The Azul Systems' Hardware Transactional Memory Story

Cliff Click

Who Am I?

Cliff Click

Leader, Founder

Cratus, Rocket School, Neurensic,
H2O.ai, Azul, Sun

cliffc@acm.org

PhD Computer Science

1995 Rice University

HotSpot JVM Server Compiler

“showed the world JITing is possible”

45 yrs coding

40 yrs building compilers

35 yrs distributed computation

30 yrs OS, device drivers, HPC, HotSpot

15 yrs Low-latency GC, custom java hardware,
NonBlockingHashMap

10 yrs ML tool building, ML applications

20+ patents, dozens of papers

100s of public talks

Spoiler: HTM doesn’t “work”

- **H**ardware **T**ransactional **M**emory will not allow serious parallel execution of “junk Java code”
- For reasons that are obvious… in hindsight
- Can have good impact in “small library use-cases”
  - Such as various kernel ops (eg context switch)
  - Or very tight code (eg hi-freq trading netstack)
- Industry found other solutions to parallel workloads
  - Clusters & Microservices vs Large Shared Memory
- This is a story about Azul Systems’ HTM effort

Some History of This Talk

- Azul Systems: Started ~2002
- Target: Big Business Java; portals, web servers
- Big Parallel Java
  - Big Shared Memory coding style
    - Perceived much easier than distributed coding
  - Thread pools & worklists very common
  - GC pauses a huge issue
  - Irregular locking around e.g. caches
- Big Effort for Big Gain: Custom Hardware

Big Gains: Custom Hardware

- Big Core Count
  - A core for every task, lambda, Runnable
  - Cores dedicated to GC, JIT’ing and I/O
- Simple cores (but not too simple, e.g. GPUs)
  - Classic 3-addr 64-bit RISC cores
  - Small Caches, Low(er) Freq
  - Read Barrier for Low-Latency GC
  - Other custom ops to make up for low-frequency
  - Hardware Transactional Memory to for Java Locks
    - *In 2002! Intel Haswell has HTM a decade later*

What Worked Well

- Irregular thread-parallel workloads
  - “Parallel Worker Threads” doing unrelated tasks
  - Handling e.g. web service requests
- Hardware was amazingly good at big web portals
- GC untouchable even 15 yrs later
  - 40G/sec allocation on 1Tb live heap
  - Max pause ~low milliseconds
- Read barrier op, inline-cache op, range-check op
- Built-in JVM profiling just catching up
  - Still nothing on e.g. L1 cache profiling

Expected Use Case

- “Parallel Worker Threads” doing unrelated tasks
  - Expect lots of locking & synchronization around shared infrastructure
  - e.g. DB caches, Web caches
- Expect “useless” contention
  - Locking for Readers against rare Writer
  - But also readers blocking each other
  - Difficult to debug, so “junk locks” added just-in-case
- Expect HTM to allow <u>parallel execution thru locks</u>

HashMap for Caches

- Poster Child use-case:
  - Shared Large HashMap used as a Cache
  - Mostly Reads
  - Rare Writers updating Cache
  - Readers blocked by Rare Writer
  - Readers blocked by Each Other!
- HTM will allow parallel readers, writers
  - As long as touching different parts of table
- **Parallel access without lock-free computing**

Java Locks

- Well accelerated in software already
- Three performance domains:
  - **No contention**: CAS to lock/unlock
    - Atomic Compare-And-Swap
  - **Light contention** – spinning & retries
  - **Heavy contention**: block in OS
- Late-in-life added 4th domain:
  - No contention **and frequently locked**: Biased locking
    - “Pre-lock”, difficult to release the lock but free to use

Java Locks

- **Biased Locking:** Azul ahead of Sun by a decade
- **No contention**: CAS to lock/unlock
  - Azul CAS can “hit” in L1 cache
  - Repeated un-contended locks take 3 clks to acquire & release (plus fencing costs)
  - X86 gets there in ~2012
- **Light contention** – spinning & retries
- **Heavy contention**: block in OS
  - Azul: fair locks in the OS
  - Good support for 1000 cores & 100K runnable threads

Java Locks

- **Biased Locking:** Azul ahead of Sun by a decade
- **No contention**: CAS to lock/unlock
  - Azul CAS can "hit" in L1
  - Repeated un-contended locks take 3 clks to acquire & release (plus fencing costs)
  - X86 gets there in ~2012
- **Light contention** – spinning & retries
- **Heavy contention**: block in OS
  - Azul: fair locks in the OS
  - Good support for 1000 cores & 100K runnable threads

HTM Targeted Here

HTM running A Java Lock in Parallel

- **H**ardware **T**ransactional **M**emory
  - Allows a **set** of memory ops to happen **atomically**
  - In a *transaction*
- Two (or more) threads running in the same code
  - But only reading memory
  - Or writing unrelated memory addresses
  - Can execute a transaction in parallel
- Use HTM to guard the transaction
  - Will abort the XTN on a conflict

HTM in L1 D-Cache

- L1 is 16K: 512 lines of 32b each
  - And 4-way associative
- L1 controls visibility to all other cores
  - What’s in my L1 not visible to any other core
  - Except when L1 evicts a dirty line
- Store transaction updates in your L1
  - If aborted: invalidate dirty lines instead writing back
  - If success: show dirty lines as-needed
  - Needs 1 extra bit-per-line for in/out of XTN

HTM in L1 D-Cache

- In theory XTNs as large as 16K
  - Limited by 4-way associativity
- Mark touched lines with XTN bit
  - As long as hardware does not “lose a XTN line”
  - Then known to be atomic!
- Java lock/unlock uses HTM
  - Uncontended locks use CAS (or biased locking)
  - Heavy contention blocks in OS
  - Everybody else: <u>try the HTM first</u>

HTM’ing a Lock

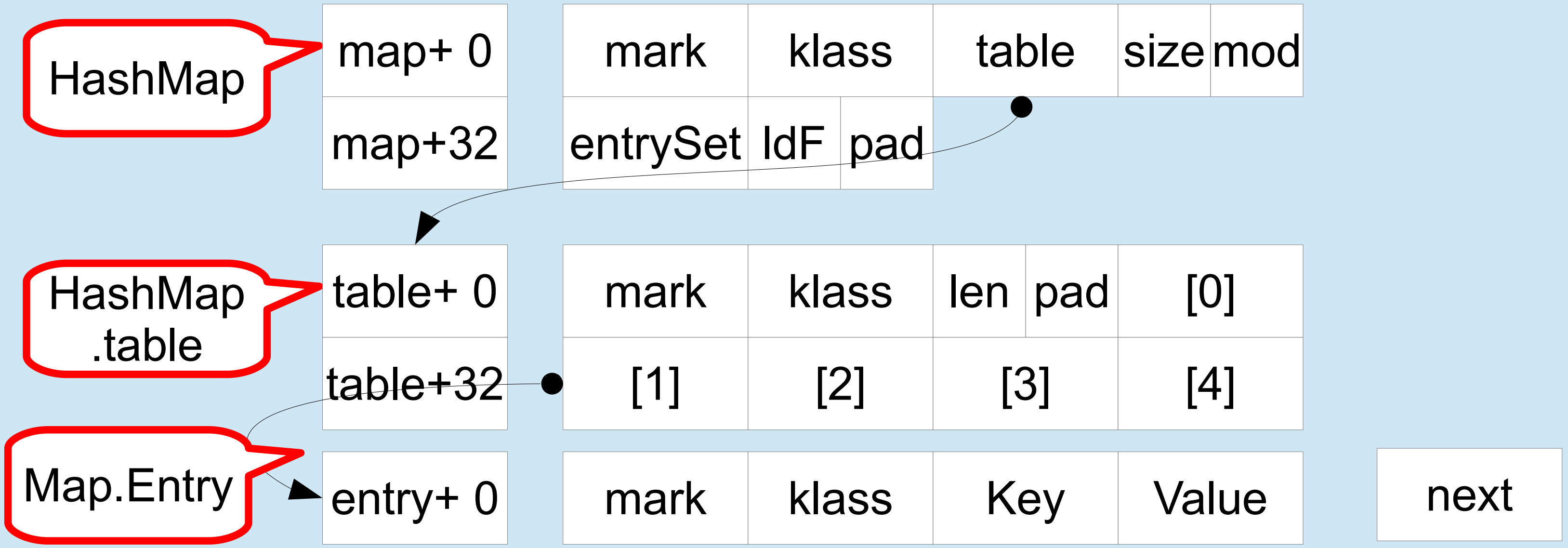
- Attempt acquire w/CAS
- CAS fails
- Inspect for inflated lock header
  - Install inflated lock as needed
- Inspect inflated lock for contention / HTM fails
  - Heuristic: bail to OS for blocking
- Bump counters in inflated lock
- Turn on HTM –
  - And now every touched line is XTN marked

HTM’ing a Lock

- Run Java code until
  - Hit Unlock or L1 attempts to throw out XTN line
- Unlock: “commit” HTM, turn off special marking
  - Update counters in inflated lock
  - Carry-on! Lock worked, was atomic
- Lose-A-Line: Abort!
  - Hardware throws interrupt instead of writing line out
  - All dirty & marked lines are marked Invalid
  - No write-back of dirty
  - Software retry – might CAS, spin, block or HTM

Example!

- Shared synchronized HashMap
  - Some active readers
  - Writer writes to unrelated part of table
  - Want everybody to run concurrently
- HashMap data layout for 64b JVM:

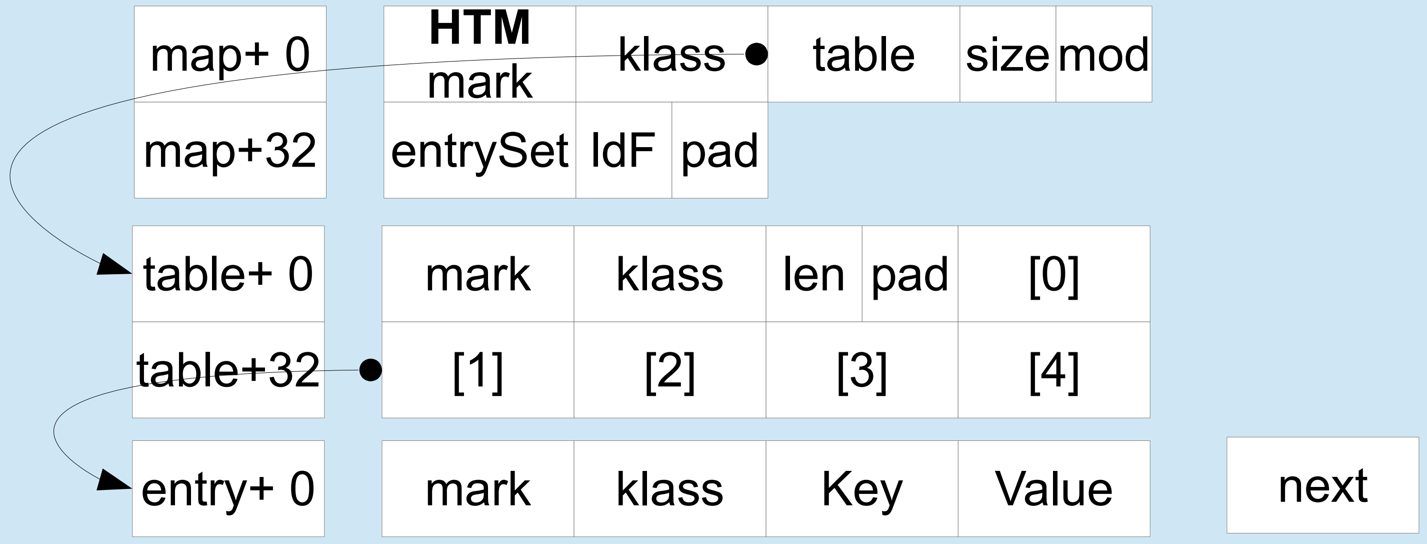


Example!

