Systems@ETH zürich

Exercise Session 13

Computer Architecture and Systems Programming

Virtual Memory

Herbstsemester 2024

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- **Kahoots**: now on website n.ethz.ch/~falkbe/
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Agenda

Systems@ETH zürich

- Virtual Memory: Recap and Quiz
- Lecture Recap: Multiprocessing
  - Symmetric Multiprocessing (SMP)
  - Cache Coherency: MSI, MESI, MOESI, MESIF
  - Memory Consistency Models
  - Sync: TAS, CAS, TTAS/CAS Spinlock
  - NUMA

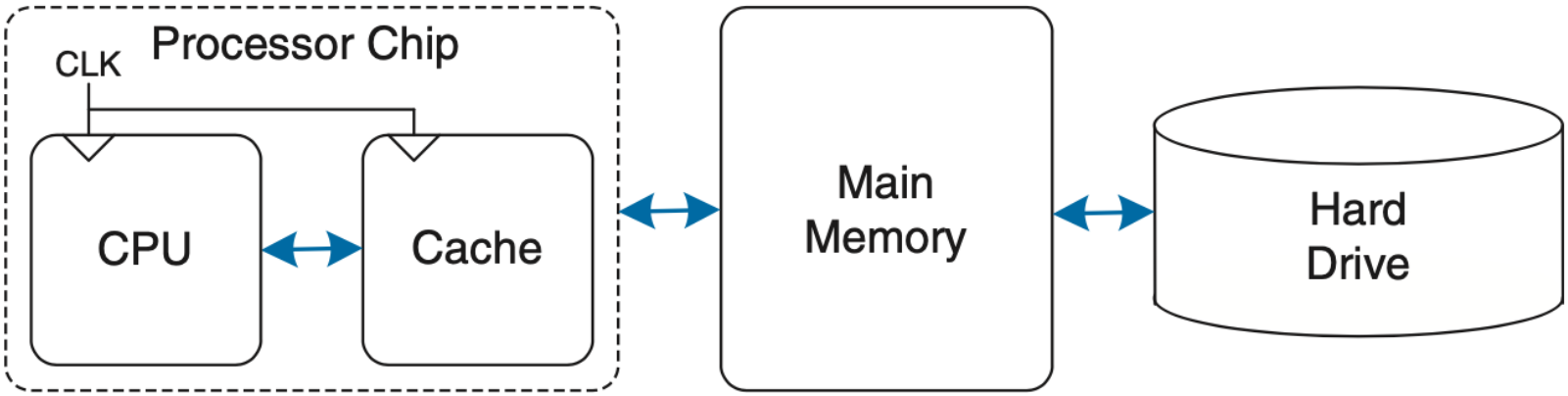
Virtual Memory Recap

Virtual Memory Recap

Caches

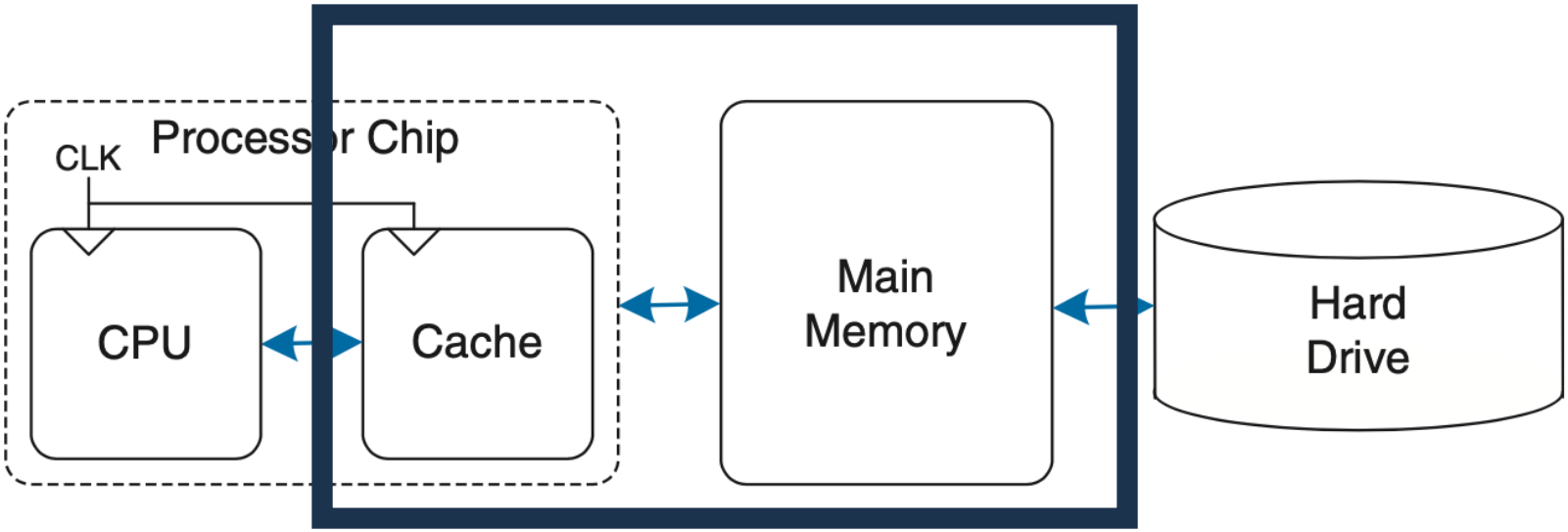
Virtual Memory Recap

- **Memory hiearachy**



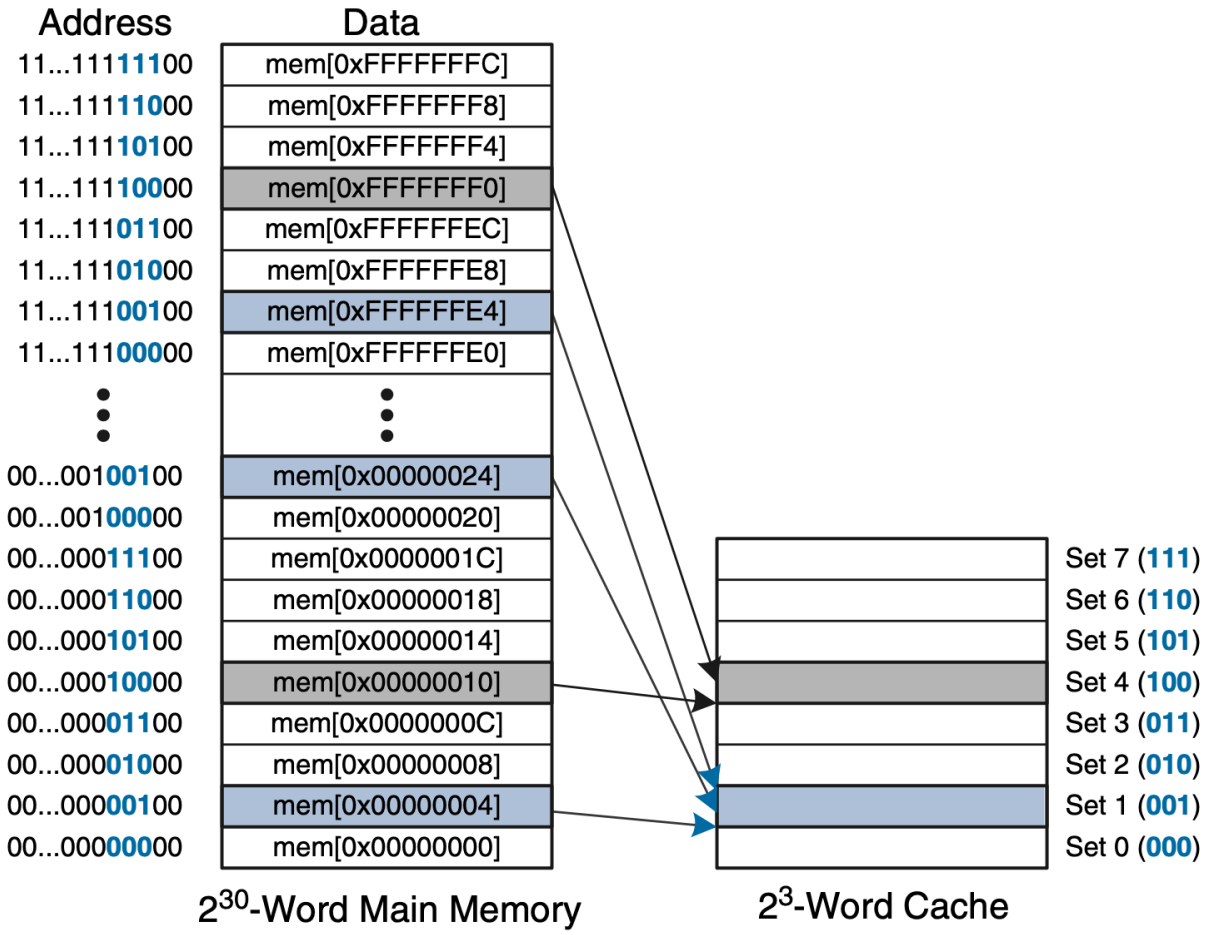
Virtual Memory Recap

- **Memory hiearachy**

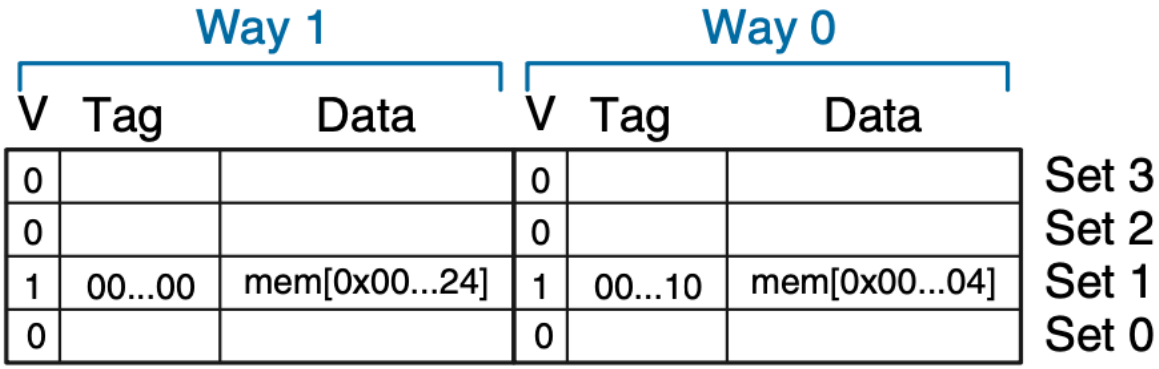


Caches: Direct Mapped

- **Direct mapped**: each set contains one block
- Bottom 2 bits 0 because its word (here 4 byte) aligned
- Next log2(S)=3 bits indicate set (mod 8)

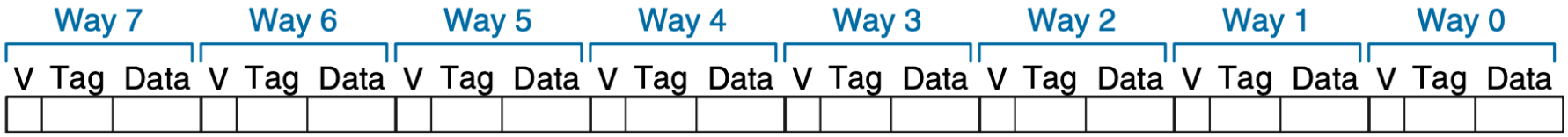


Caches: N-way Set Associative



- **Advantage**: the higher the associativity, the less conflicts we have
  - Set associative caches generally have lower miss rate (only need to evict if both ways are full)

Caches: Fully Associative Cache



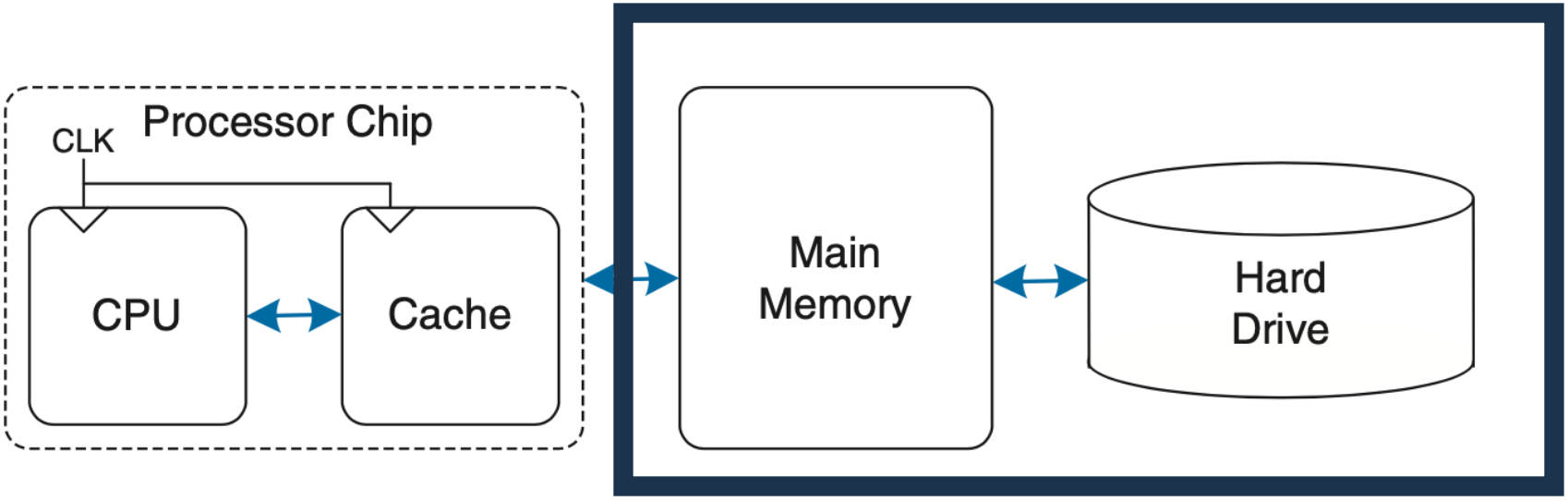
- **Fully associative**: B ways (number of blocks), i.e. no **conflict misses anymore**
- **Issue**: need a lot of comparators (compare 8 values in parallel)

