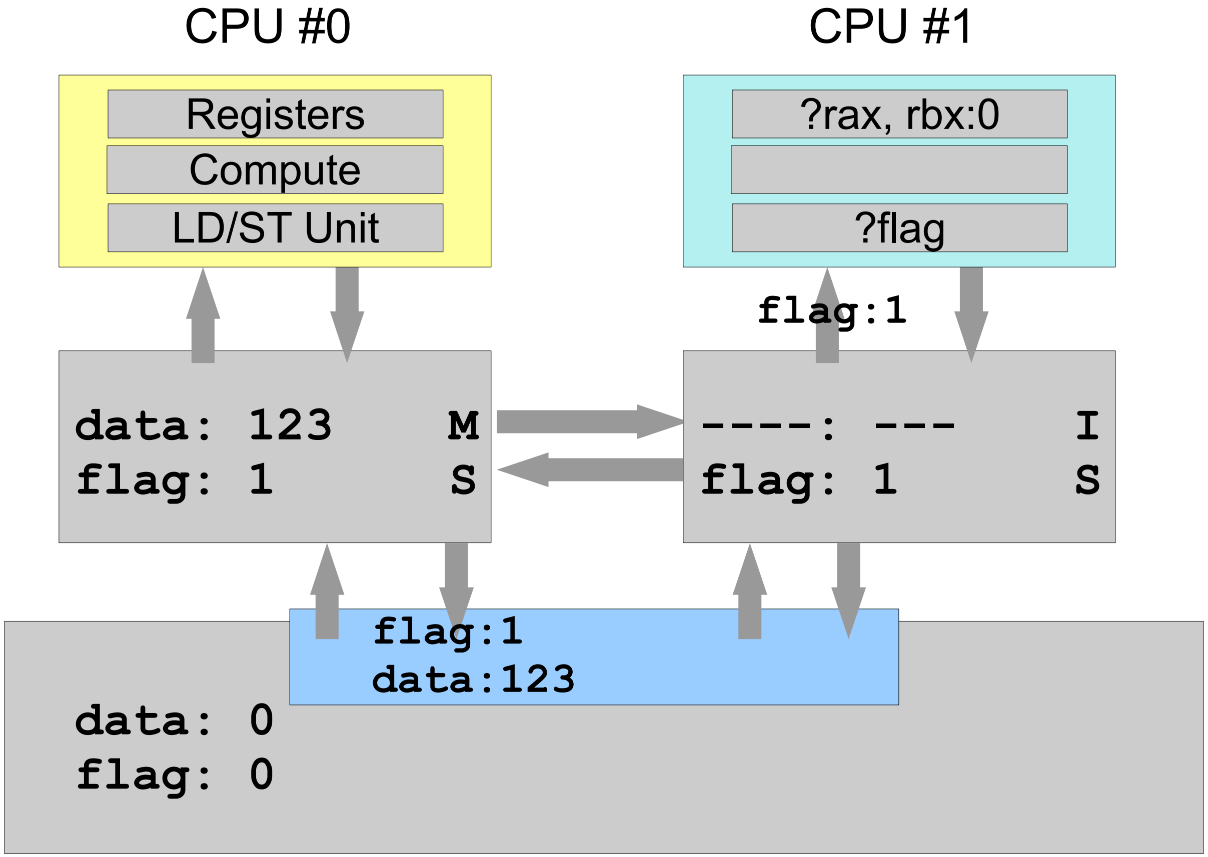
Real Chips Reorder Stuff

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

if(!~~flag~~) …
return data;