- Every thread does:
  - synchronize / Java **lock** bytecode
  - Loads map, map.table & table.length
  - Calls key.hashCode() & mod/mask to table.length
  - Index into table[hash], Load Entry
  - Compare Entry.Key; Read or Write value
  - Java **unlock** bytecode

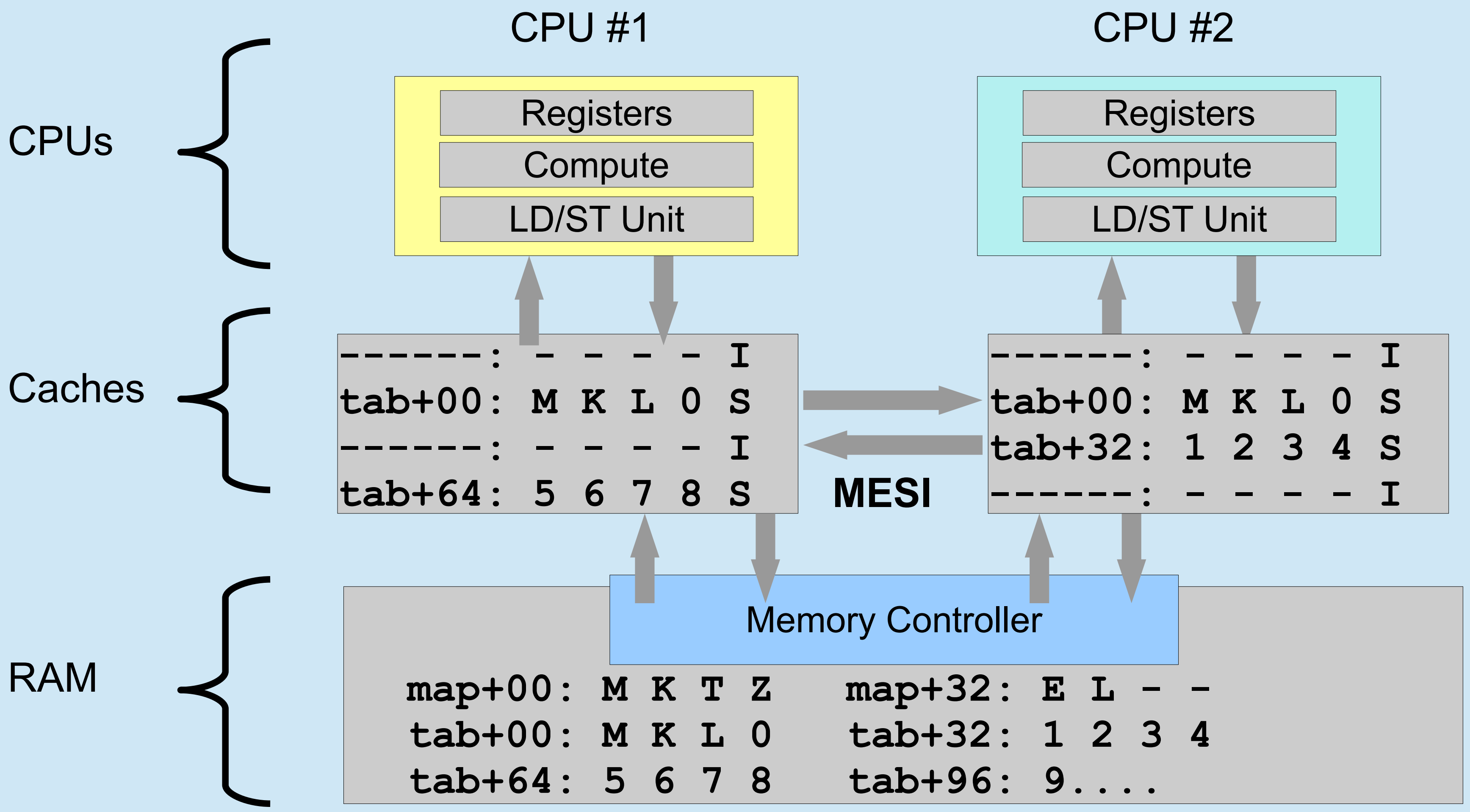
Example!

- Lock is flagged for HTM
- T1 (reader) starts into HashMap
  - Starts a XTN



  - Meanwhile…
- T2 (writer) also starts into HashMap

Real Chips Are Complicated

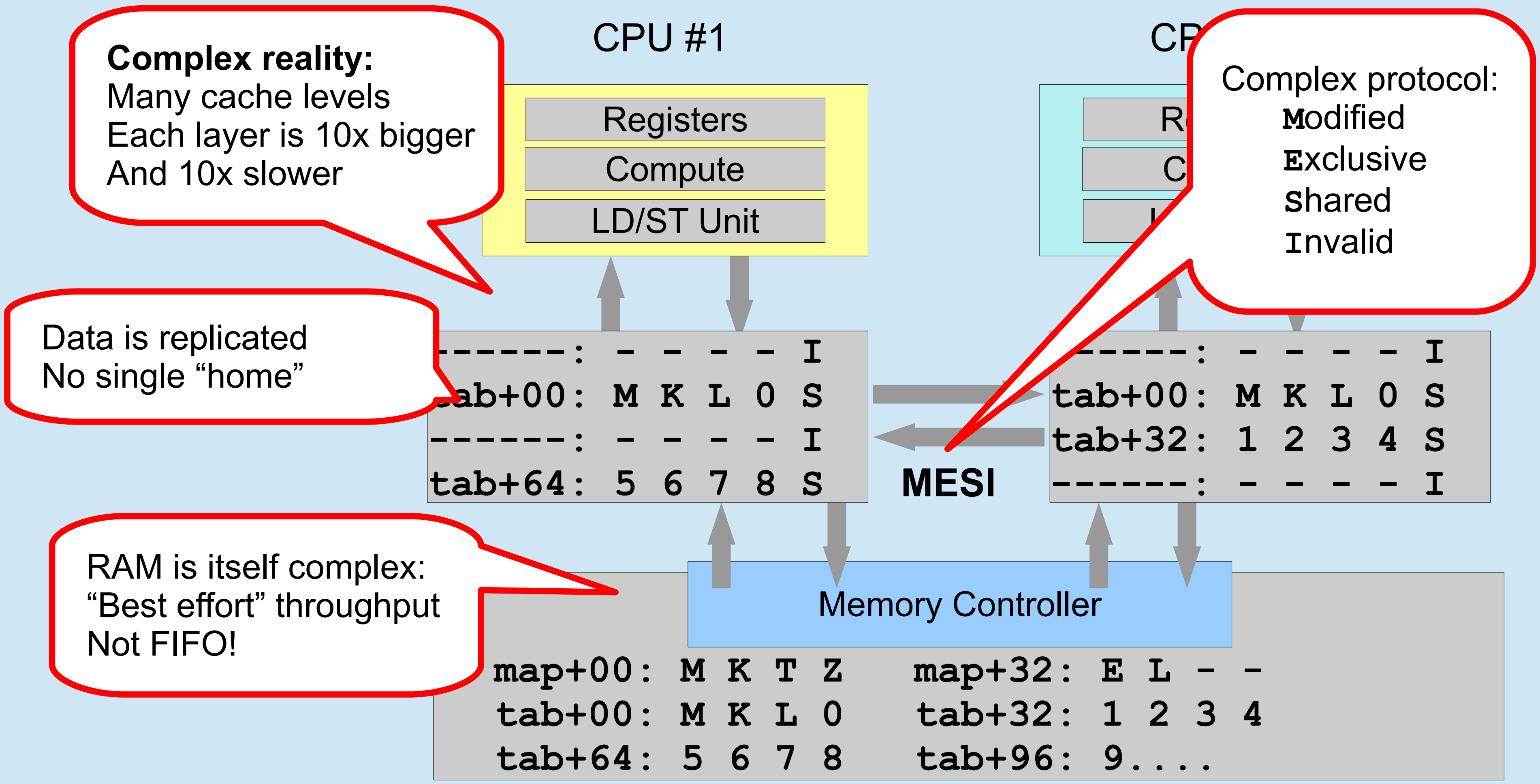


```
------: - - - - I
tab+00: M K L 0 S
------: - - - - I
tab+64: 5 6 7 8 S
```

```
------: - - - - I
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

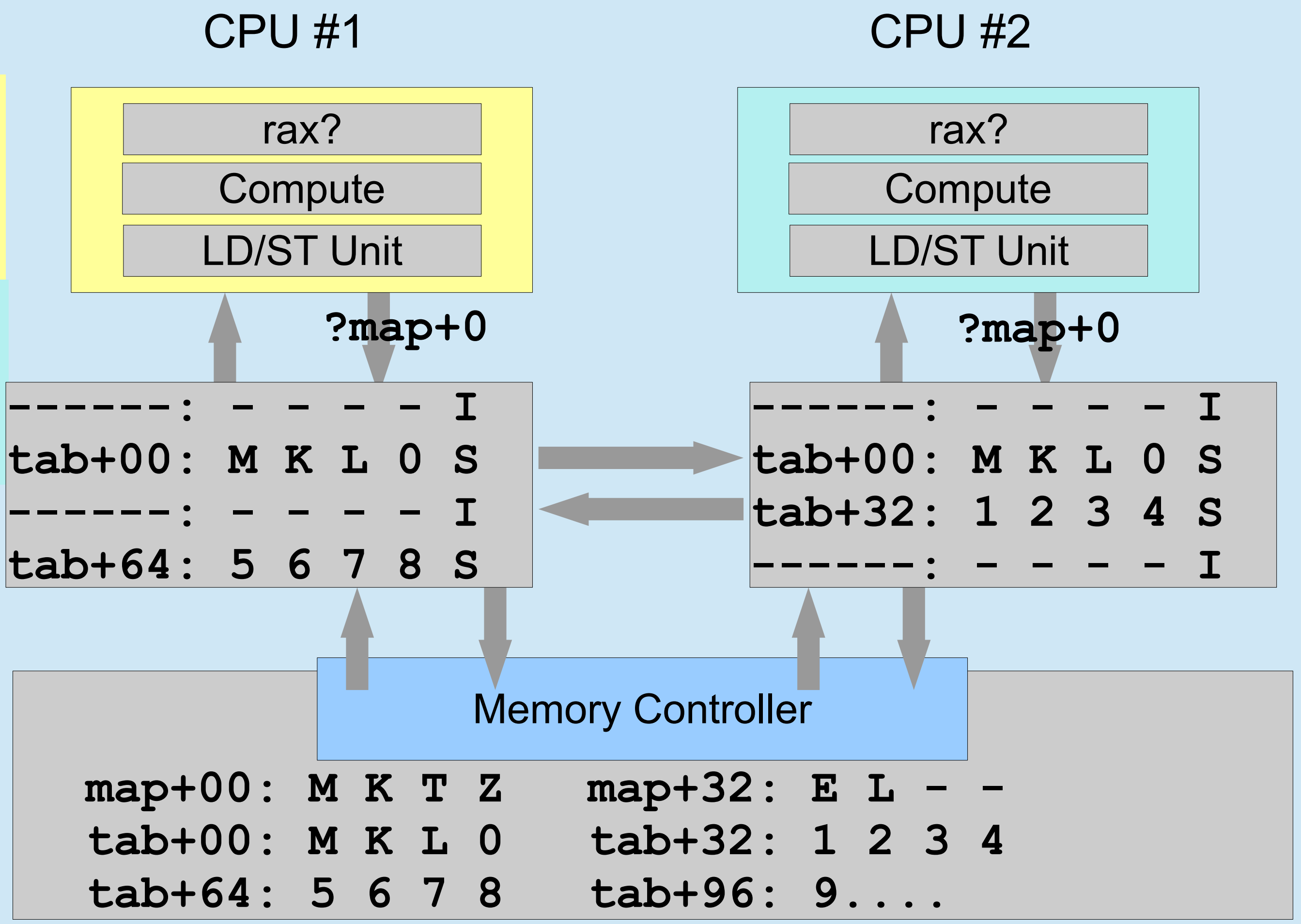
Real Chips Are Complicated



Real Chips Are Complicated