Virtual Memory Recap

Virtual memory

Virtual Memory Recap

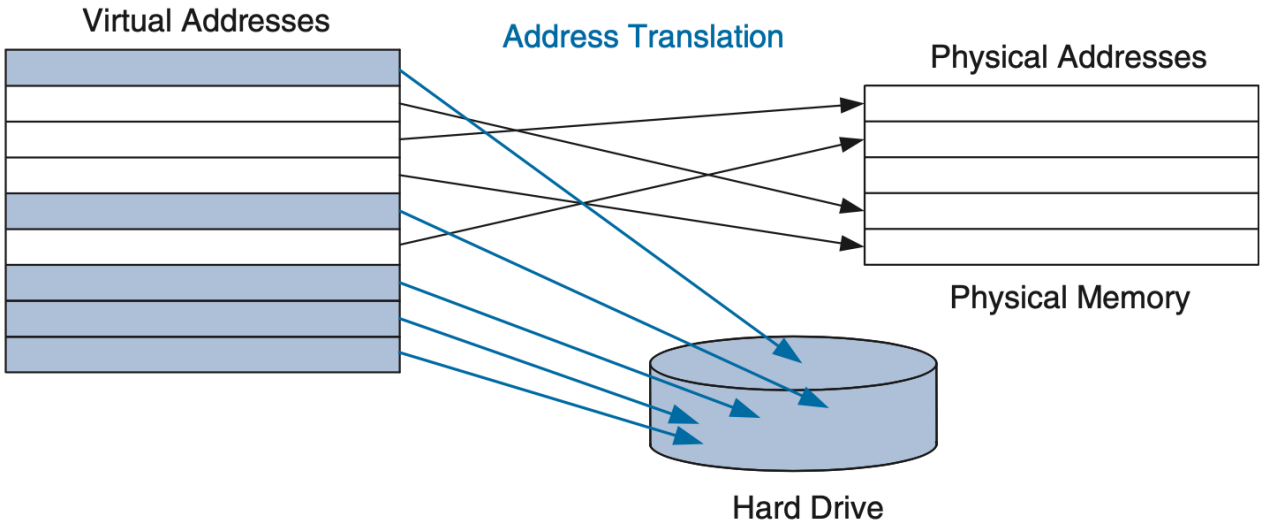
- **Memory hiearachy**



Virtual Memory

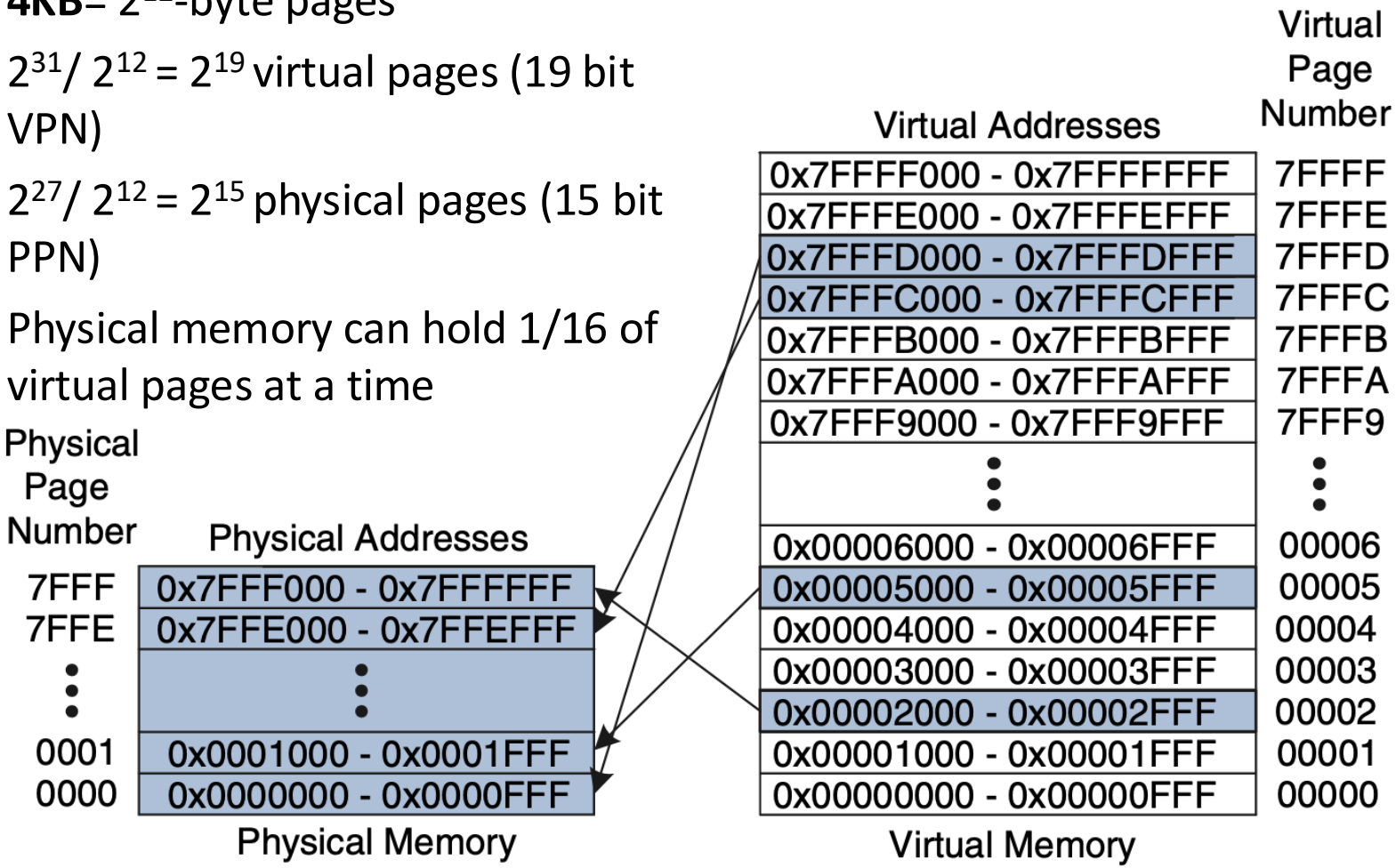
- **Virtual Memory**: divided into **virtual pages** (typically 4KB size)
- **Physical Memory**: divided into **physical pages** (same size)

- **Virtual page** may be located in i) **physical memory** (DRAM) or on **hard drive** (disk)



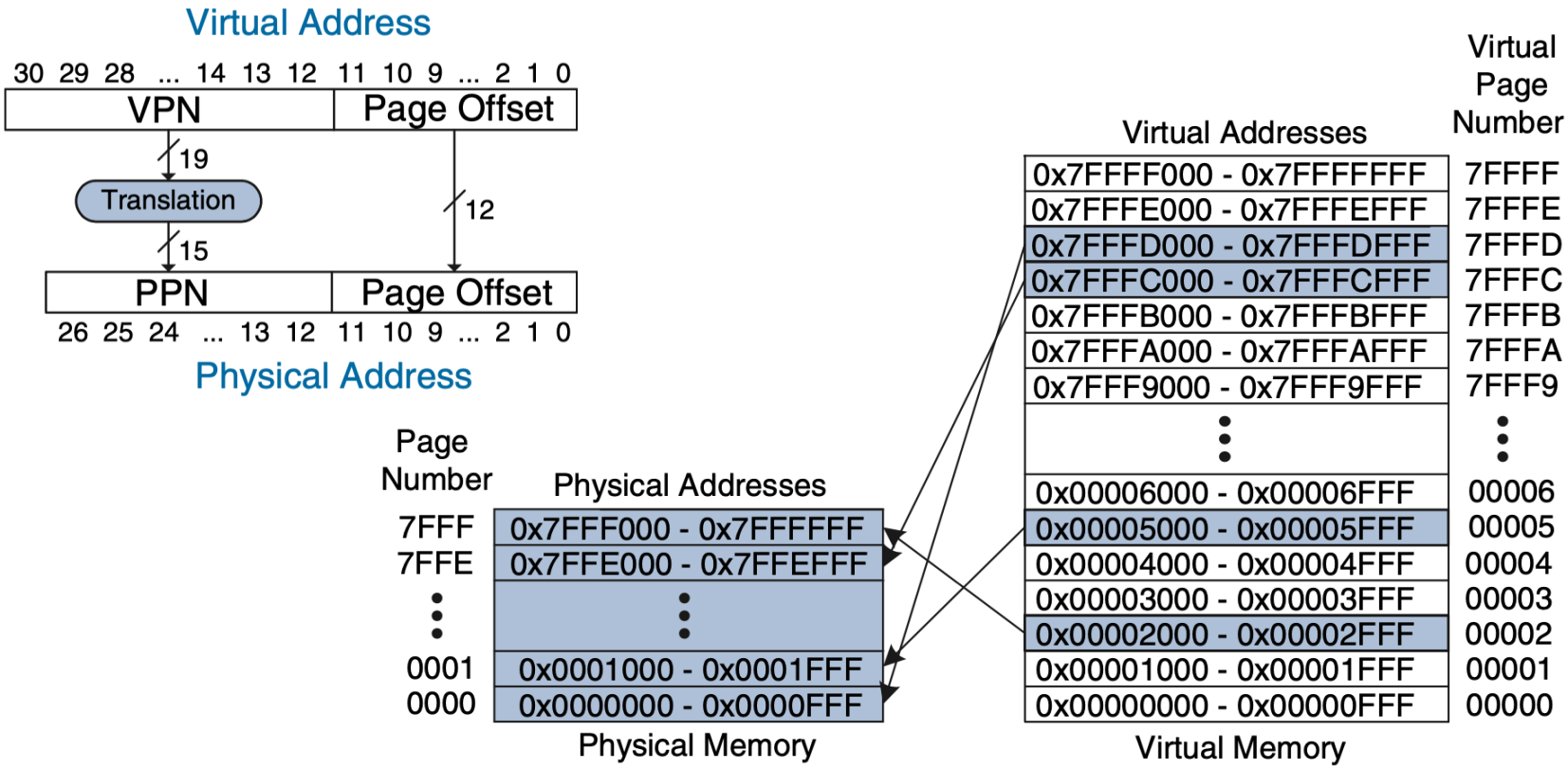
Virtual Memory

- **2GB**=2^{31}-byte virtual memory
- **128MB**=2^{27}-byte physical memory
- **4KB**= 2^{12}-byte pages
- $2^{31}/ 2^{12} = 2^{19}$ virtual pages (19 bit VPN)
- $2^{27}/ 2^{12} = 2^{15}$ physical pages (15 bit PPN)
- Physical memory can hold 1/16 of virtual pages at a time



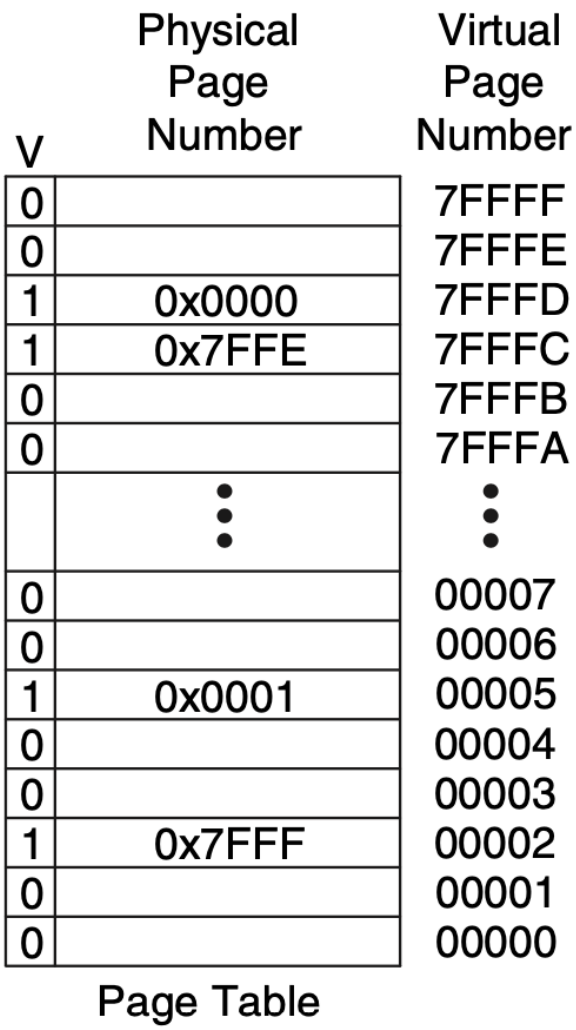
Virtual Memory

- **Address translation**: Process of determining physical address, given a virtual address