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

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



CPU #0

Registers
Compute
LD/ST Unit

CPU #1

?rax, rbx:0

?flag

flag:1

data: 123 M
flag: 1 S

----: --- I
flag: 1 S

flag:1
data:123

data: 0
flag: 0

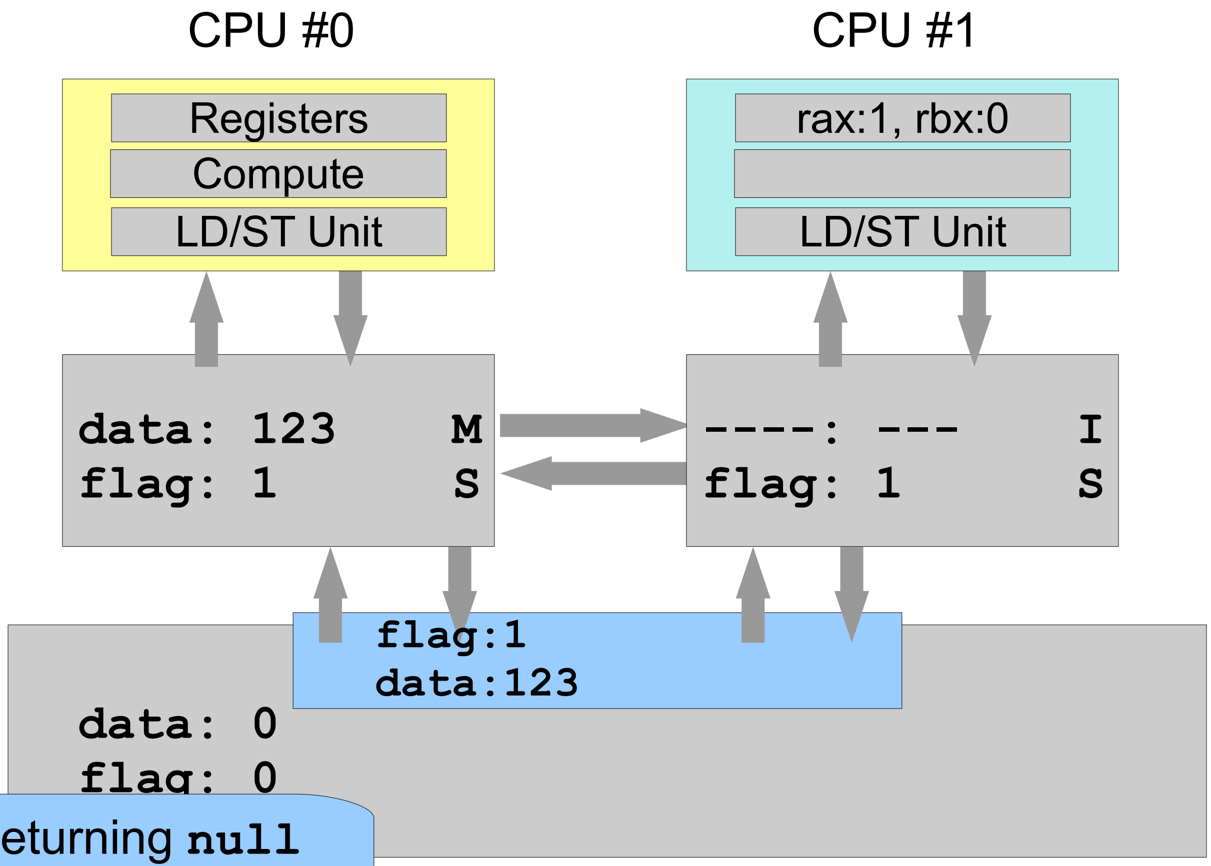
Real Chips Reorder Stuff

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

~~if(!flag) …~~
return data;

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

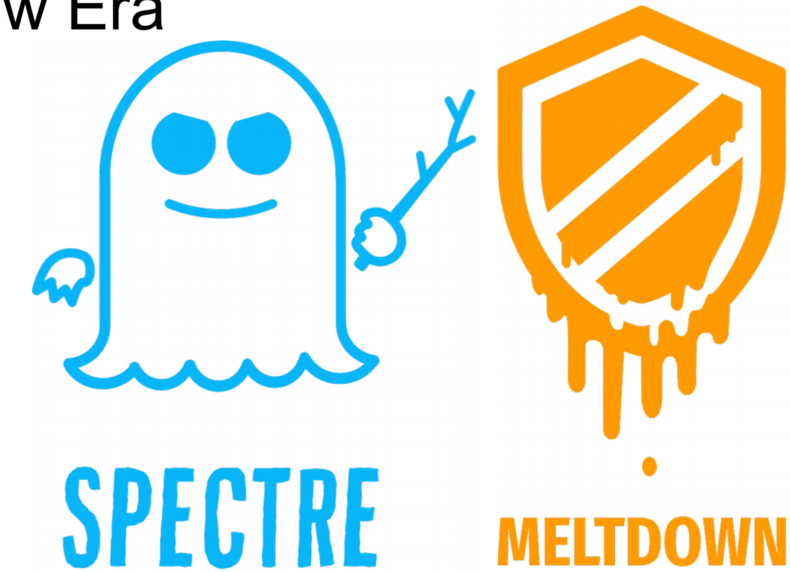
~~`ld  rax,[&flag]`~~
~~`jeq rax,...`~~
~~`ld  rbx,[&data]`~~
`ret rbx`



Returning **`null`**
Crash-n-burn time!

Agenda

- Introduction
- The Quest for ILP
- Memory Subsystem Performance and Data Races
- **Specter and Meltdown**
- New Performance Models for a New Era



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 normal process
  - Security works as expected
- Then read cache via timing
  - *Side-channel timing attack*

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

Specter

- Find 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

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

- 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
key	*k*
junk+64	0
junk+96	0

Specter: Run the Attack

- Load **`ary.length`** misses in cache

```
ld4     Rlen,[Rary+4]
```

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

Specter: Run the Attack

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

```
ld4    Rlen,[Rary+4]
cmp    Rx,Rlen
```

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

Specter: Run the Attack

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

```
ld4   Rlen,[Rary+4]
cmp   Rx,Rlen
ja    fail_check
```

address	value
junk	0
key	*k*
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+4]`**
`cmp`	**`Rx,Rlen`**
`ja`	**`fail_check`**
`ld1`	**`Rk,[Rary+Rx]`**
`mul`	**`Rtmp,Rk*256`**

address	value
junk	0
key	*k*
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+4]`
`cmp`	`Rx,Rlen`
`ja`	`fail_check`
`ld1`	`Rk,[Rary+Rx]`
`mul`	`Rtmp,Rk*256`
`ld8`	`Ry,[Rtmp+Rbuf]`

address	value
junk	0
key	*k*
junk+64	0
*k**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
- The Quest for ILP
- Memory Subsystem Performance and Data Races
- Specter and Meltdown
- **New Performance Models for a New Era**

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

- 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

Share & 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!

New metrics, new tools

- With Chip-Multi-Threading, speedup depends on how memory is used:
  - 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 cache-utilization tools, bandwidth tools
- Out-of-cache is hard to spot in most profilers
  - Just looks like all code is slow...

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

For more information

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

Dr. Cliff Click
cliffc@acm.org
http://www.cliffc.org/blog