```
ld rMark,map+0
beq rMark,use_HTM
```

```
ld rMark,map+0
beq rMark,use_HTM
```



```
------: - - - - I
tab+00: M K L 0 S
------: - - - - I
tab+64: 5 6 7 8 S
```

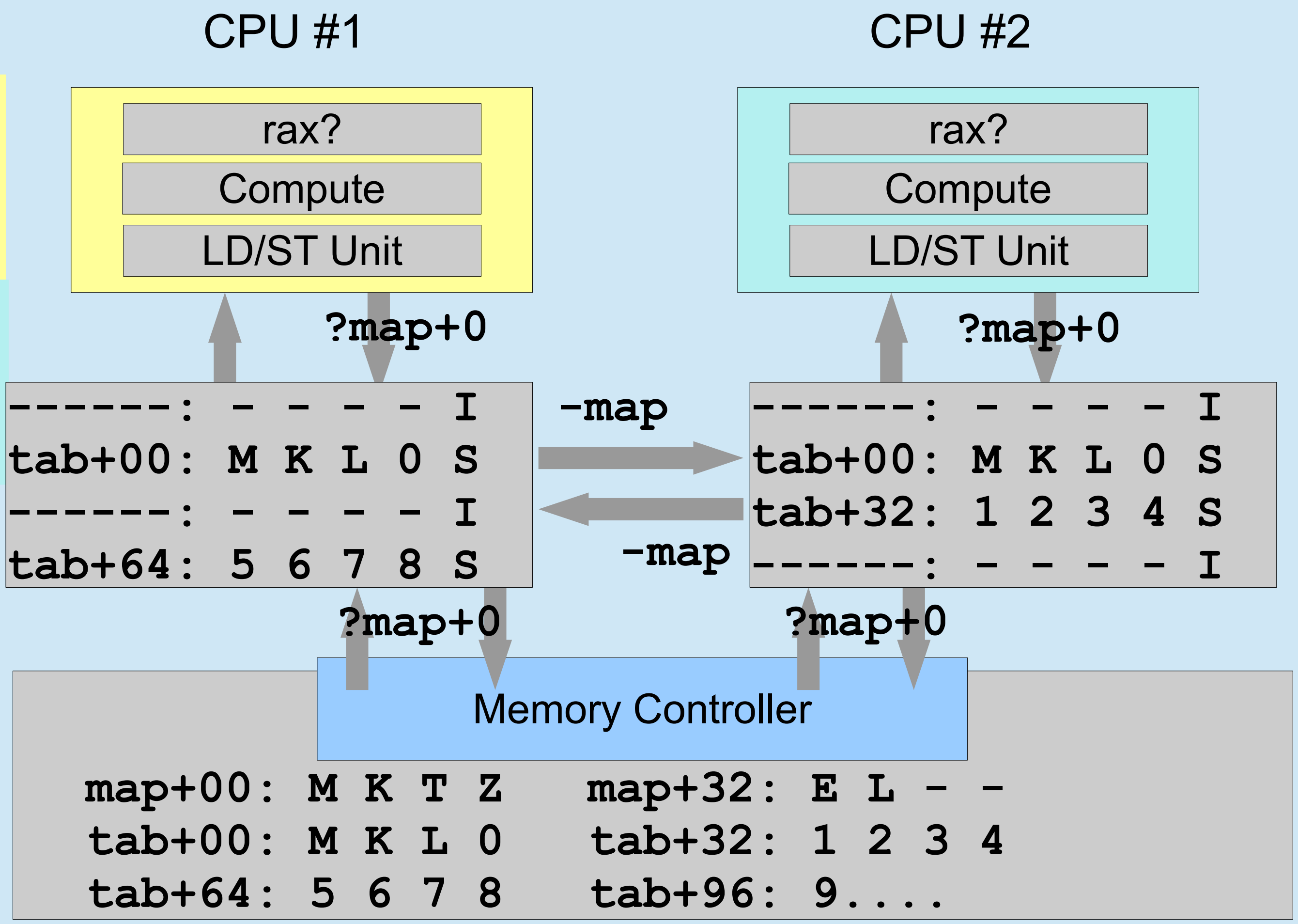
```
------: - - - - I
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
beq rMark,use_HTM
```

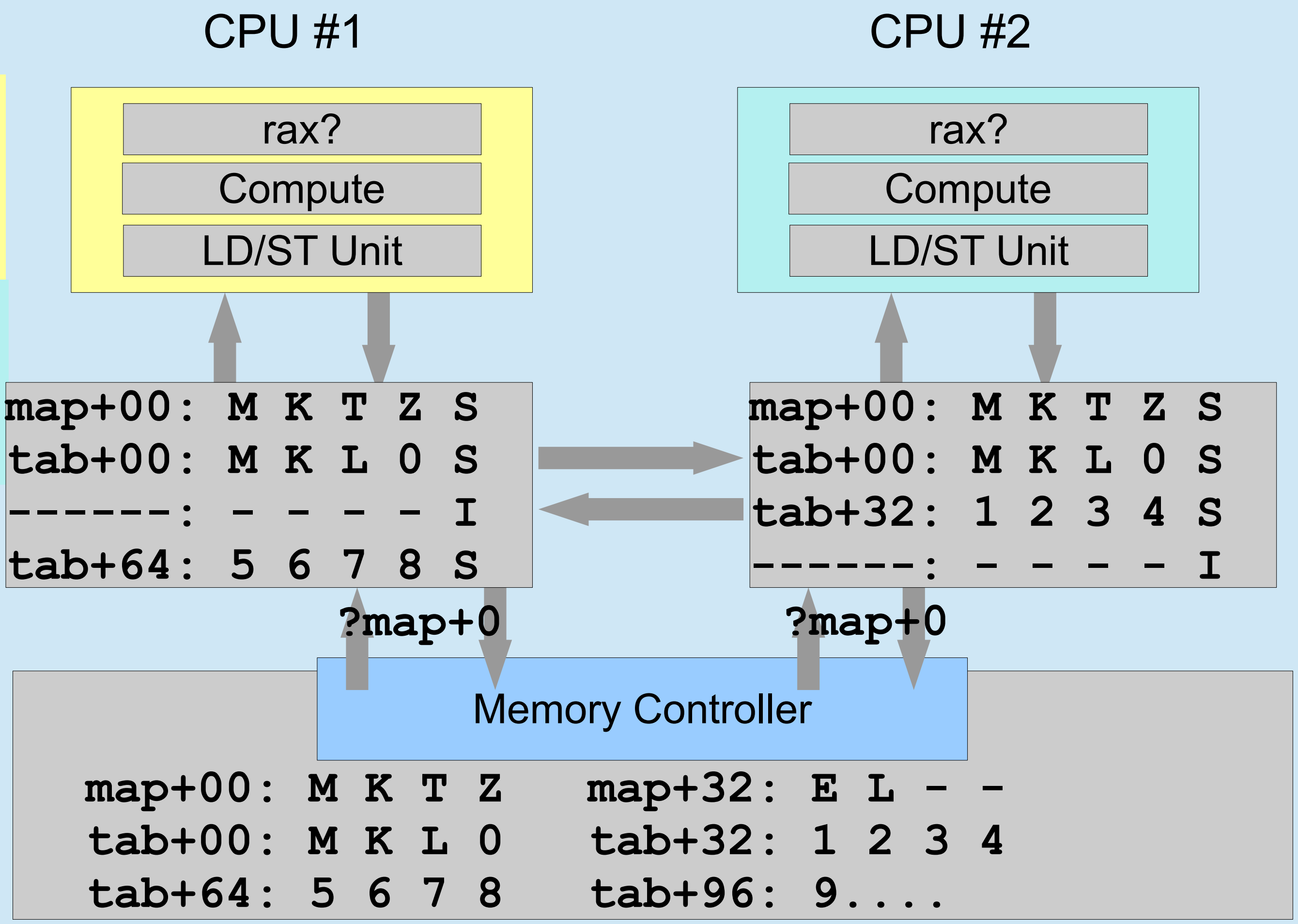
```
ld rMark,map+0
beq rMark,use_HTM
```



Real Chips Are Complicated

```
ld rMark,map+0
beq rMark,use_HTM
```

```
ld rMark,map+0
beq rMark,use_HTM
```



```
map+00: M K T Z S
tab+00: M K L 0 S
------: - - - - I
tab+64: 5 6 7 8 S
```

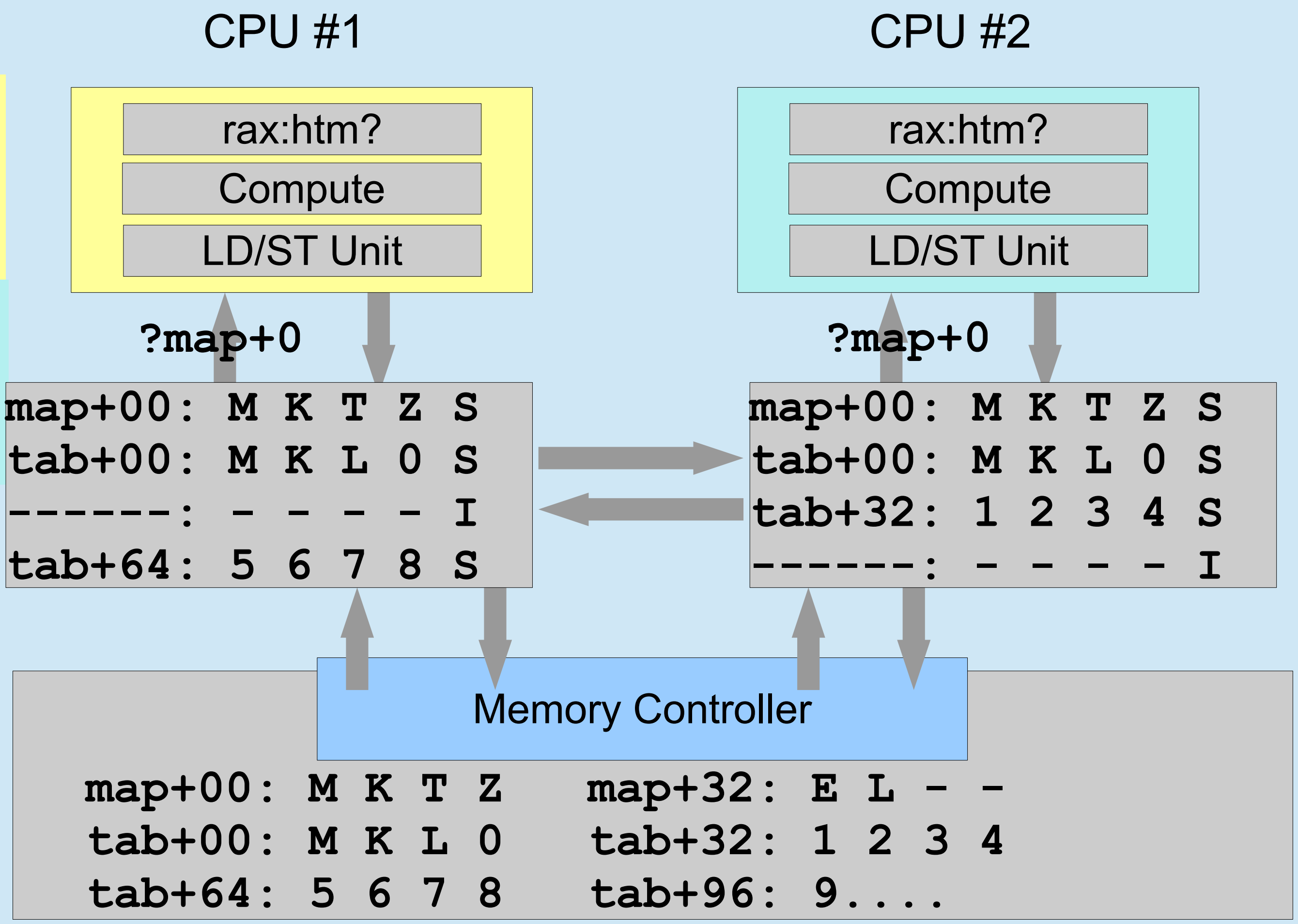
```
map+00: M K T Z S
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
beq rMark,use_HTM
```

```
ld rMark,map+0
beq rMark,use_HTM
```



```
map+00: M K T Z S
tab+00: M K L 0 S
-------: - - - - I
tab+64: 5 6 7 8 S
```

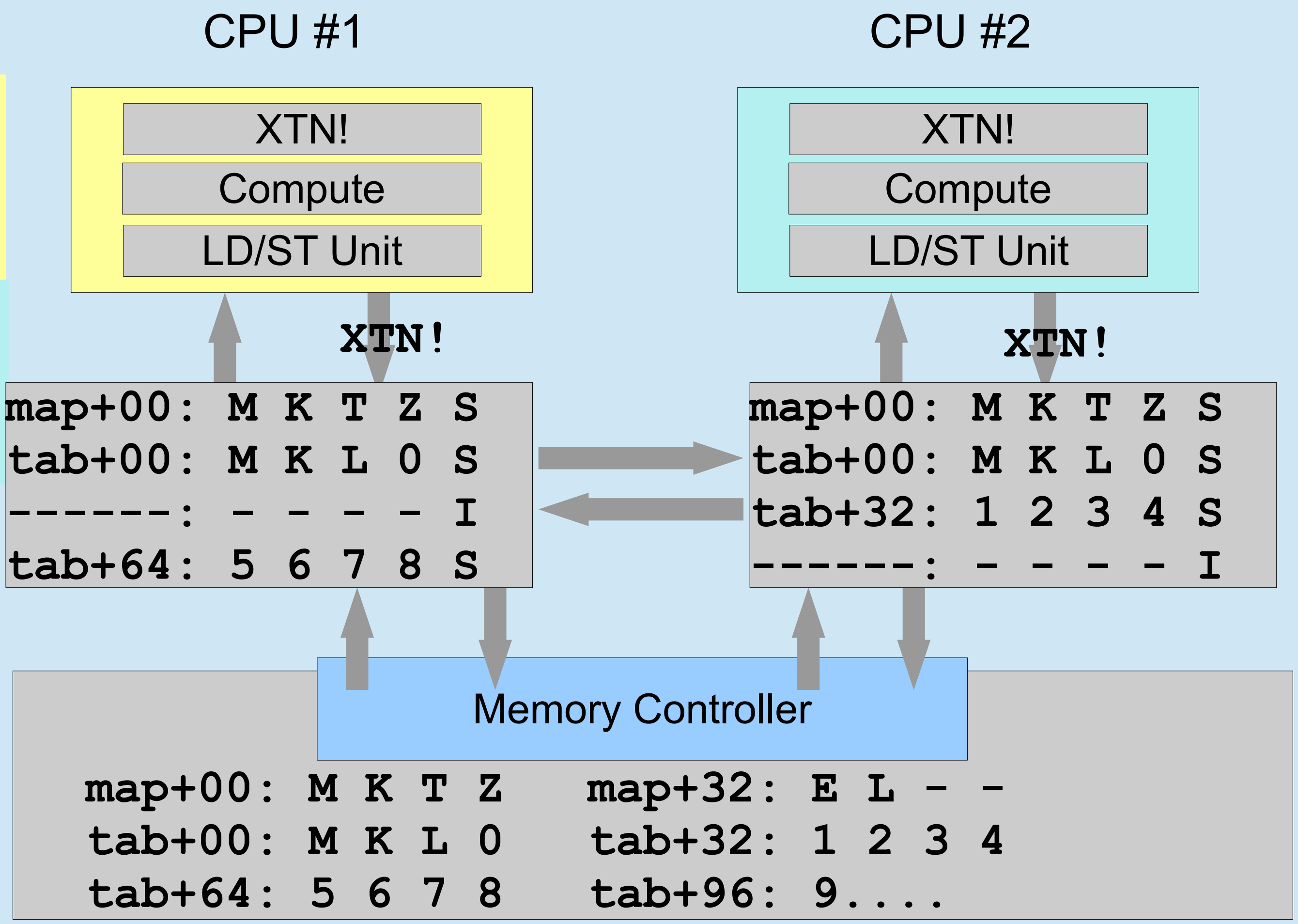
```
map+00: M K T Z S
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
-------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
beq rMark,use_HTM
...
XTN!
```