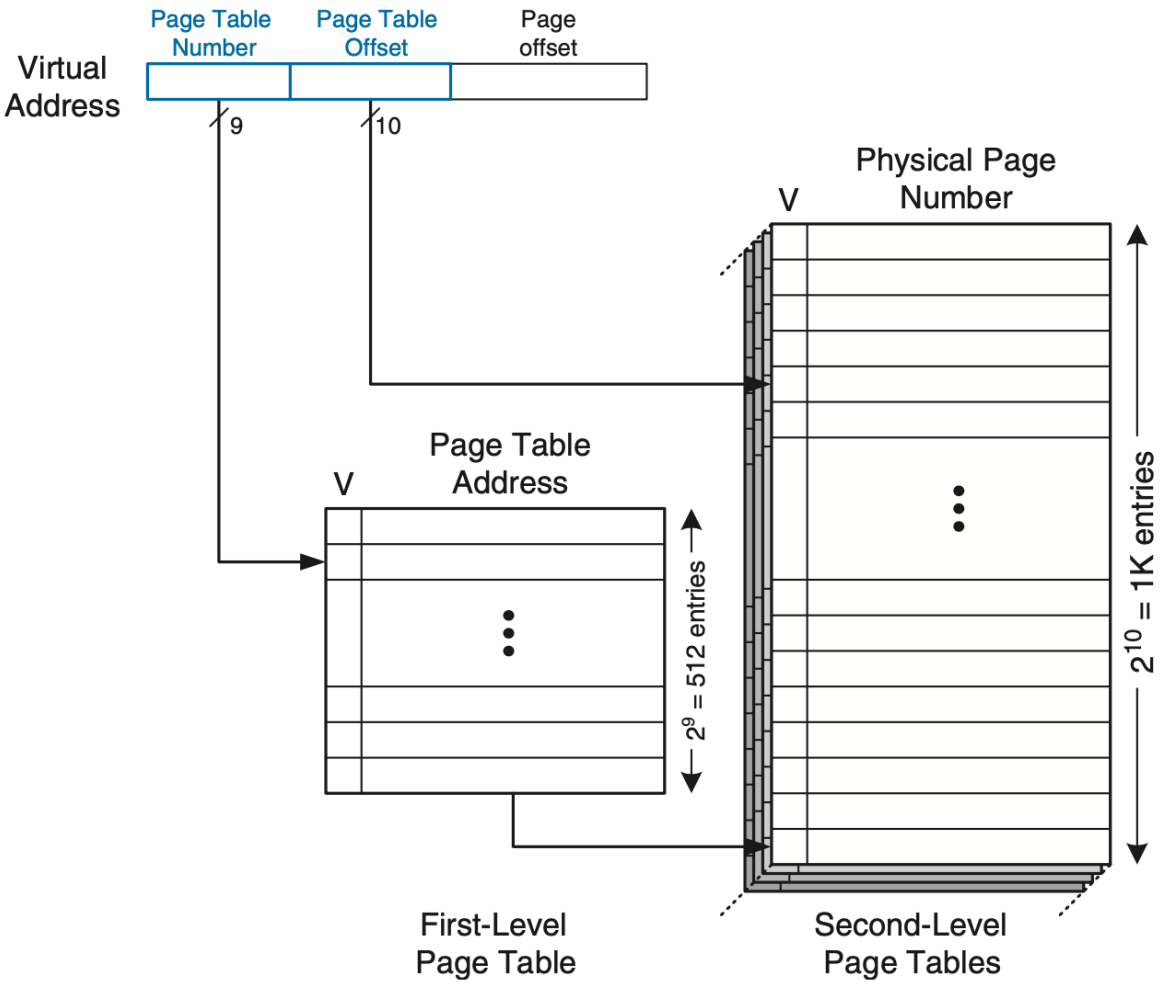
Virtual Memory

- Processor uses **page table** to translate VPN->PPN
- Contains **entry** for each virtual page: Valid bit (if currently in physical memory)
- Indexed with **virtual page number**
- **Entry 5**: specifies virtual page 5 maps to physical page 1
- **Entry 6**: Invalid (V=0) so located on disk



Page Table

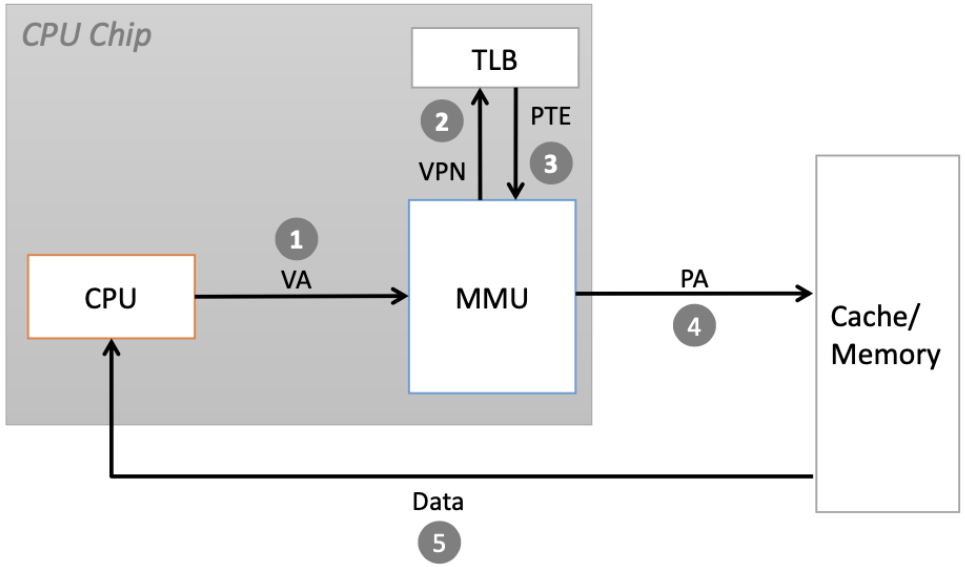
Virtual Memory



- **Page Table Number**: indexes 1st level page table (gives base address of second tbale)
- **Page Table Offset**: indexes 2nd level page table

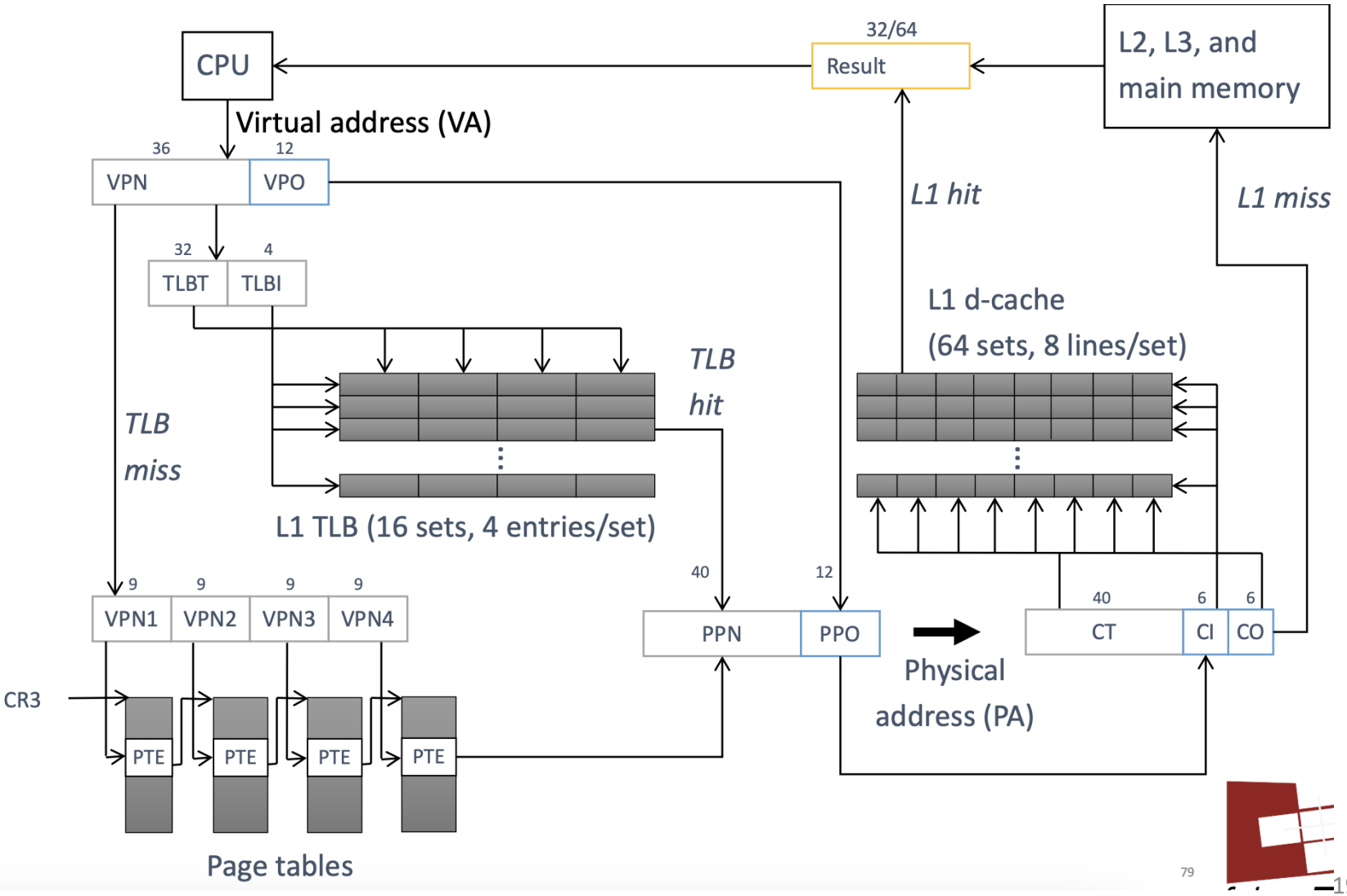
Virtual Memory

TLB hit



A TLB hit eliminates a memory access

Caches and Virtual Memory



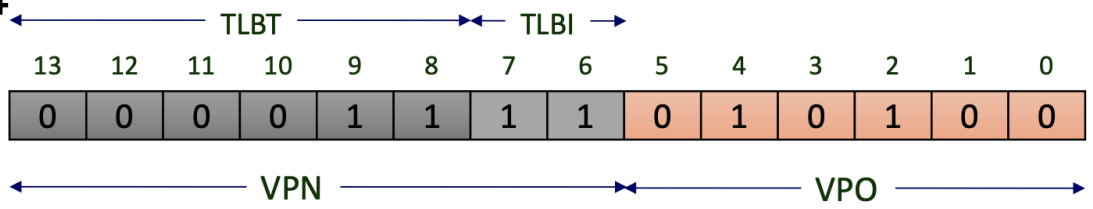
Virtual Memory Recap

Quiz

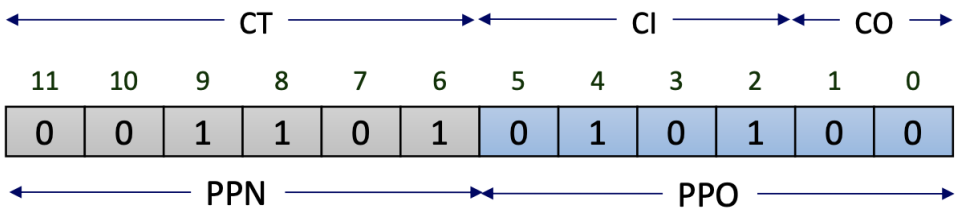
Virtual Memory Quiz

- Recall: TLBI, TLBT, CI, CO, CT positions

Virtual Address: 0x03D4



Physical Address



Virtual Memory Quiz

Systems@ETH zürich

Consider a small memory system with a TLB and a L1 data cache. We make the following assumptions to simplify the question:

- The memory is byte addressable, each access is always to **a single byte**.
- Virtual addresses are 14 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 64 bytes.
- The TLB is two-way set associative with 8 total entries.
- The L1 (data) cache is physically addressed, direct mapped, and has a 4-byte block size; there are 16 sets.

Virtual Memory Quiz

Systems@ETH zürich

TLB: (all values are hexadecimal)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	01	22	1	07		0
3	01	22	1	02		0

Page table: (all values are hexadecimal)

VPN	PPN	Valid
00		
01	02	1
02	03	1
03		
04		
05		
06	22	1
07	22	1

VPN	PPN	Valid
08		
09	20	1
0a		
0b		
0c		
0d	04	1
0e		
0f		

Cache: (all values are hexadecimal)

Index	Tag	Valid	Block[0]	Block[1]	Block[2]	Block[3]
00	00	1	de	ad	fa	ce
01	31	0				
02	24	1	02	13	e1	de
03	22	1	22	23	e2	2e
04	21	0				
05	22	0				
06	18	0				
07	22	1	9a	01	00	de
08	20	0				
09	22	1	83	1a	09	ce
0a	20	1	0d	1f	f1	d0
0b	3a	0				
0c	3f	0				
0d	24	1	be	fb	57	02
0e	23	0				
0f	22	1	cf	7a	9b	a0

Symmetric Multiprocessing (SMP)