```
ld rMark,map+0
beq rMark,use_HTM
...
XTN!
```



```
map+00: M K T Z S
tab+00: M K L 0 S
-------: - - - - I
tab+64: 5 6 7 8 S
```

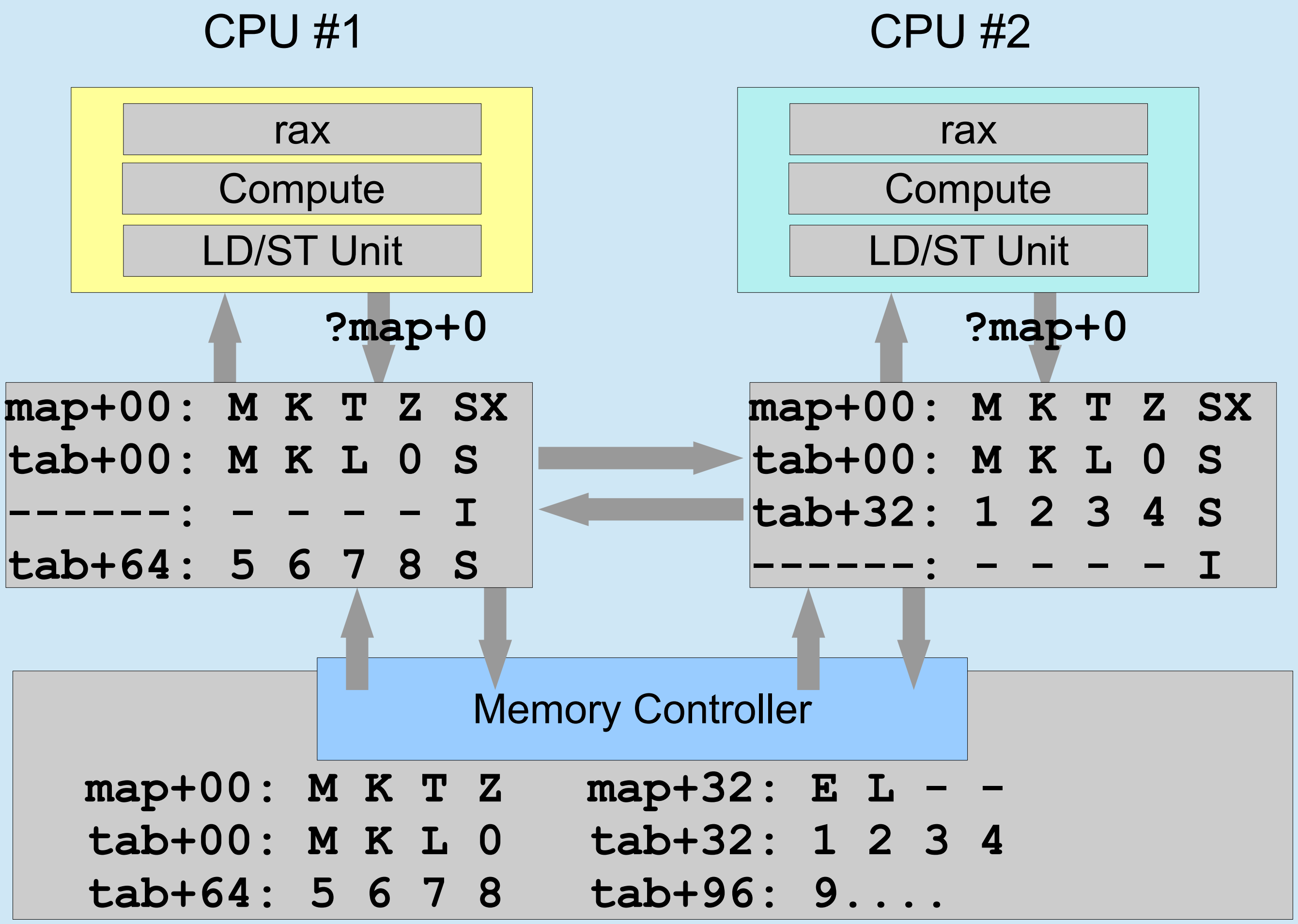
```
map+00: M K T Z S
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
-------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
```

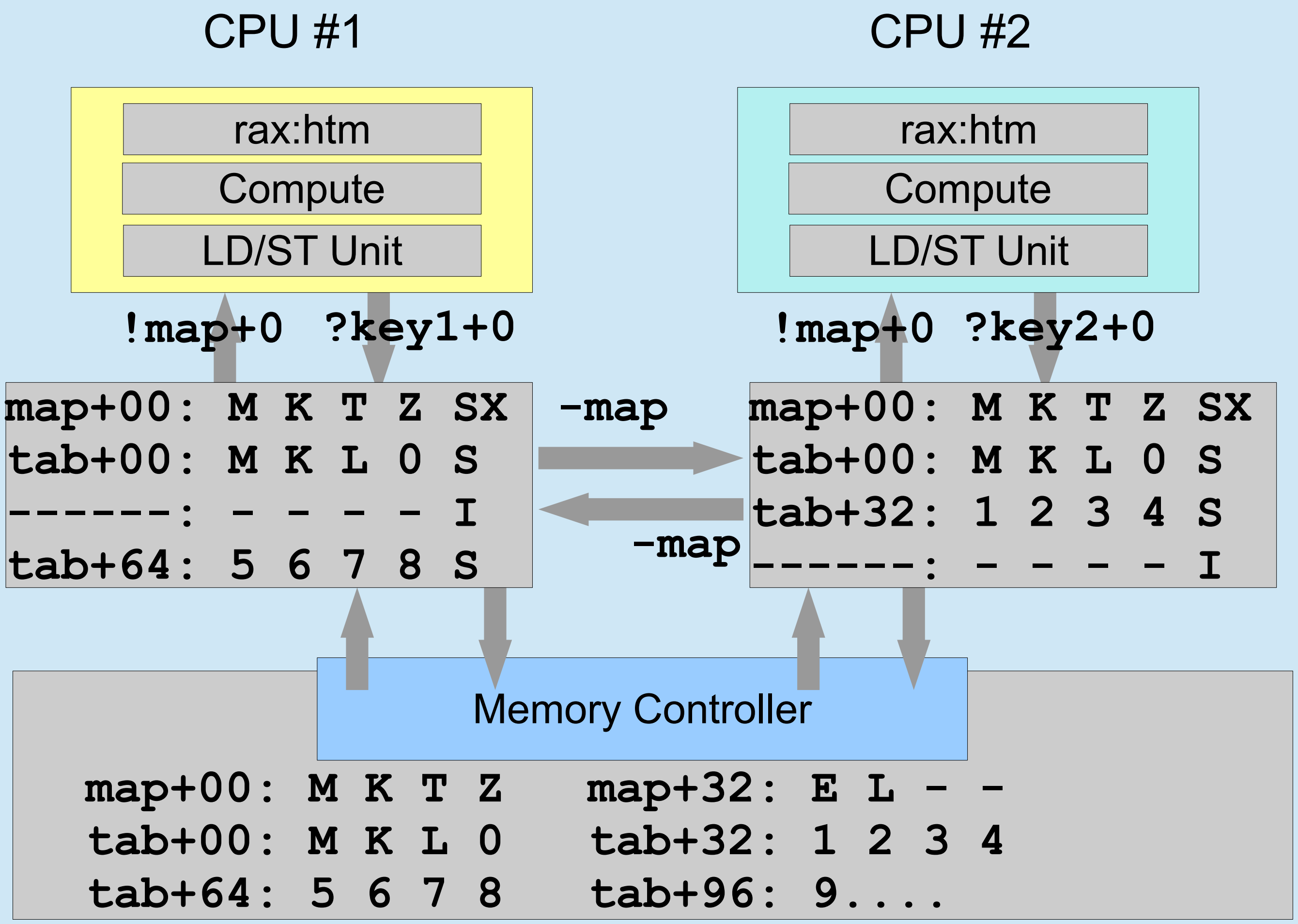
```
ld rMark,map+0
```



Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
```

```
ld rMark,map+0
ld rHash,key2.hash
```



```
map+00: M K T Z SX
tab+00: M K L 0 S
------: - - - - I
tab+64: 5 6 7 8 S
```

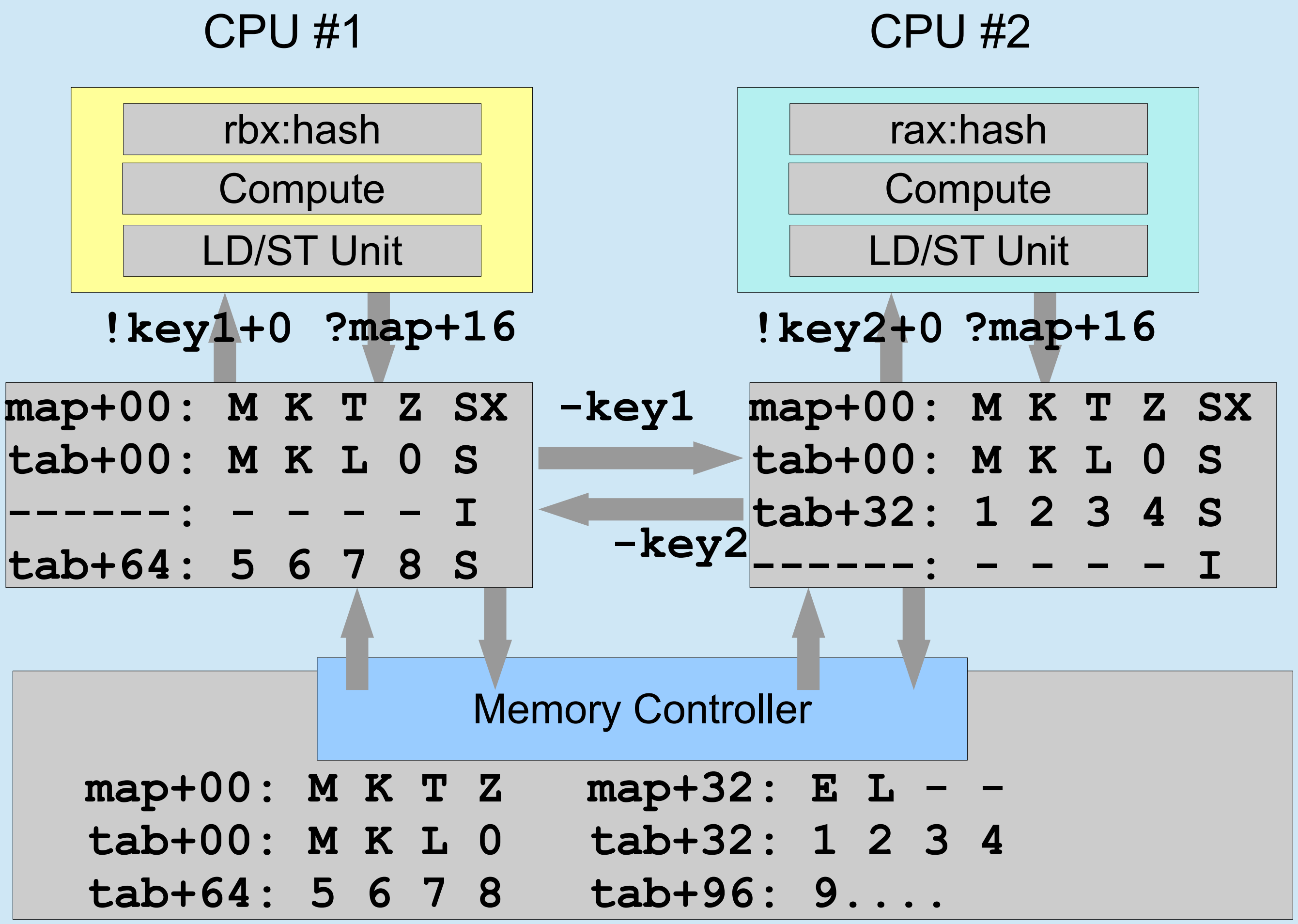
```
map+00: M K T Z SX
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
```