SMP

- Computer Architecture: processor design

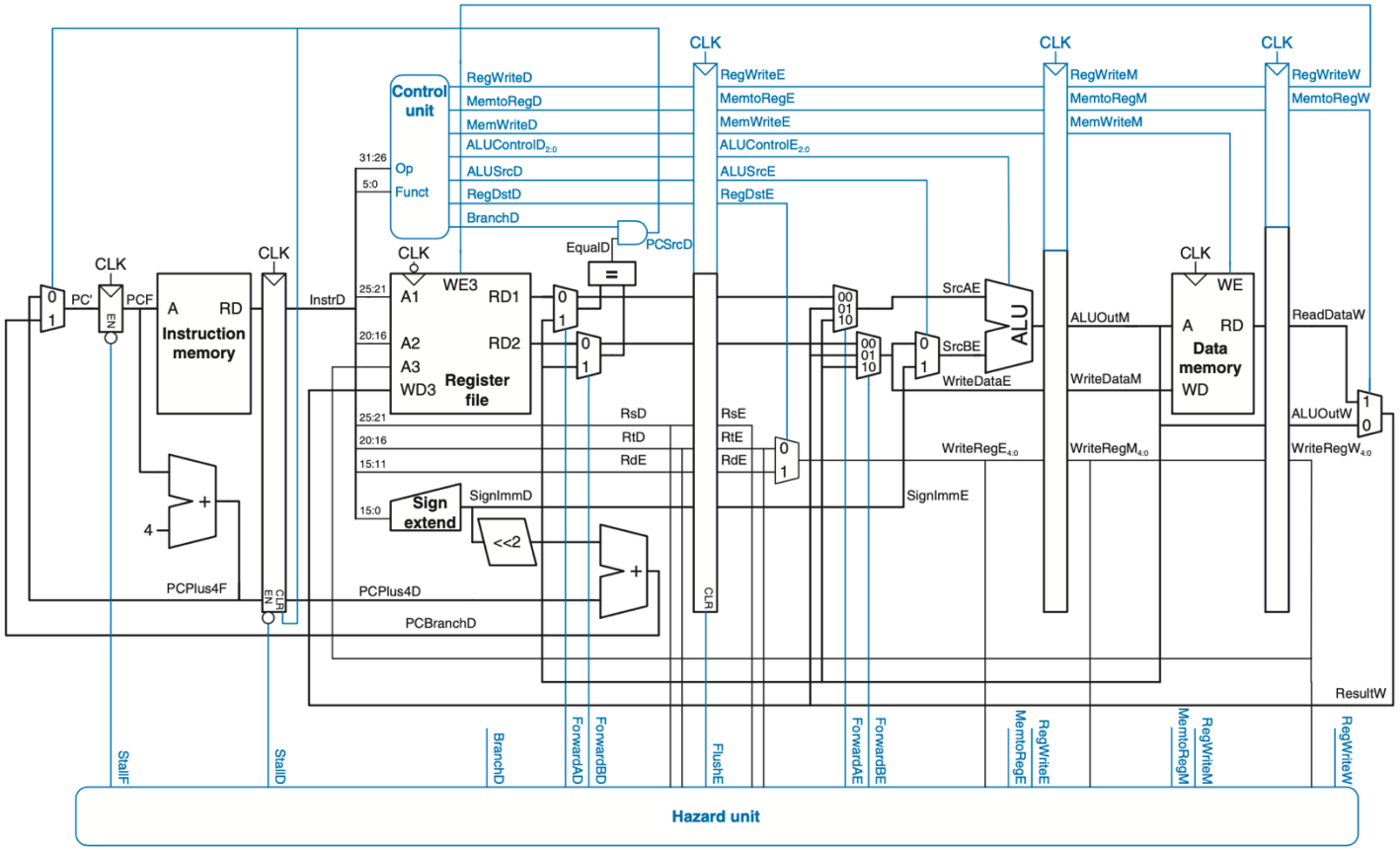
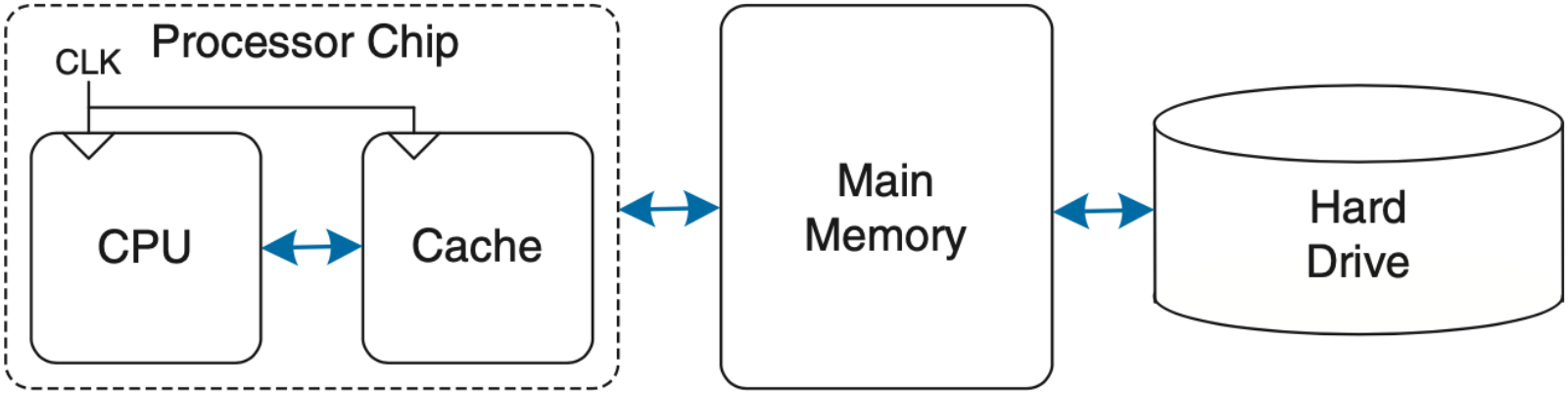


Figure 7.58 Pipelined processor with full hazard handling

SMP

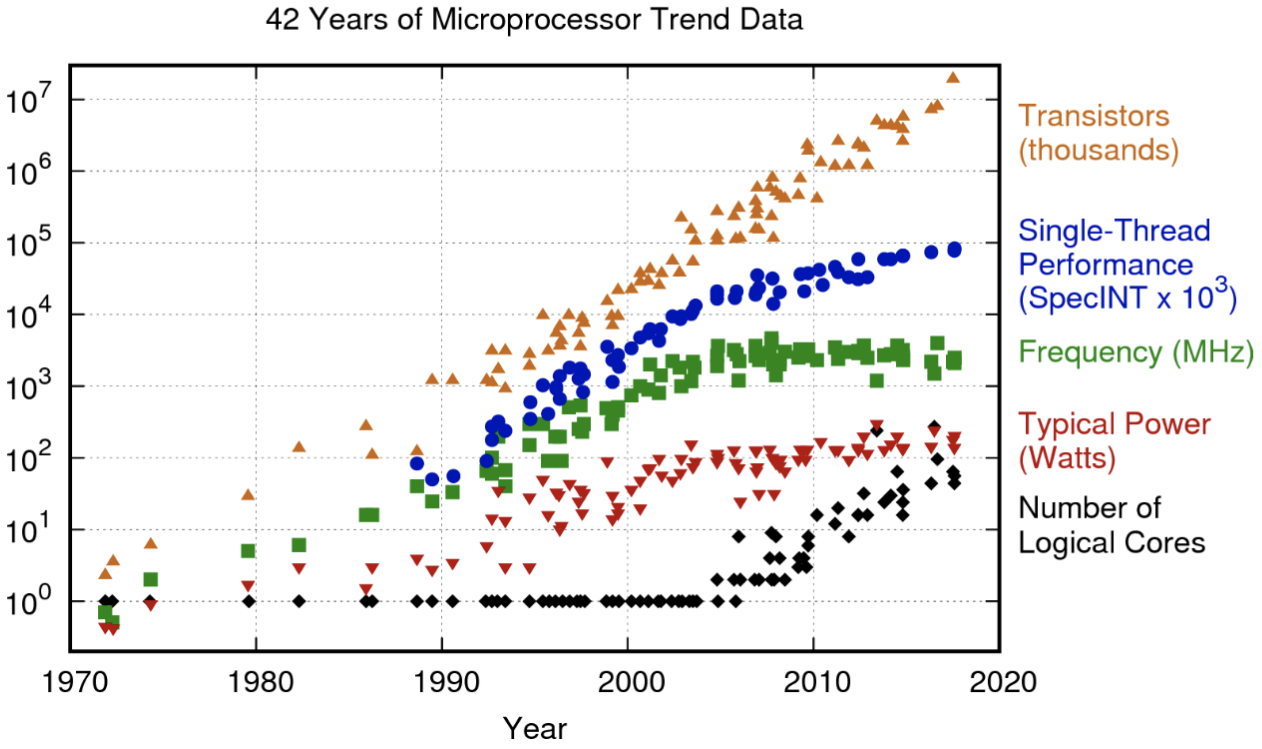
- Last time: **sequential processor design**
- Issue?



SMP

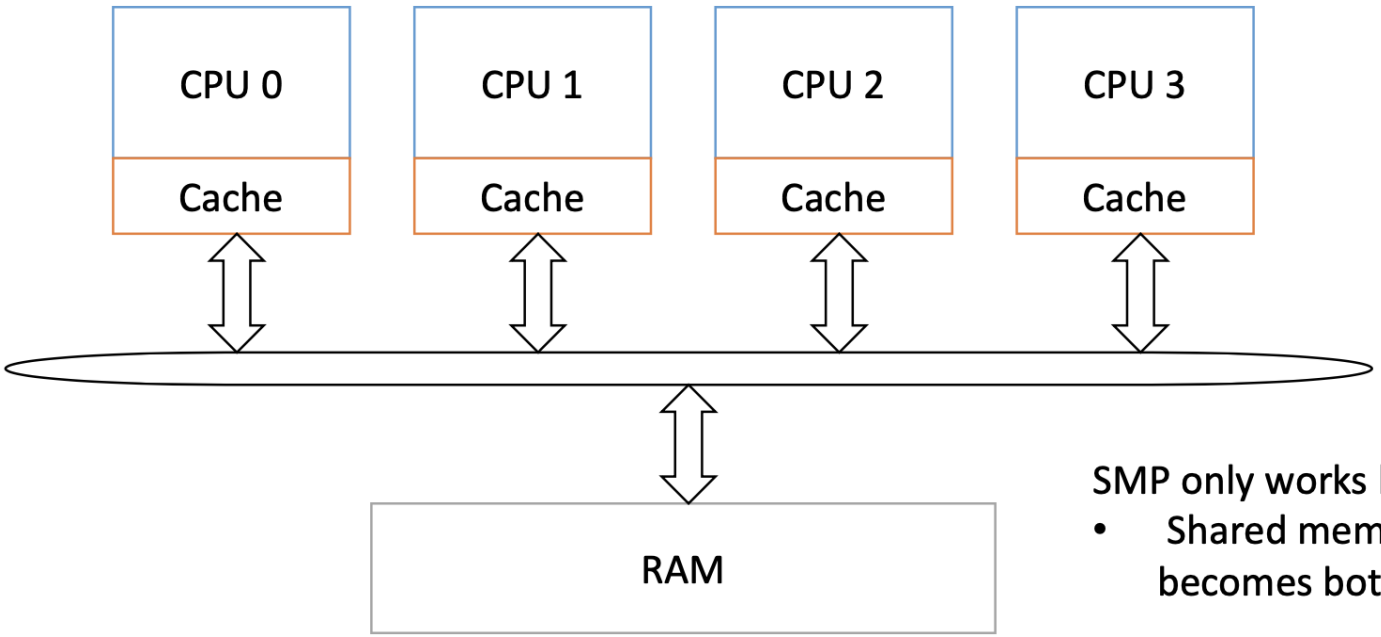
- Power wall + ILP Wall + memory wall => End of serial hardware

Trends



SMP

- This time: **multiple processors per chip**



SMP only works because of caches!

- Shared memory rapidly becomes bottleneck

SMP

Systems@ETH zürich

Multicore processors

- Multiple processor cores per chip
  - This is what computing looks like **today**
- Mostly so far: **shared memory multiprocessors**
  - Single physical address space shared by all processors
  - Communication between processors happens through shared variables in memory
  - Hardware typically provides cache coherence
- This chapter is about this kind of machine.

Cache Coherency

Cache Coherency

Systems@ETH zürich

Coherency and Consistency

- With several processors, memory can change under a cache
  - Leads to 2 important concepts:

1. **Coherency**:
   - Values in caches all match each other
   - Processors all see a coherent view of memory
2. **Consistency**:
   - The order in which changes to memory are seen by different processors

Cache Coherency

Systems@ETH zürich

Cache coherency

- Most CPU cores on a modern machine are **cache coherent**
  - Behave as if all accessing a single memory array
  - We'll see what this *really* means in a moment
- Big advantage: ease of programming
  - Shared-memory programming models work!
    - Pthreads, OpenMP, etc.
- Disadvantages:
  - Complex to implement (lots of transistors, bug-prone)
  - Memory is slower as a result

Cache Coherency

Systems@ETH zürich

- **Cache Coherence**: if one processor updates a value in its cache, other processor see this update when accessing the same value

- **Cache Coherence in Write-through caches**: we "snoop" reads/writes from the bus: if someone writes to a memory location we keep in our cache we invalidate our cache line
- **Cache Coherence in Write-back caches**: **issue**: we don't get to know if someone updates the cache!!