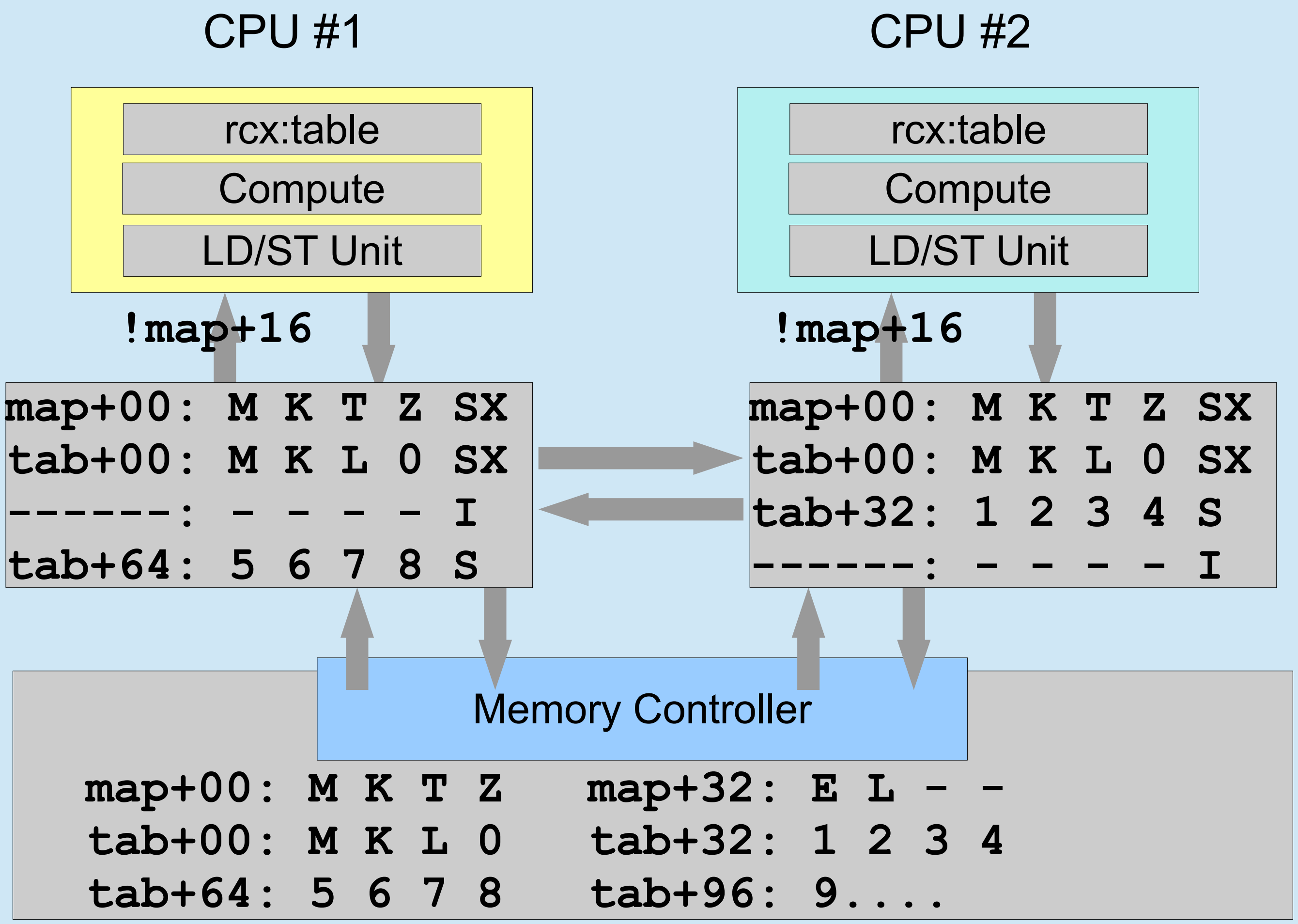
```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
```



Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
```

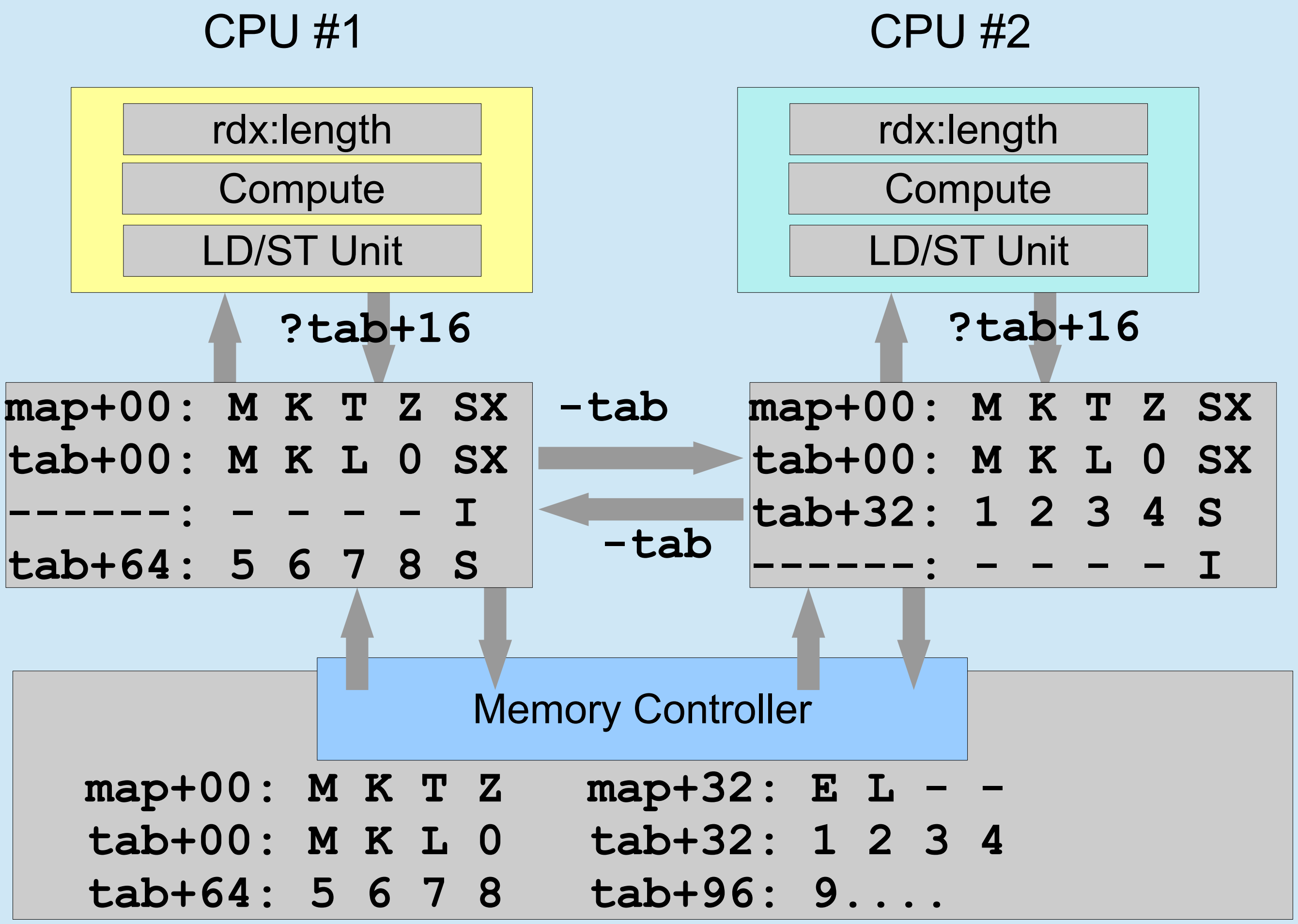
```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
```



Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
```



```
map+00: M K T Z SX
tab+00: M K L 0 SX
------: - - - - I
tab+64: 5 6 7 8 S
```

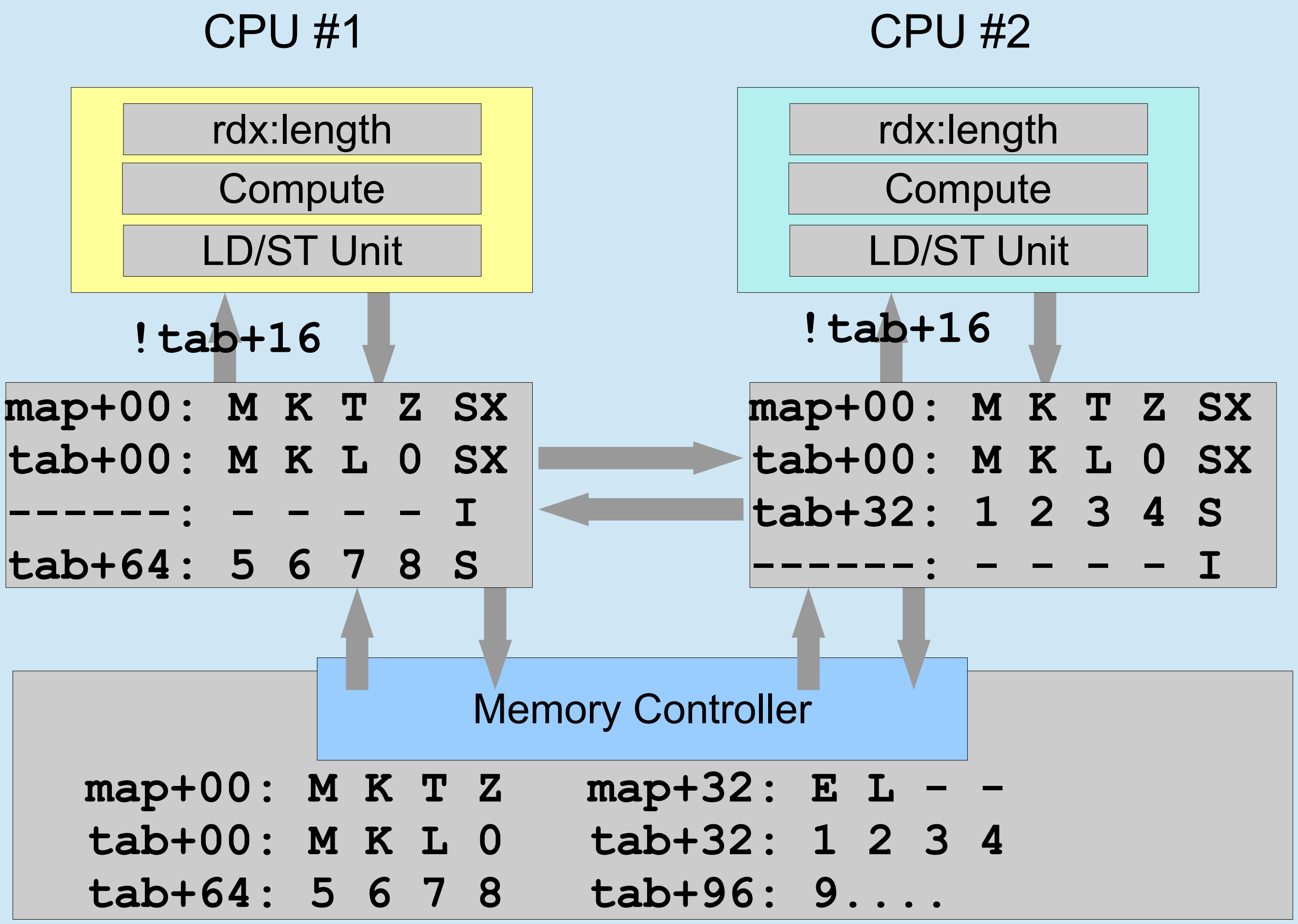
```
map+00: M K T Z SX
tab+00: M K L 0 SX
tab+32: 1 2 3 4 S
------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
```



```
map+00:  M  K  T  Z  SX
tab+00:  M  K  L  0  SX
------:  -  -  -  -  I
tab+64:  5  6  7  8  S
```

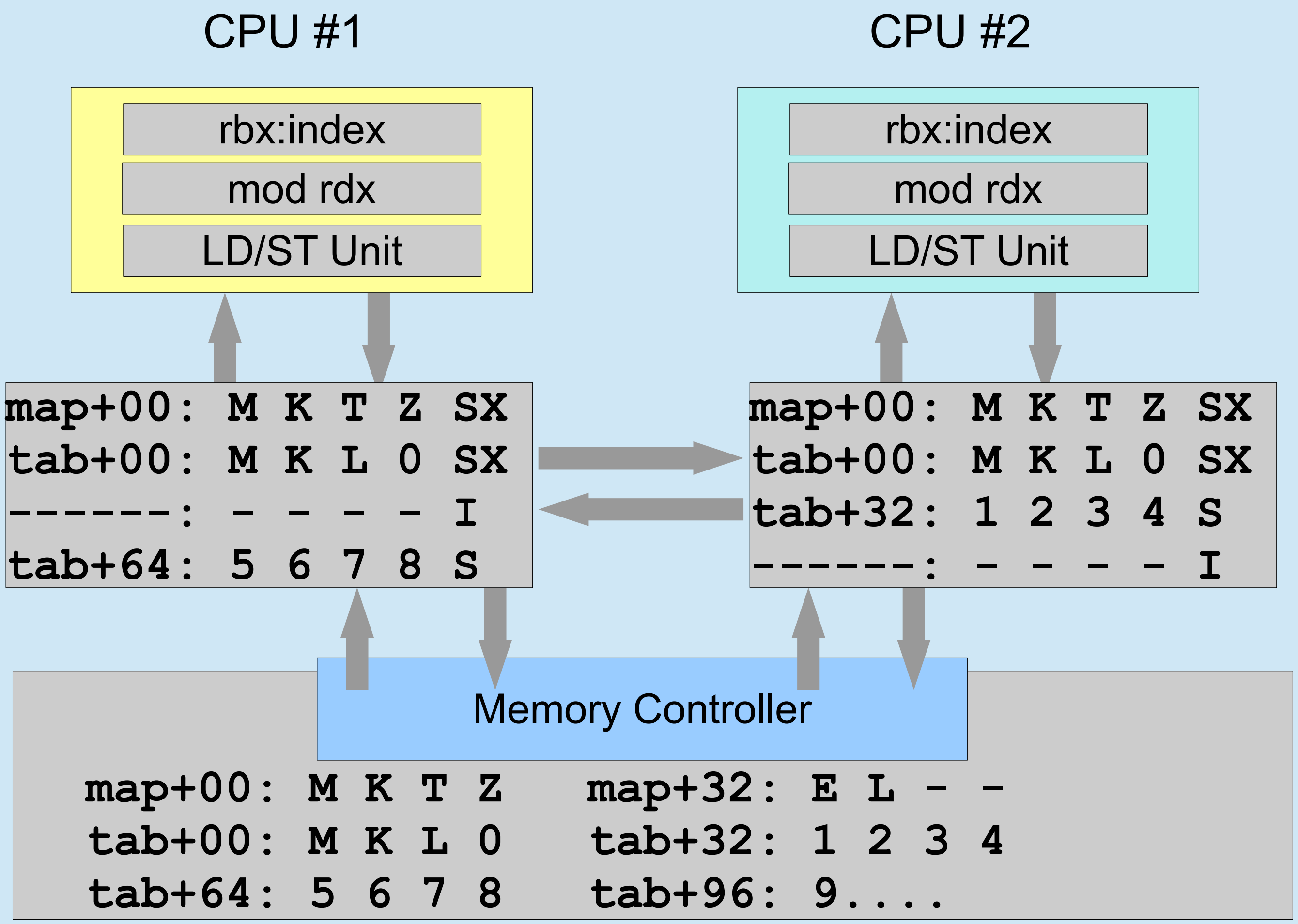
```
map+00:  M  K  T  Z  SX
tab+00:  M  K  L  0  SX
tab+32:  1  2  3  4  S
------:  -  -  -  -  I
```

```
map+00:  M  K  T  Z    map+32:  E  L  -  -
tab+00:  M  K  L  0    tab+32:  1  2  3  4
tab+64:  5  6  7  8    tab+96:  9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
```



```
map+00:  M  K  T  Z  SX
tab+00:  M  K  L  0  SX
-------:  -  -  -  -  I
tab+64:  5  6  7  8  S
```

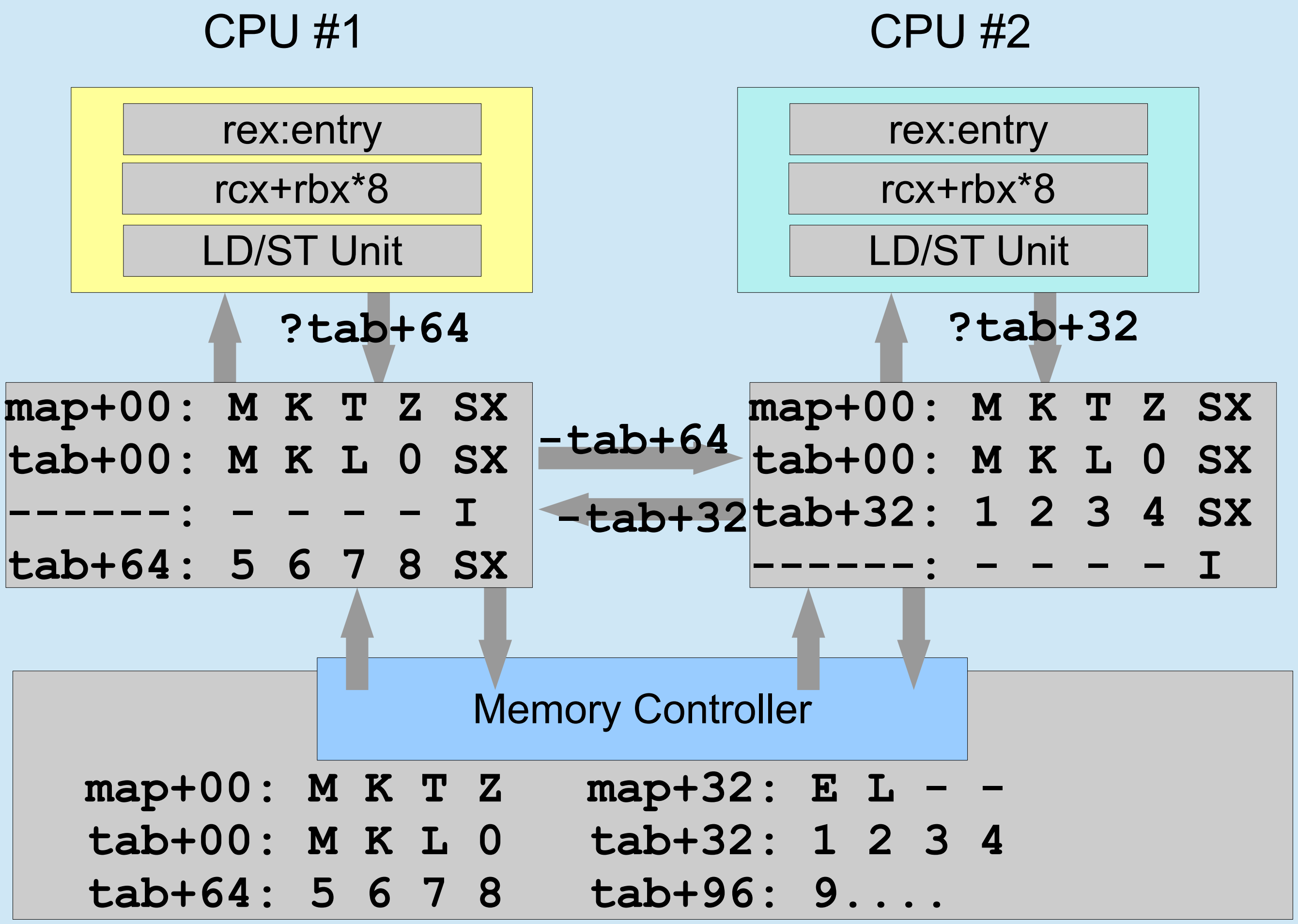
```
map+00:  M  K  T  Z  SX
tab+00:  M  K  L  0  SX
tab+32:  1  2  3  4  S
-------:  -  -  -  -  I
```

```
map+00:  M  K  T  Z    map+32:  E  L  -  -
tab+00:  M  K  L  0    tab+32:  1  2  3  4
tab+64:  5  6  7  8    tab+96:  9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
```



```
map+00: M K T Z SX
tab+00: M K L 0 SX
------: - - - - I
tab+64: 5 6 7 8 SX
```

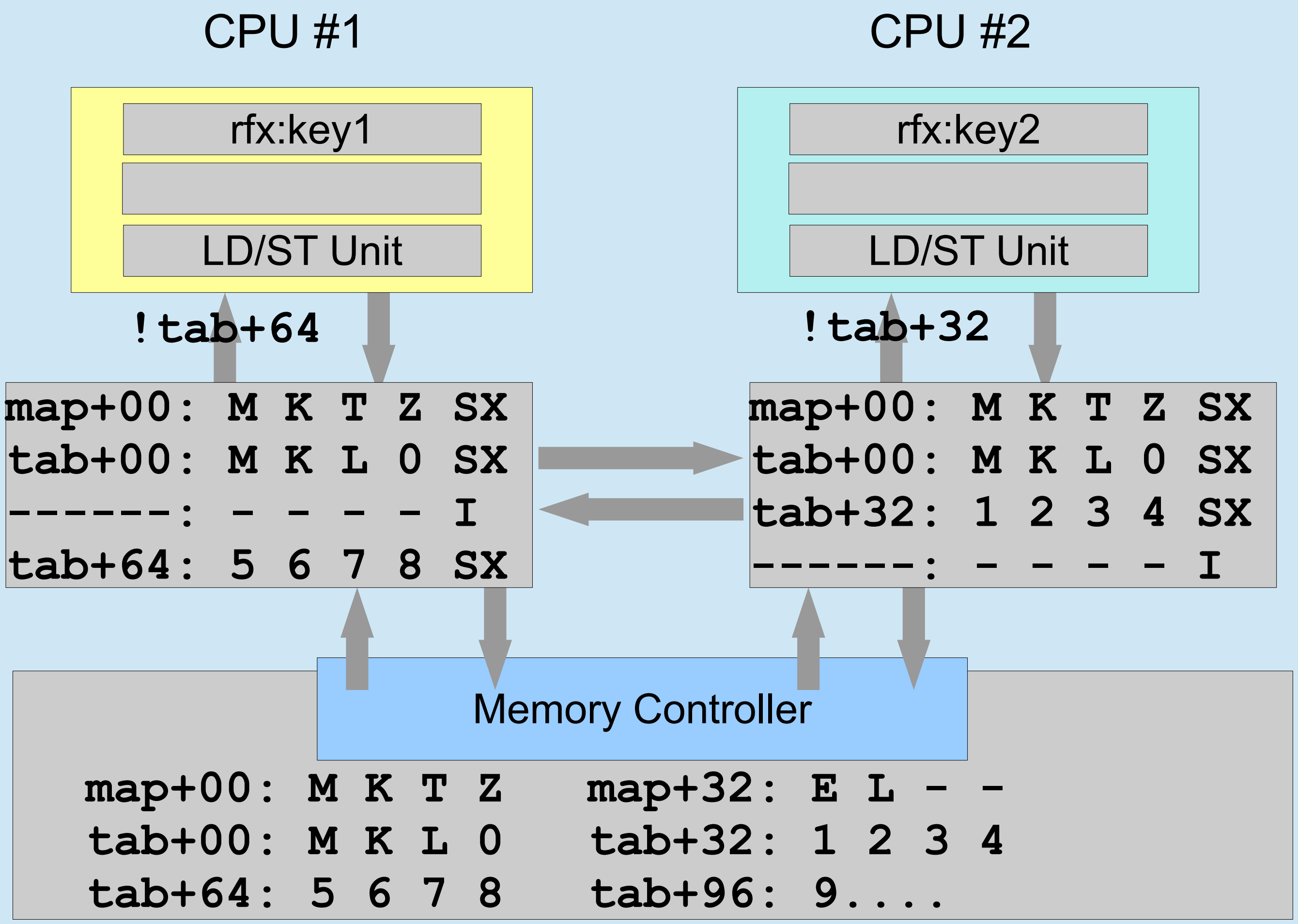
```
map+00: M K T Z SX
tab+00: M K L 0 SX
tab+32: 1 2 3 4 SX
------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
ld rKey,rEntry.key
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
ld rKey,rEntry.key
```



```
map+00: M K T Z SX
tab+00: M K L 0 SX
-------: - - - - I
tab+64: 5 6 7 8 SX
```