Cache Coherency

Systems@ETH zürich

- **Cache Coherency Protocol**: defines how caches communicate to enforce coherent view of memory

- 1. **MSI**: Basic protocol
- 2. **MESI**: Advanced MSI
- 3. **MOESI**: AMD Advanced MESI
- 3. **MESIF**: Intel Advanced MESI

Cache Coherency

Systems@ETH zürich

- **Cache Coherency Protocol**: defines how caches communicate to enforce coherent view of memory

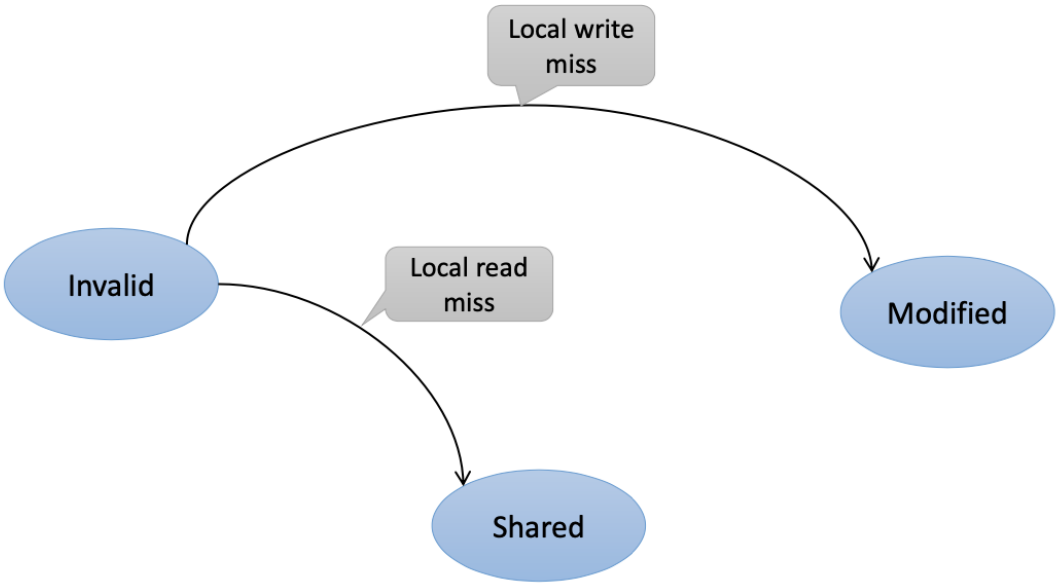
- 1. **MSI**: Basic protocol
- 2. **MESI**: Advanced MSI
- 3. **MOESI**: AMD Advanced MESI
- 3. **MESIF**: Intel Advanced MESI

Cache Coherency

- Simplest protocol: **MSI**
  - Each line has 3 states: Modified, Shared, Invalid
  - Line can only be dirty in one cache
- Cache logic must respond to:
  - Processor reads and writes
  - Remote bus reads and writes
- and must:
  - Change cache line state
  - Write back data (flush) if required

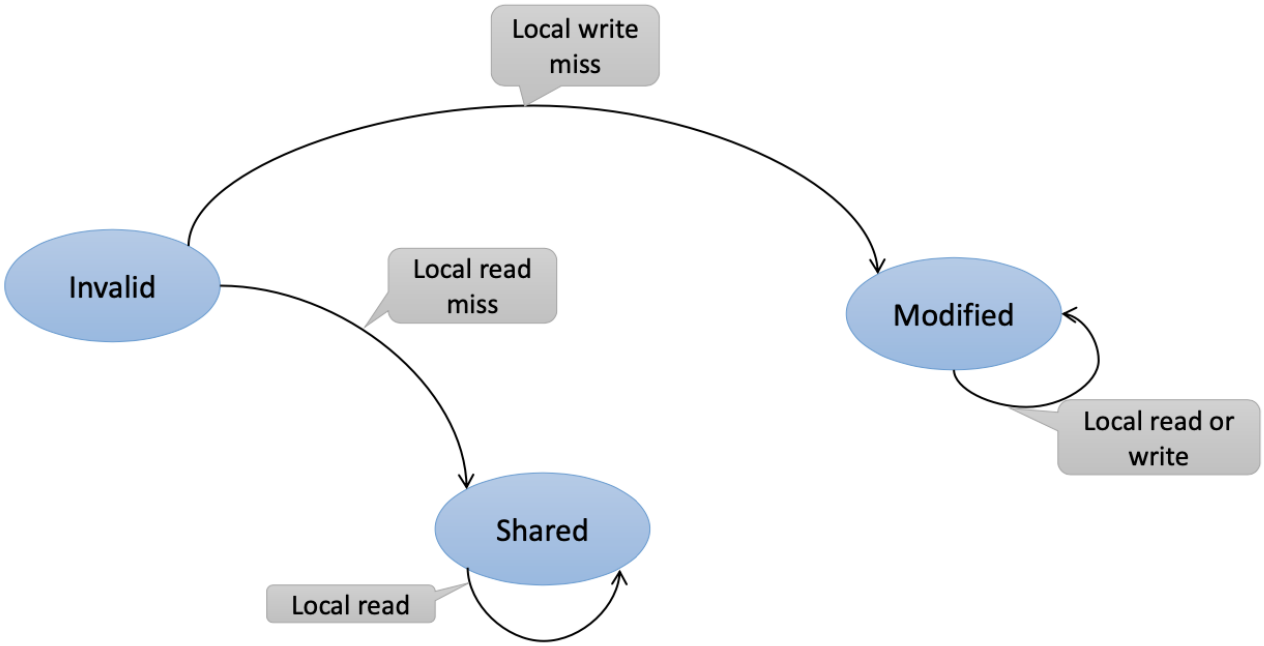
Cache Coherency

MSI state machine: local (processor) transitions



Cache Coherency

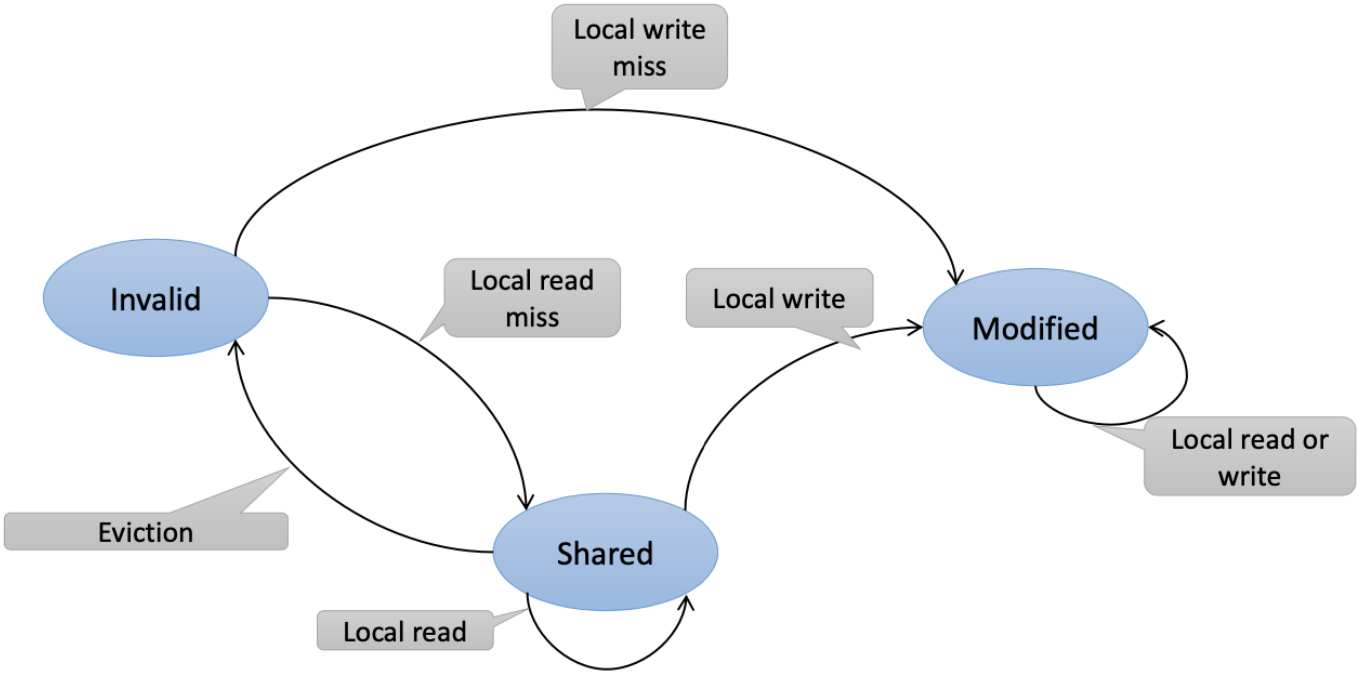
MSI state machine: local (processor) transitions



Cache Coherency

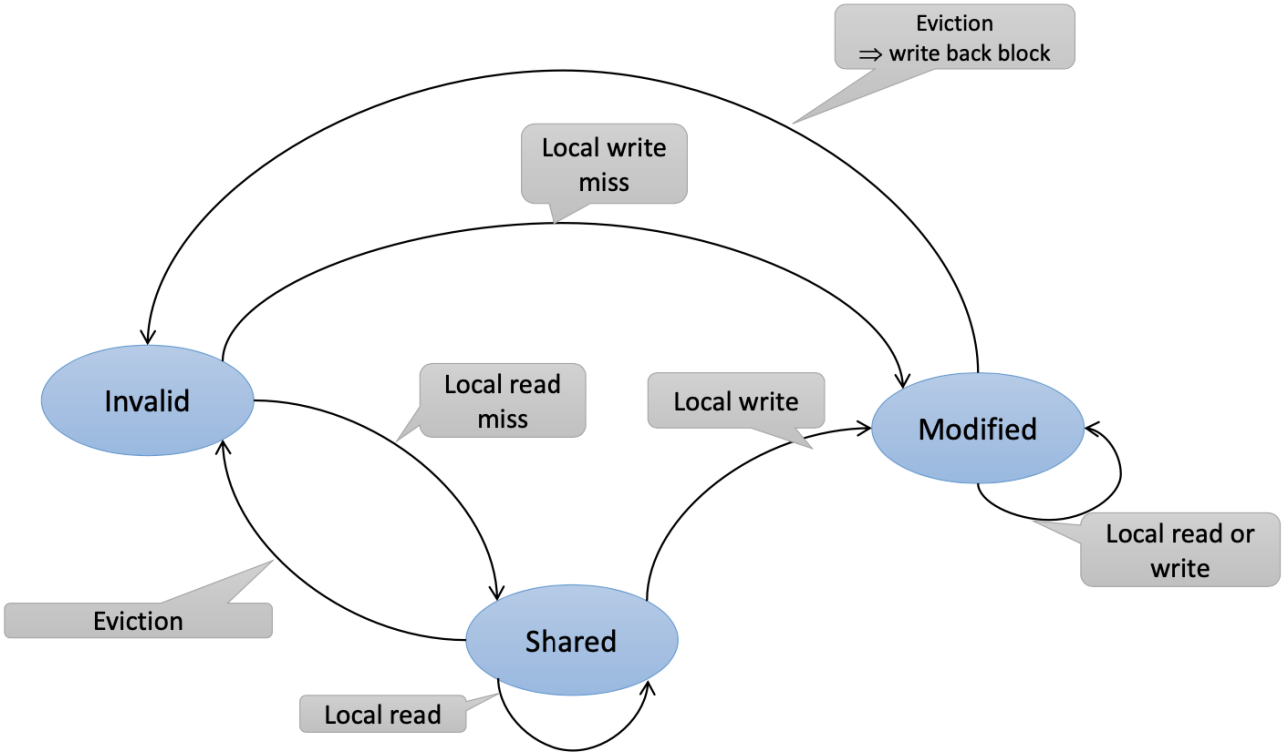
Systems@ETH zürich

MSI state machine: local (processor) transitions



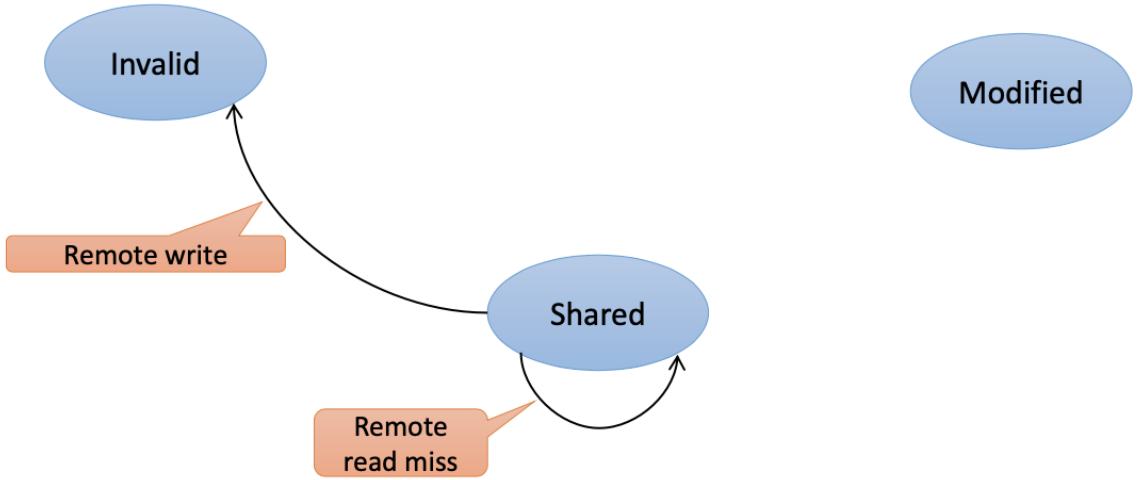
Cache Coherency

MSI state machine: local (processor) transitions



Cache Coherency

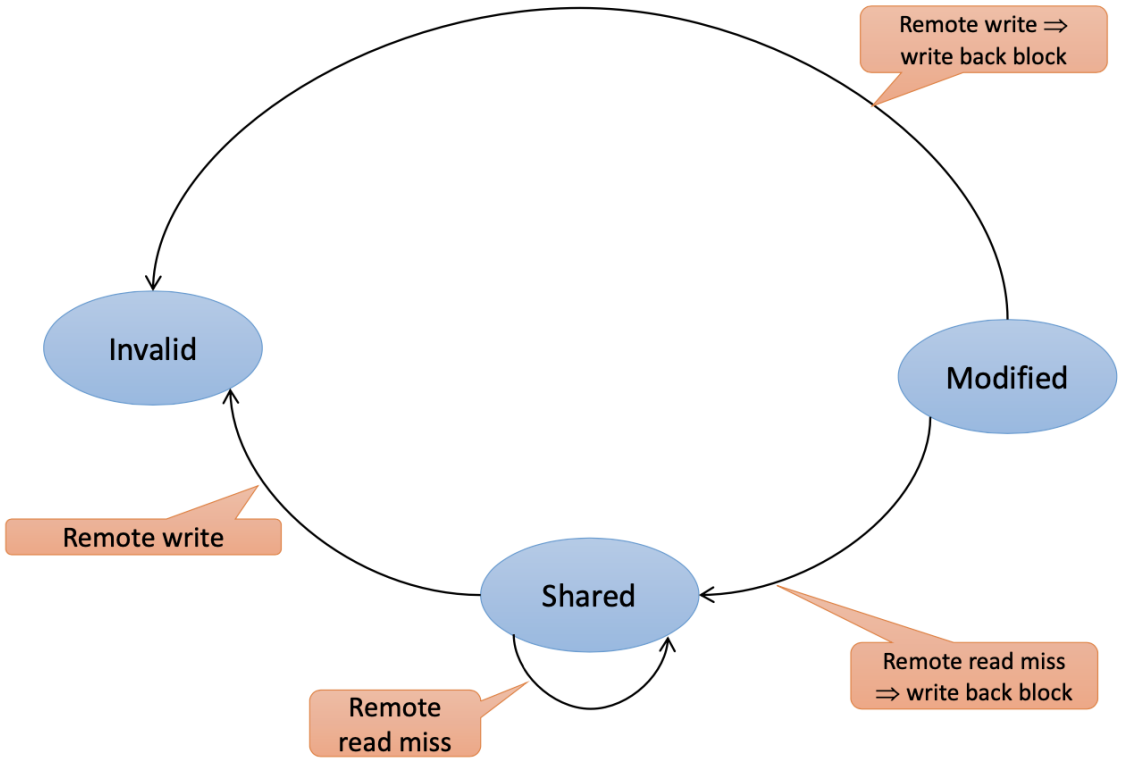
MSI state machine: remote (snooped) transitions