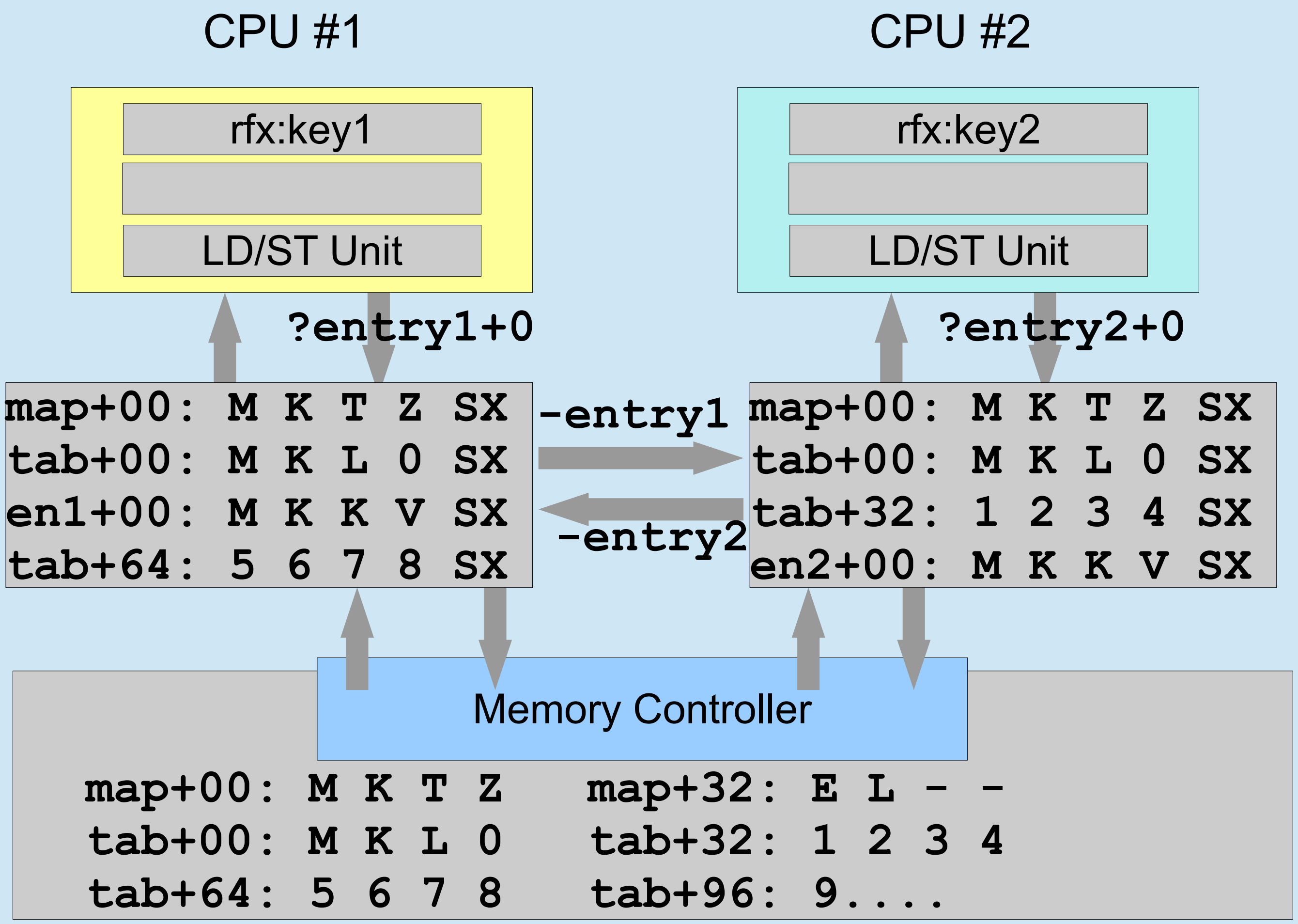
```
map+00: M K T Z SX
tab+00: M K L 0 SX
tab+32: 1 2 3 4 SX
-------: - - - - I
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
ld rKey,rEntry.key
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
ld rKey,rEntry.key
```



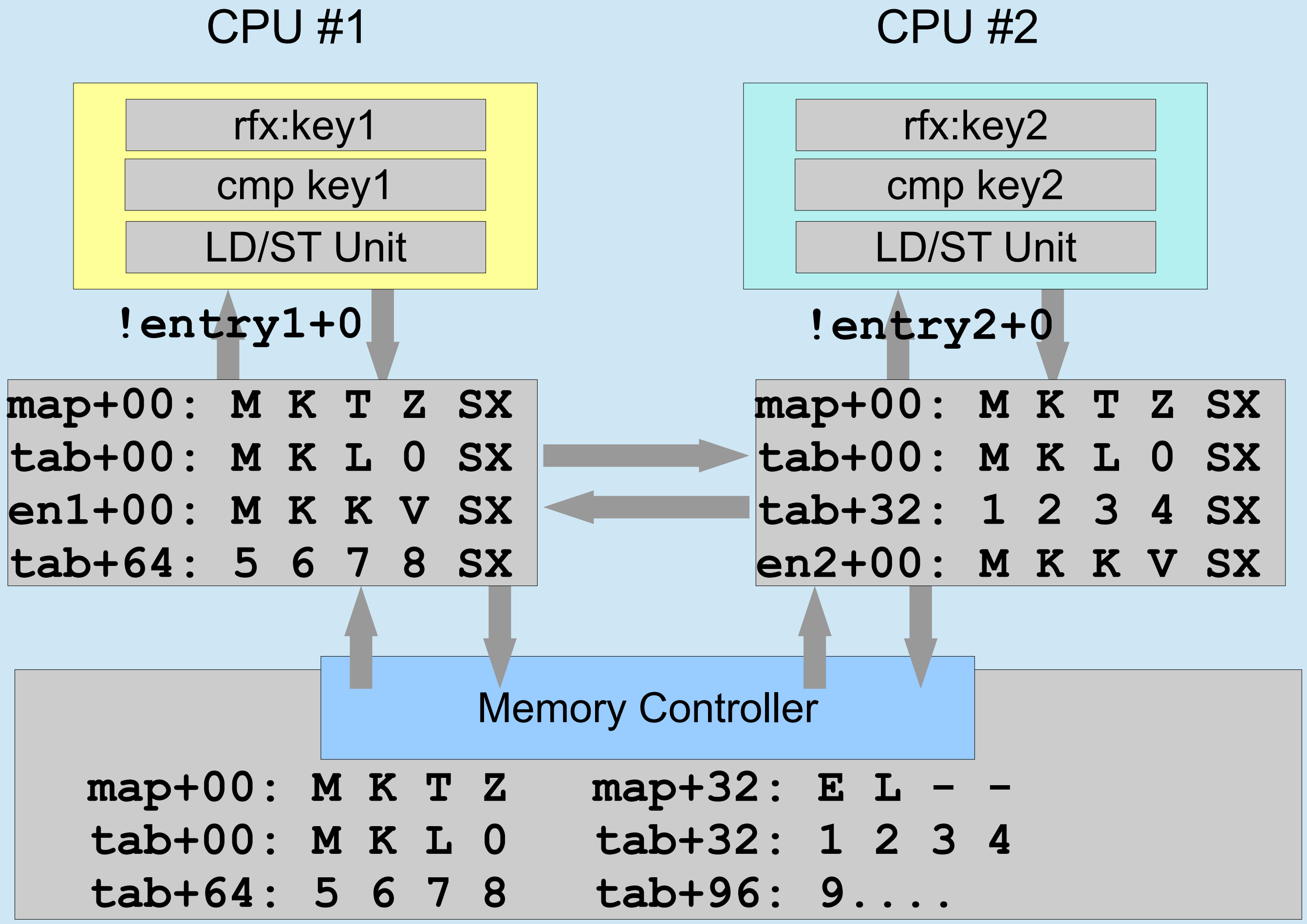
```
map+00: M K T Z SX        map+00: M K T Z SX
tab+00: M K L 0 SX        tab+00: M K L 0 SX
en1+00: M K K V SX        tab+32: 1 2 3 4 SX
tab+64: 5 6 7 8 SX        en2+00: M K K V SX
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
ld rKey,rEntry.key
bne rKey,key1,reprobe
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
ld rKey,rEntry.key
bne rKey,key2,reprobe
```



```
map+00: M K T Z SX
tab+00: M K L 0 SX
en1+00: M K K V SX
tab+64: 5 6 7 8 SX
```

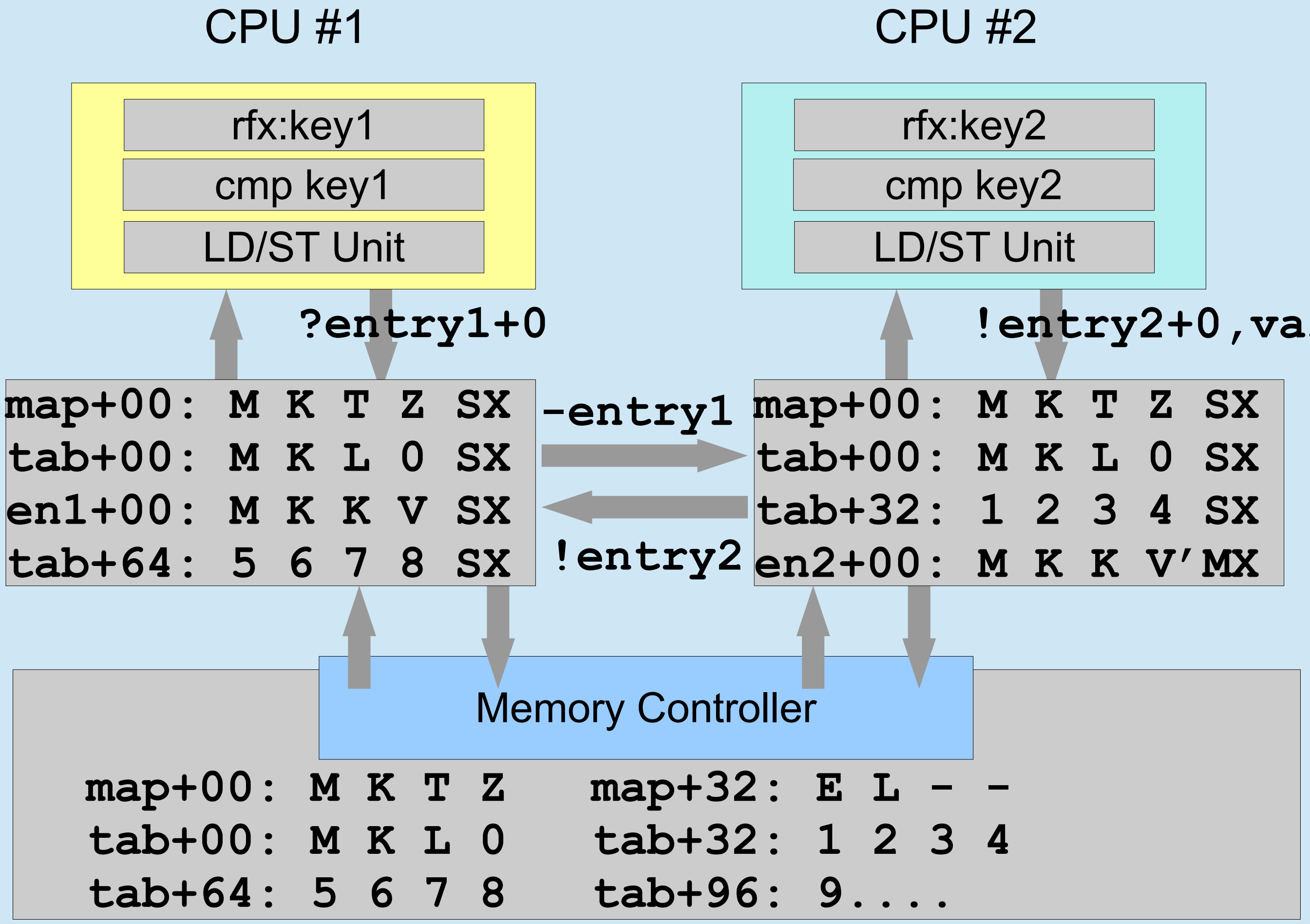
```
map+00: M K T Z SX
tab+00: M K L 0 SX
tab+32: 1 2 3 4 SX
en2+00: M K K V SX
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
ld rKey,rEntry.key
bne rKey,key1,reprobe
ld rVal,rEntry.val
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
ld rKey,rEntry.key
bne rKey,key2,reprobe
st rVal,rEntry.val
```



```
map+00: M K T Z SX
tab+00: M K L 0 SX
en1+00: M K K V SX
tab+64: 5 6 7 8 SX
```

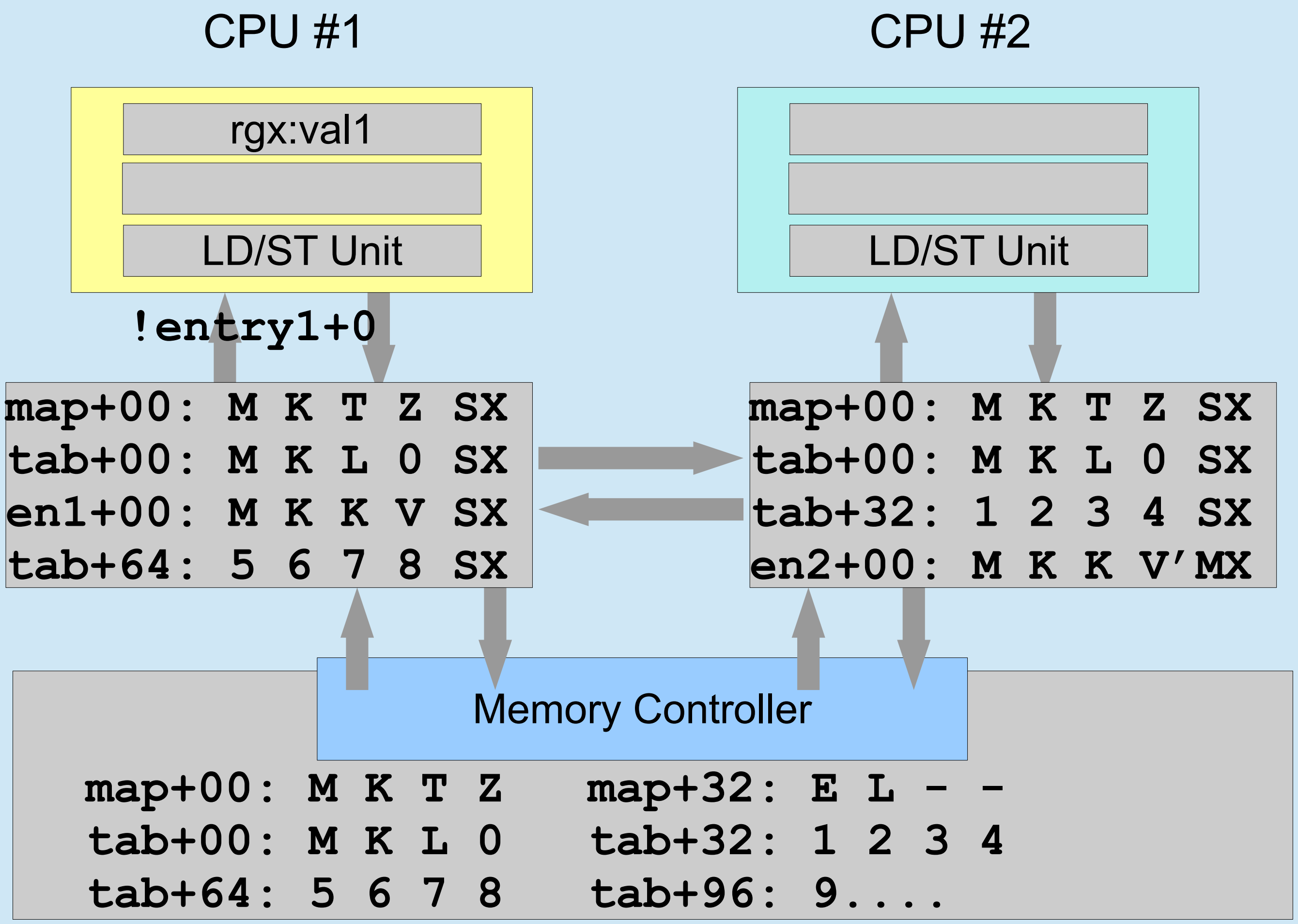
```
map+00: M K T Z SX
tab+00: M K L 0 SX
tab+32: 1 2 3 4 SX
en2+00: M K K V'MX
```

```
map+00: M K T Z   map+32: E L - -
tab+00: M K L 0   tab+32: 1 2 3 4
tab+64: 5 6 7 8   tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
ld rKey,rEntry.key
bne rKey,key1,reprobe
ld rVal,rEntry.val
XTN commit
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
ld rKey,rEntry.key
bne rKey,key2,reprobe
st rVal,rEntry.val
XTN commit
```



```
map+00: M K T Z SX
tab+00: M K L 0 SX
en1+00: M K K V SX
tab+64: 5 6 7 8 SX
```

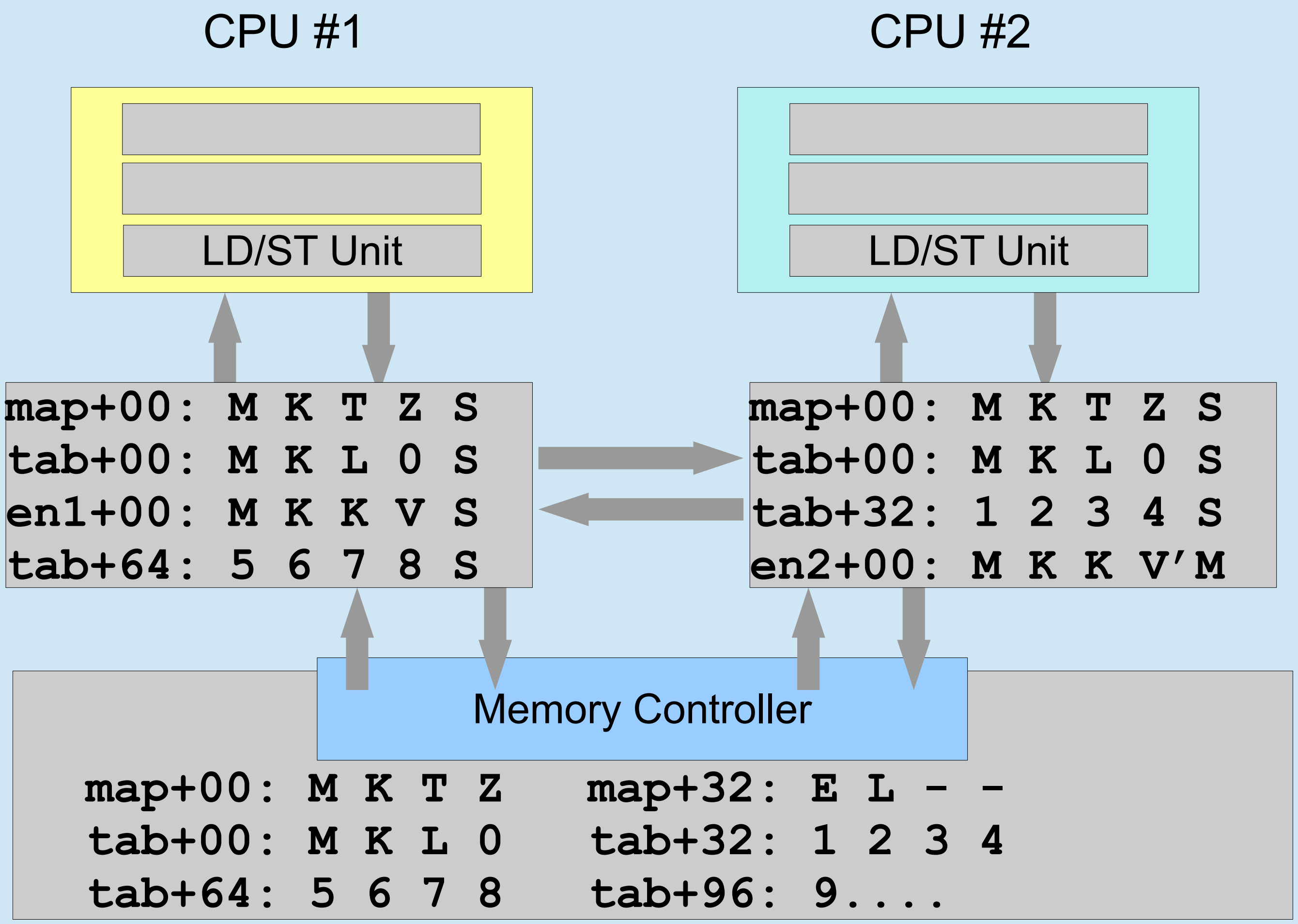
```
map+00: M K T Z SX
tab+00: M K L 0 SX
tab+32: 1 2 3 4 SX
en2+00: M K K V'MX
```

```
map+00: M K T Z   map+32: E L - -
tab+00: M K L 0   tab+32: 1 2 3 4
tab+64: 5 6 7 8   tab+96: 9....
```

Real Chips Are Complicated

```
ld rMark,map+0
ld rHash,key1.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry,rTab[rHash]
ld rKey,rEntry.key
bne rKey,key1,reprobe
ld rVal,rEntry.val
XTN commit
```

```
ld rMark,map+0
ld rHash,key2.hash
ld rTab,map+16
ld rLen,rTab+16
mod rHash,rLen
ld rEntry.rTab[rHash]
ld rKey,rEntry.key
bne rKey,key2,reprobe
st rVal,rEntry.val
XTN commit
```



```
map+00: M K T Z S
tab+00: M K L 0 S
en1+00: M K K V S
tab+64: 5 6 7 8 S
```

```
map+00: M K T Z S
tab+00: M K L 0 S
tab+32: 1 2 3 4 S
en2+00: M K K V'M
```

```
map+00: M K T Z    map+32: E L - -
tab+00: M K L 0    tab+32: 1 2 3 4
tab+64: 5 6 7 8    tab+96: 9....
```

Example Is Too Simple

- Showing: 4 cache lines, 6 lds, +1 ld/st
- More if Key.hashCode() needs to be computed
- More if Key.equals() is a v-call
- More if need to reprobe, even a little
- More to touch modcount & check CME
- More to check load factor vs size
  - Many many more if resize…
- Typical successful read-only HashMap is ~15-20 **lines** – not loads. Can be lots more!

Lesson: HTM needs many lines!

- Handling a small count of lines is not enough
  - At least for generic Java code
  - Dozen probably fine for tiny hand-crafted cases
  - Or for very carefully crafted Java cases
- But you still touch way more lines
  than it looks like
- Map, table, Entry, Key, Value plus their vtables, plus some reprobes & extra Entry, Keys, plus whats needed to run key-compare, plus stack footprint, plus...

But HTM Fails for True Contention

- Two threads touch same **cache line** and at least one writes
  - **False Sharing:** contention on **`modCount`**:
  - Even if all Readers plus only one Writer
    - And Writer in unrelated part of table
  - At least one aborts (generally 1st one)
- Its not just HashMap: Other utilities often have “perf counters” or “mod counter”.
  - e.g. used to detect CME
- Means a single Writer aborts all Readers

Not Just Fails Once...

- Lock was busy & contended
  - Else was using CAS + spin-locks
  - Hence LOTS of Readers and a rare Writer
- Writer aborts, retries...
- But always another Reader before Write commits
- And Write aborts all Readers also…
- So rapidly becomes live-lock
  ...and must bail out to an OS blocking lock

HTM Fails for True Contention

- Frequently fails on “Poster Child” case
- But also: Failed to get True Parallelism
- Biz Model: Must be 5x faster than X86
  - Need at least 4 working threads to equal 1 X86…
  - So really need ~20 active working threads where X86 needs only 1
  - Need applications to scale by 20x
- Applications Failed-to-Scale
- Main reason: no “big scale” thinking in code

Fail-To-Scale

- Industry trying to figure out Scaling Model
  - Micro-services in infancy, one of many approaches
  - Azul tried Big Parallel Shared Memory
  - General Nativity on what would work
- Turns out: You Need Discipline to Scale
  - And that Discipline works for distributed memory same as shared memory
- Must manage communication
  - Either via “synchronized” or SaaS / tcp-socket
  - And then do not need big shared-memory machine

Slow Processors

- Ultimately too hard to get 4x threads going
  - Needed to match 1 X86
- Industry pivoted to clusters (e.g. Hadoop) with various streaming models
- Cheaper to organically grow cluster
  - Understandable coding model “in the small”
  - Still see all kinds of problems “in the large!”
  - But cheap hardware to experiment on
  - Incremental perf bug fixes work
- Clusters of cheap distributed X86s win the day

Summary

- $250M spent
- Max ~850core machine
  - Roughly equal to 100 X86's but shared memory
- Lots of fun custom hardware
- HTM: Very aggressive: All of L1
- HTM fails for "stupid" reasons
  - But would require major software rewrite to use
  - Which defeated the purpose
- Azul lives on selling low-latency GC on X86s

Q&A

Cliff Click