Cache Coherency

Systems@ETH zürich

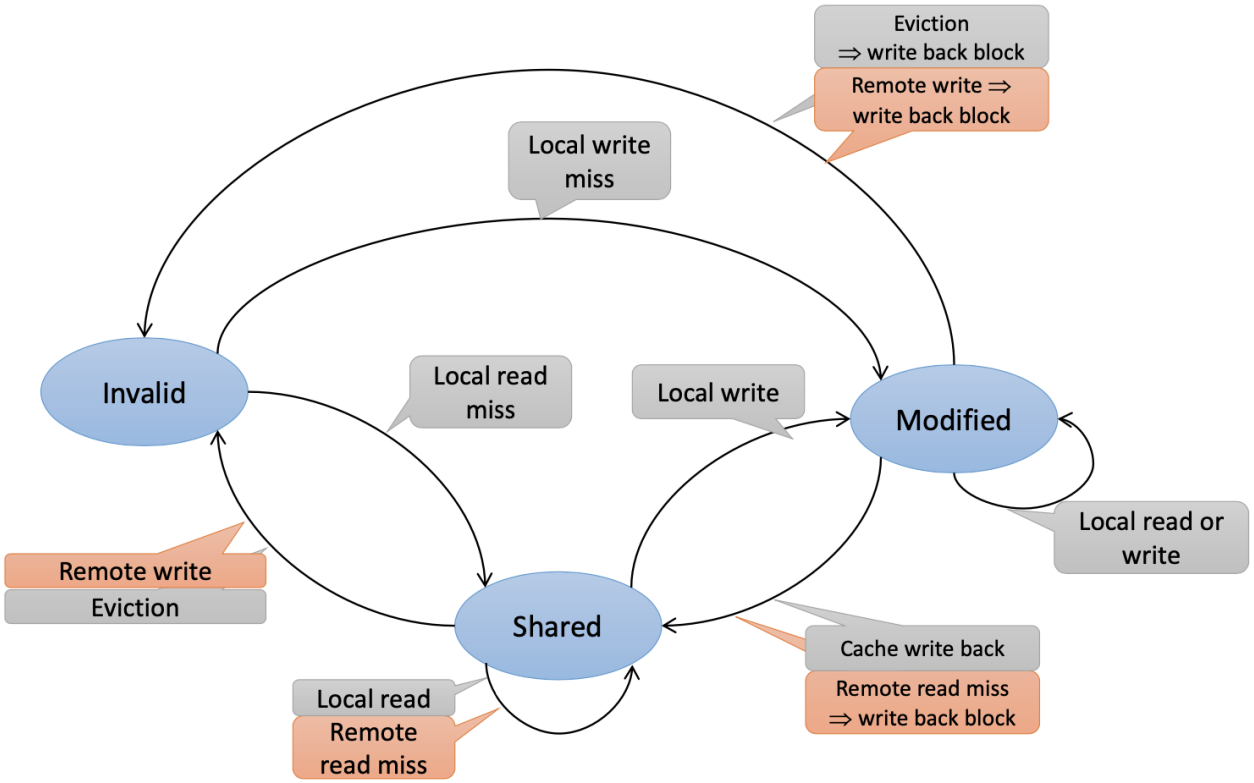
MSI state machine: remote (snooped) transitions



42

Cache Coherency

MSI state machine: all transitions



43

Cache Coherency

MSI invariants

- A block can only be in *Modified* state in up to one cache
- Multiple blocks can be in *Shared* state (if no cache has the block in *Modified*)

	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

Cache Coherency

Systems@ETH zürich

- **Cache Coherency Protocol**: defines how caches communicate to enforce coherent view of memory

- 1. **MSI**: Basic protocol
- 2. **MESI**: Advanced MSI
- 3. **MOESI**: AMD Advanced MESI
- 3. **MESIF**: Intel Advanced MESI

Cache Coherency

Systems@ETH zürich

MESI protocol

- Add a new line state: “exclusive”
  - **Modified**: This is the only copy, it’s dirty
  - **Exclusive**: This is the only copy, it’s clean
  - **Shared**: This might be one of several copies, all clean
  - **Invalid**
- **HIT** signal
  - Signals to a remote processor that its read hit in local cache.
  - Cache can load a block into either “shared” or “exclusive” states based on whether the block is a HIT in remote processor caches
- First x86 appearance in the Pentium

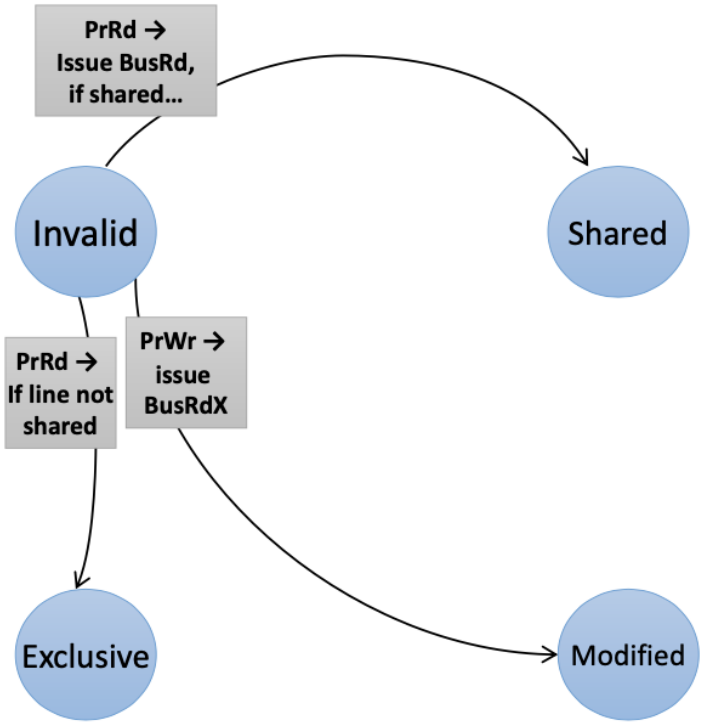
Cache Coherency

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

Processor-initiated



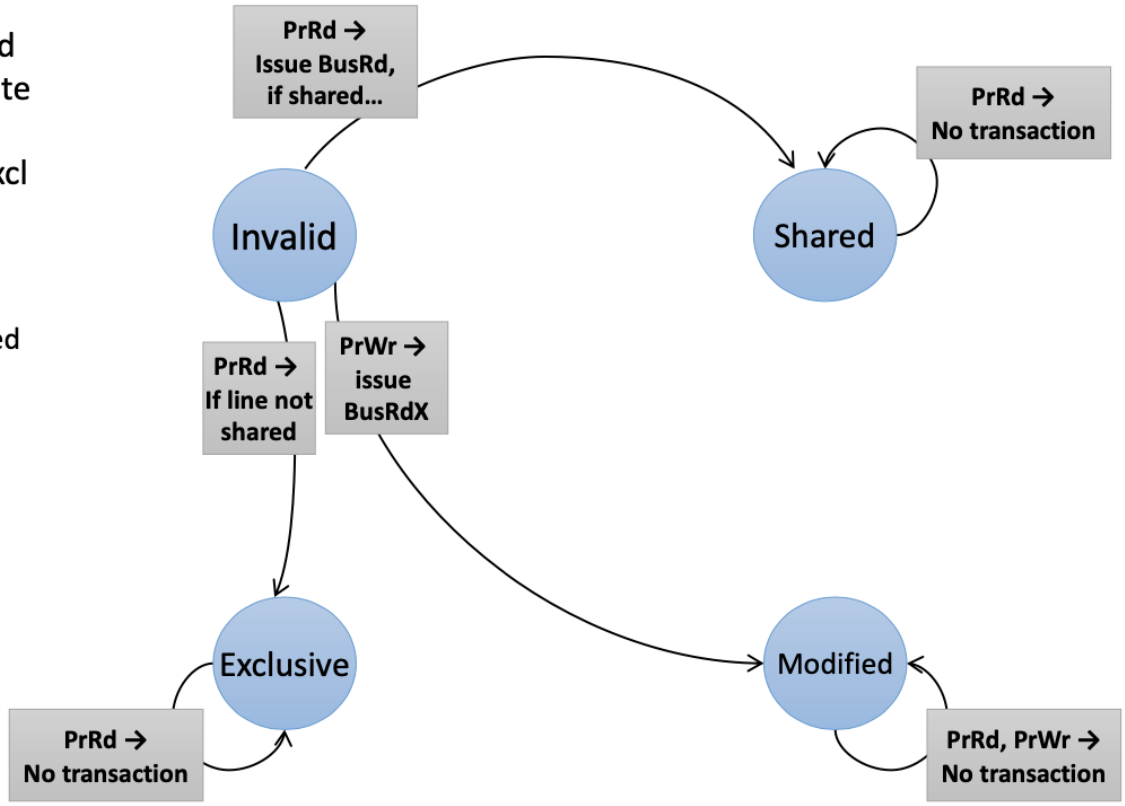
Cache Coherency

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

Processor-initiated



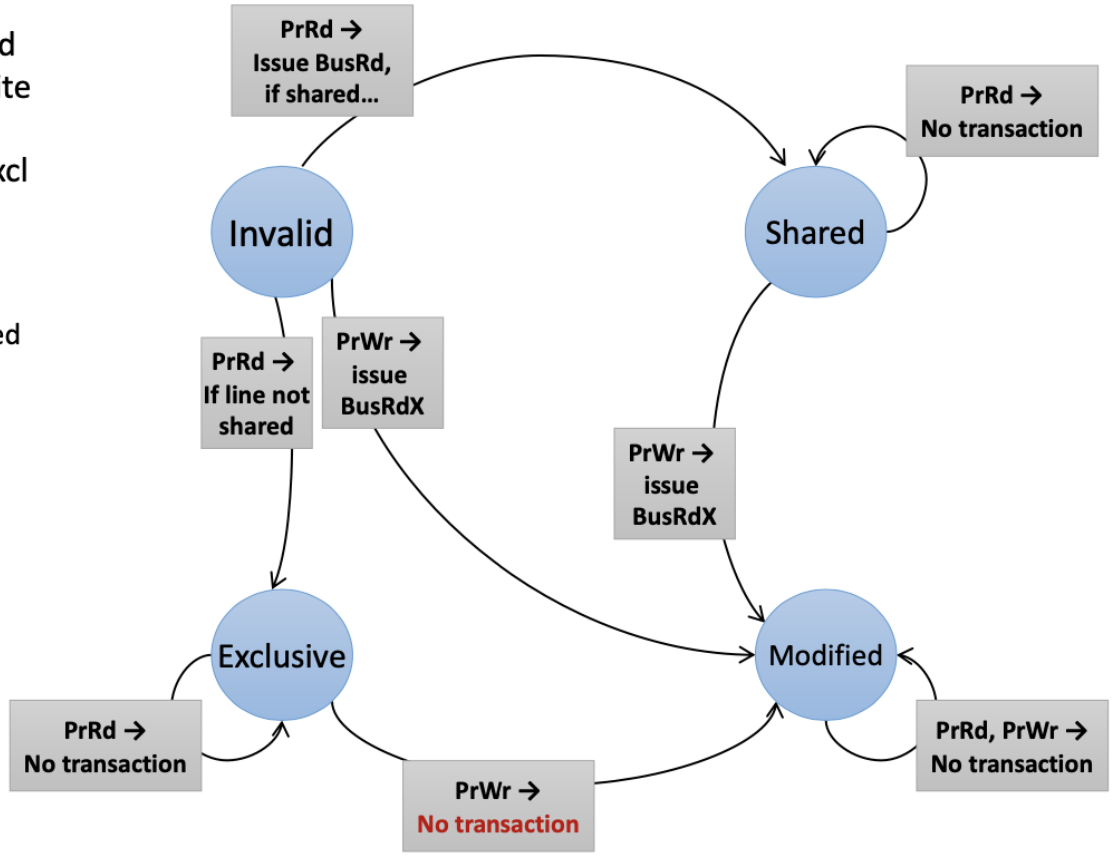
Cache Coherency

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

Processor-initiated



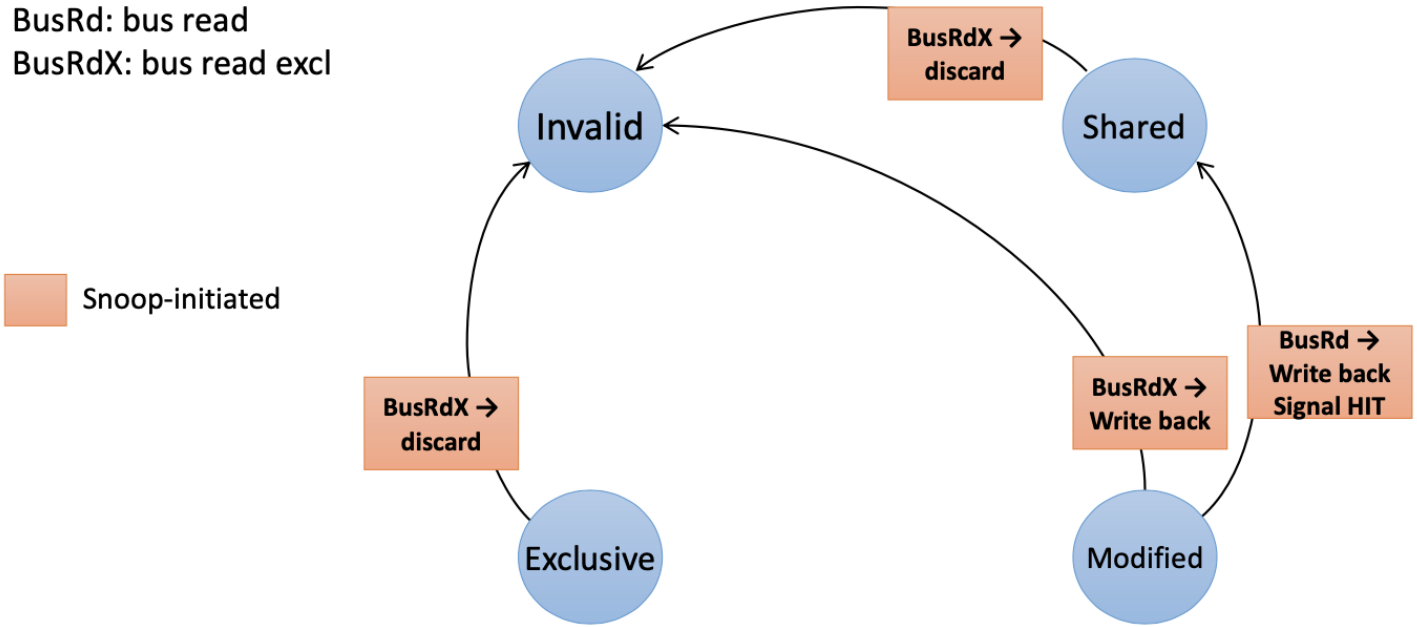
Cache Coherency

Systems@ETH zürich

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

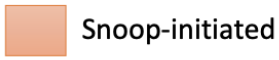


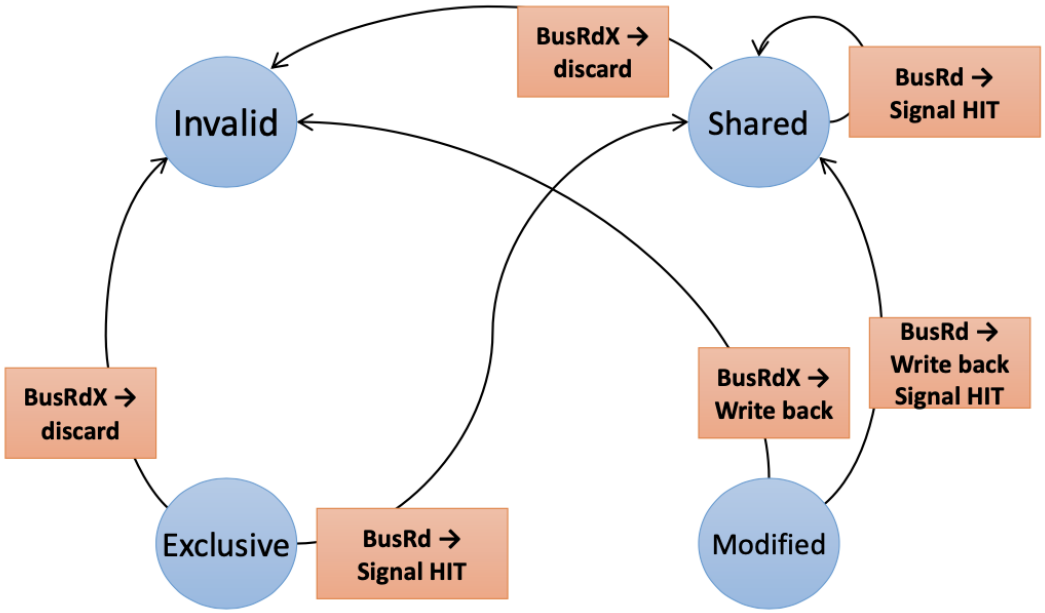
Cache Coherency

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



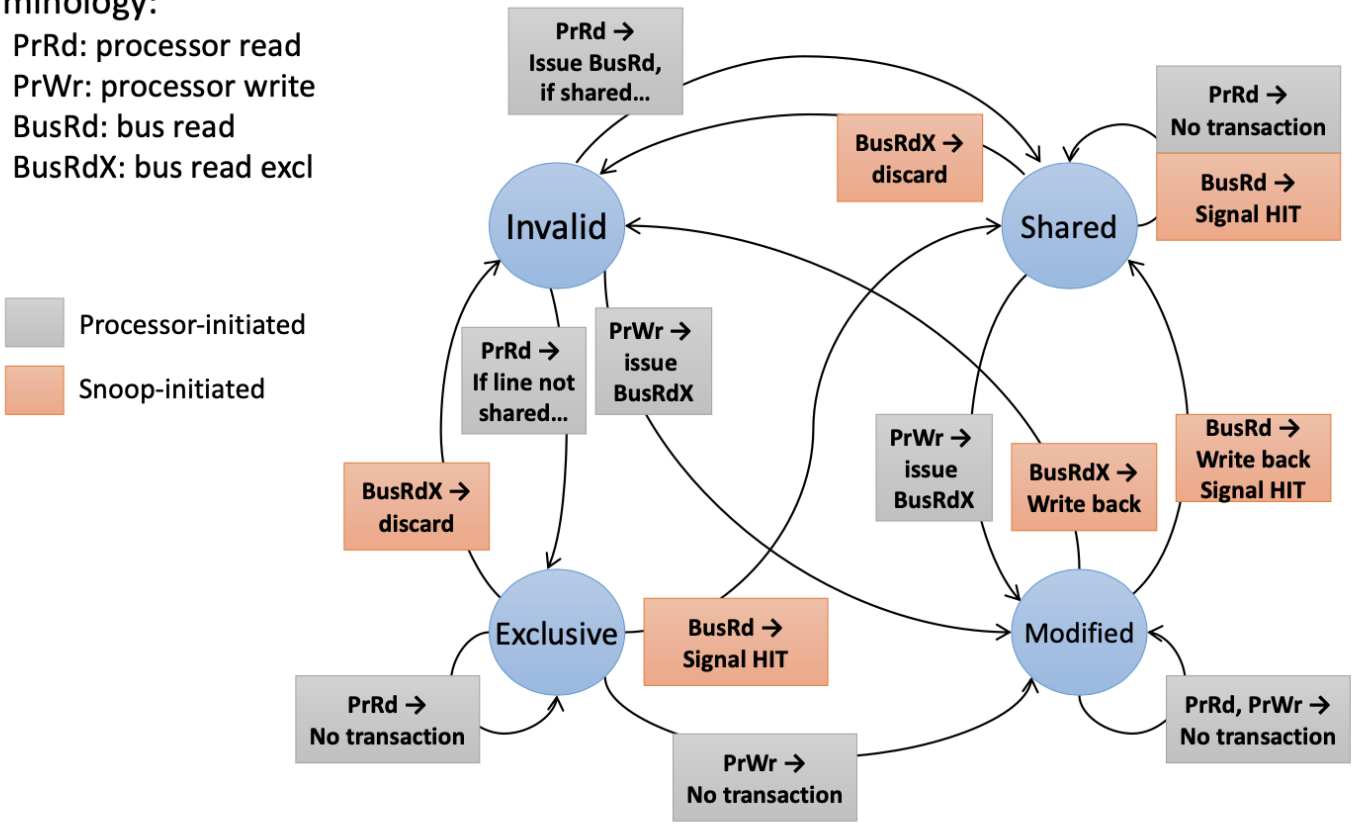


Cache Coherency

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



Cache Coherency

Systems@ETH zürich

MESI invariants

- Allowed combination of states for a line between any pair of caches:

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

- Protocol must preserve these invariants

MSI invariants:

	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

Cache Coherency

Systems@ETH zürich

- **Cache Coherency Protocol**: defines how caches communicate to enforce coherent view of memory

- 1. **MSI**: Basic protocol
- 2. **MESI**: Advanced MSI
- 3. **MOESI**: AMD Advanced MESI
- 3. **MESIF**: Intel Advanced MESI

Cache Coherency

Systems@ETH zürich

MOESI protocol (AMD)

Add new "Owner" state: allow line to be modified, but other dirty copies to exist in other caches.

Modified:
No other cached copies exist, local copy dirty

Owner:
Multiple dirty copies exist (all consistent).
This copy has sole right to modify line.

Exclusive:
No other cached copies exist, local copy clean

Shared:
Other cached copies exist (all consistent, but might be dirty or clean).
One other copy might be able to write (state Owner)

Invalid:
Not in cache.

Cache Coherency

MOESI invariants

- Can quickly satisfy read request for dirty cache line without writeback to memory
  - Owner cache must respond with line.
- Read requests for clean, shared line must be served by memory
- Good if latency of remote cache < latency of main memory

	M	O	E	S	I
M					✓
O				✓	✓
E					✓
S		✓		✓	✓
I	✓	✓	✓	✓	✓

Cache Coherency

Systems@ETH zürich

- **Cache Coherency Protocol**: defines how caches communicate to enforce coherent view of memory

- 1. **MSI**: Basic protocol
- 2. **MESI**: Advanced MSI
- 3. **MOESI**: AMD Advanced MESI
- 3. **MESIF**: Intel Advanced MESI

Cache Coherency

Systems@ETH zürich

MESIF protocol (Intel)

Add new "Forward" state:
designates at-most-one Shared line to serve remote requests.

Modified:
No other cached copies exist, local copy dirty

Exclusive:
No other cached copies exist, local copy clean

Shared:
Other cached copies exist, all copies are clean
At most one other (clean) copy might be in Forward state.

Invalid:
Not in cache.

Forward:
As Shared, but this is the *designated responder* for requests
Always the most recent cache to request line

Cache Coherency

MESIF invariants

- At most one copy is in Forward state.
  - Most recent cache to request line
  - Avoids incast storm if line is widely shared
- If none, request served by main memory
  - Even if shared copies exist

	M	E	S	I	F
M				✓	
E				✓	
S			✓	✓	✓
I	✓	✓	✓	✓	✓
F			✓	✓	

Memory Consistency Models

Consistency Models

Systems@ETH zürich

Coherency and Consistency

- With several processors, memory can change under a cache
  - Leads to 2 important concepts:

1. **Coherency**:
   - Values in caches all match each other
   - Processors all see a coherent view of memory
2. **Consistency**:
   - The order in which changes to memory are seen by different processors

Consistency Models

Systems@ETH zürich

- **Cache coherence** give guarantees to **individual memory locations**
- **Memory consistency models** make guarantees on ordering of operations across multiple memory locations

- => Cache coherence is lower level mechanism helping maintain a consistent view of memory on hardware level
- => MCMs provide higher level guarantees on order of operations: effective cache coherence models are essential to implement MCM

Consistency Models

Systems@ETH zürich

Memory consistency

- When several processors are reading and writing memory, what value is read by each processor?
  - Not an easy question to answer
  - "Last value written":
    - By which processor?
    - What do we mean by "last"?
- Important to have an answer!
  - Defines semantics of order-dependent operations
    - E.g. does Dekker's algorithm work?
    - How to ensure that it does work?
- There are many memory consistency **models**

Consistency Models

Systems@ETH zürich

Consistency models: terminology

- **Program order**: order in which a program on a processor appears to issue reads and writes
  - Refers only to local reads/writes
  - Even on a uniprocessor ≠ order the CPU issues them!
  - Write-back caches, write buffers, out-of-order execution, etc.
- **Visibility order**: order which all reads and writes are seen by one or more processors
  - Refers to all operations in the machine
  - Might not be the same for all processors
  - Each processor reads the value written by the last write in visibility order

Consistency Models

Systems@ETH zürich

- There are **two consistency models** we looked at in this course

- 1. **Sequential Consistency**
- 2. **Processor Consistency**

Consistency Models

Systems@ETH zürich

- There are **two consistency models** we looked at in this course

- 1. **Sequential Consistency**
- 2. **Processor Consistency**

Consistency Models

Systems@ETH zürich

Sequential consistency

1. Operations from a processor appear (to all others) in program order
2. Every processor's visibility order is the same *interleaving* of all the program orders.

Requirements:

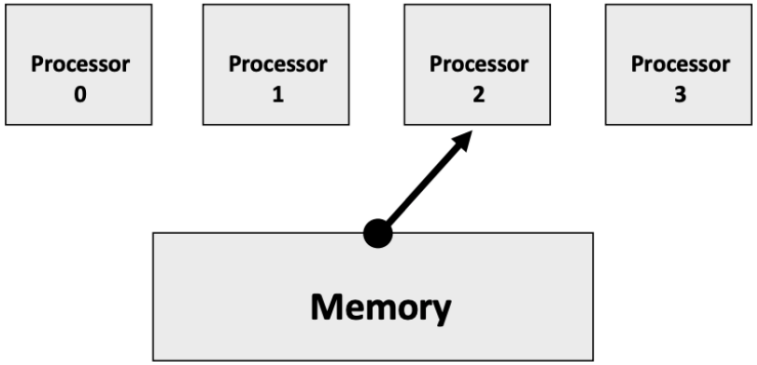
- Each processor issues memory ops in program order
- RAM totally orders all operations
- Memory operations are (globally) atomic

Consistency Models

Systems@ETH zürich

Sequential consistency

- Switch metaphor:
  - All processors issue loads and stores in program order
  - Memory chooses a processor, performs a memory operation to completion, then chooses another processor, …



Consistency Models

Systems@ETH zürich

Sequential consistency example

Assume *p = 0, *q = 0 to begin with

CPU A		CPU B	
a_1:	*p = 1;	b_1:	u = *q;
a_2:	*q = 1;	b_2:	v = *p;

Results:

- (u=1, v=1):
  - Possible under SC: (a_1, a_2, b_1, b_2)
  - (a_1, a_2) and (b_1, b_2) are both program orders
- (u=1, v=0):
  - Impossible under SC:
  - No interleaving of program orders that generates this result
  - Would require: $a_2 > b_1 > b_2 > a_1$

Consistency Models

Systems@ETH zürich

- **Advantage**
  - Easy to understand for programmer (analyze, write code)
- **Disadvantage**
  - Too slow to be practical: cannot reorder reads/writes (not in the compiler; not even in one single processor)

Consistency Models

Systems@ETH zürich

- There are **two consistency models** we looked at in this course

- 1. **Sequential Consistency**
- 2. **Processor Consistency**

Consistency Models

Systems@ETH zürich

Processor Consistency

- Standard for 64-bit x86 processors
  - Sometimes called Total Store Ordering (TSO)
  - Earlier 32-bit x86 implemented PRAM – weaker!
- Write-to-read relaxation:
  later reads can bypass earlier writes
  - All processors see **writes** from one processor in the order they were issued.
  - Processors can see **different interleavings of writes** from different processors.

Consistency Models

Systems@ETH zürich

Processor Consistency (PC)

- (u,v) = (0,0) is possible in PC
  - a2 read bypasses a1 write
  - b2 read bypasses b1 write

Assume *p = 0, *q = 0 to begin with

CPU A		CPU B	
a_1:	*p = 1;	b_1:	*q = 1;
a_2:	u = *q	b_2:	v = *p;

Consistency Models

Systems@ETH zürich

Other consistency models

	Alpha	PA-RISC	ARM	Power	X86_32	X86_64	ia64	zSeries
Reads after reads	✓	✓	✓	✓			✓	
Reads after writes	✓	✓	✓	✓			✓	
Writes after reads	✓	✓	✓	✓	✓	✓	✓	✓
Writes after writes	✓	✓	✓	✓			✓	
Dependent reads	✓							
Ifetch after write	✓		✓	✓	✓		✓	✓

Consistency Models

Systems@ETH zürich

- With a **weak consistency models**, we have **fast execution** but **low guarantees**
- What if in certain cases we **really want guarantees** (to argue for correctness of algorithms etc.)?
- **Solution**: use **barriers** (aka **fences**)

- 1. **Compiler barriers**: prevents compiler from reordering
- 2. **Memory barriers**: prevent CPU from reordering

Consistency Models

Systems@ETH zürich

- With a **weak consistency models**, we have **fast execution** but **low guarantees**
- What if in certain cases we **really want guarantees** (to argue for correctness of algorithms etc.)?
- **Solution**: use **barriers** (aka **fences**)

- 1. **Compiler barriers**: prevents compiler from reordering
- 2. **Memory barriers**: prevent CPU from reordering

Consistency Models

Systems@ETH zürich

Compiler barriers

- Prevents the ***compiler*** from reordering visible loads and stores
  - May still reorder register access (private)
- Typically part of *compiler intrinsics*
- *GCC:*

```
__asm__ __volatile__ ("" ::: "memory");
```

- Intel ECC:

```
__memory_barrier()
```

- Microsoft Visual C & C++:

```
__ReadWriteBarrier()
```

Consistency Models

Systems@ETH zürich

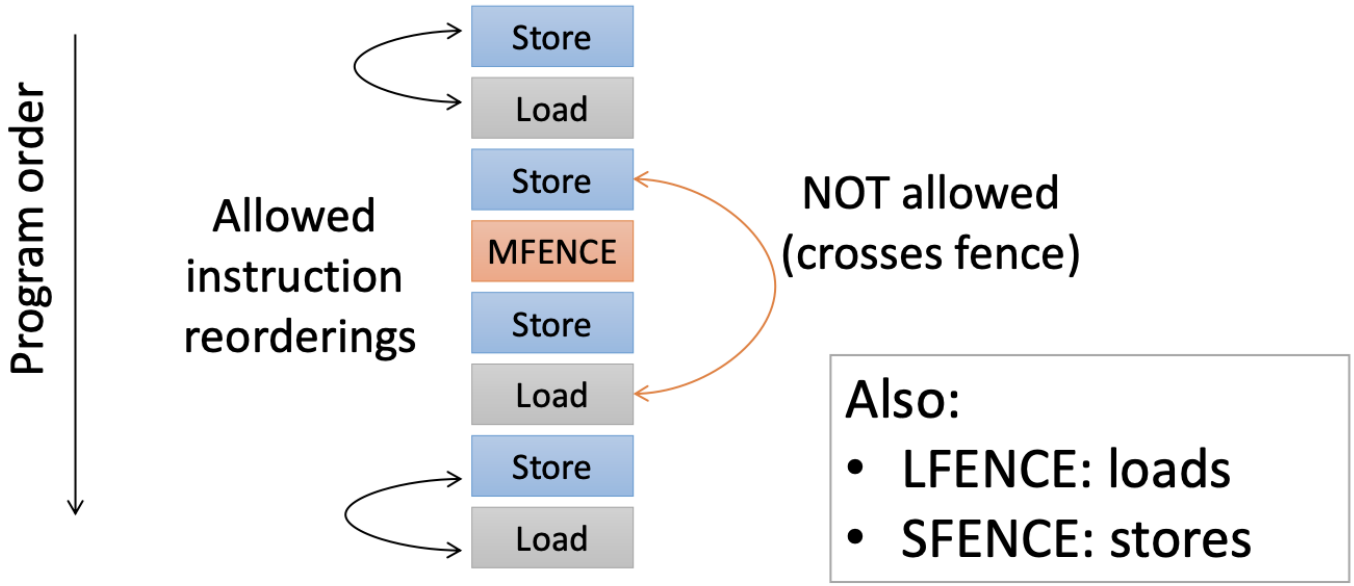
- With a **weak consistency models**, we have **fast execution** but **low guarantees**
- What if in certain cases we **really want guarantees** (to argue for correctness of algorithms etc.)?
- **Solution**: use **barriers** (aka **fences**)

- 1. **Compiler barriers**: prevents compiler from reordering
- 2. **Memory barriers**: prevent CPU from reordering

Consistency Models

Memory barriers on x86

- MFENCE instruction
  - Prevents the ***CPU*** reordering any loads or stores past it



Synchronisation methods

Synchronisation methods

Systems@ETH zürich

- **Fences** assure that the compiler/cpu doesn't reorder instructions
- But they **do not prevent race conditions**: what can we do if multiple processor **access the same memory location**?

- 1. **TAS** (TTAS Lock)
- 2. **CAS**

Synchronisation methods

Systems@ETH zürich

- **Fences** assure that the compiler/cpu doesn't reorder instructions
- But they **do not prevent race conditions**: what can we do if multiple processor **access the same memory location**?

- 1. **TAS** (TTAS Lock)
- 2. **CAS**

Synchronisation methods

Systems@ETH zürich

- **TAS** (Test-and-Set)
  - 1. Reads current value of a memory location
  - 2. Set memory location to a 1 (to indicate ”locked”)
  - 3. Returns original value

- **Memory bus must be locked** during the execution of the instruction (**need hardware support for TAS**)

Synchronisation methods

Systems@ETH zürich

Using Test-And-Set

- Acquire a mutex with TAS:

```
void acquire( int *lock) {
    while ( TAS(lock) == 1)
      ;
}
```

- This is a **spinlock**: keep trying in a tight loop
  - Often fastest if lock is not held for long
- Release is simple:

```
void release( int *lock) {
    *lock = 0;
}
```

Synchronisation methods

Systems@ETH zürich

Test And Test-And-Set

- Replace most of RMW cycles with simple reads:

```
void acquire( int *lock) {
    do {
      while (*lock == 1);
    } while ( TAS(lock) == 1);
}
```

- Think about cache traffic:
  - Reads hit in the spinner's cache
  - Write due to release invalidates cache line
    ⇒ load from main memory, returns 0
    ⇒ triggers further RMW cycle from spinner
  - Highly likely to succeed (unless contention)

Synchronisation methods

Systems@ETH zürich

- **Fences** assure that the compiler/cpu doesn't reorder instructions
- But they **do not prevent race conditions**: what can we do if multiple processor **access the same memory location**?

- 1. **TAS** (TTAS Lock)
- 2. **CAS**

Synchronisation methods

Systems@ETH zürich

Compare and Swap

```
CAS( location, old, new) atomically
{
    1. Load location into value
    2. If value == "old" then store
       "new" to location
    3. Return value
}
```

Interesting features:

- Theoretically more powerful than TAS, FAA, etc.
- Can implement almost all wait-free data structures
- Requires bus locking, or similar, in the memory system

Synchronisation methods

Systems@ETH zürich

The ABA problem

- CAS has a problem:
  - Reports when a single location is different
  - Does not report when it is written (with the same value)
- Leads to the "ABA" problem:
  1. CPU A reads value as x
  2. CPU B writes y to value
  3. CPU B writes x to value
  4. CPU A reads value as x $\Rightarrow$ concludes nothing has changed
- Many problems:
  - E.g., what if the value is a software stack pointer?

Synchronisation methods

Systems@ETH zürich

Solving the ABA problem

- Basic problem:
  - Value used for CAS comparison has not changed
  - But the data has
  - CAS doesn't say whether a write has occurred, only if a value has changed.
- Solution:
  - Ensure the value always changes!
  - Split value into:
    - Original value
    - Monotonically increasing counter
  - CAS both halves in a single instruction

NUMA

NUMA

Systems@ETH zürich

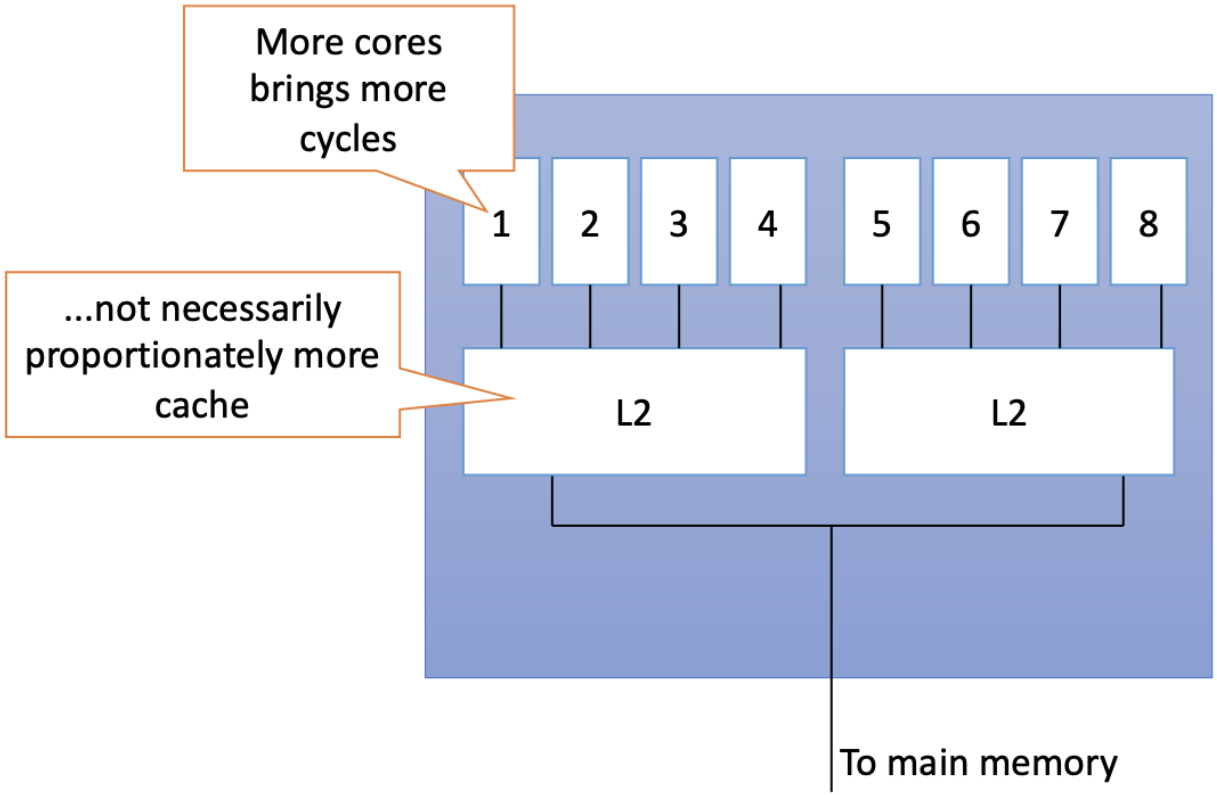
- **NUMA** (Non-Uniform-Memory-Access)
  - **Idea**: each CPU has its own memory that it can access faster
  - Accessing **local memory** has lower latency and higher bandwidth
  - Accessing **remote memory** introduces latency

- So the latency is **not the same** (not uniform) when accessing memory: thus NUMA

NUMA

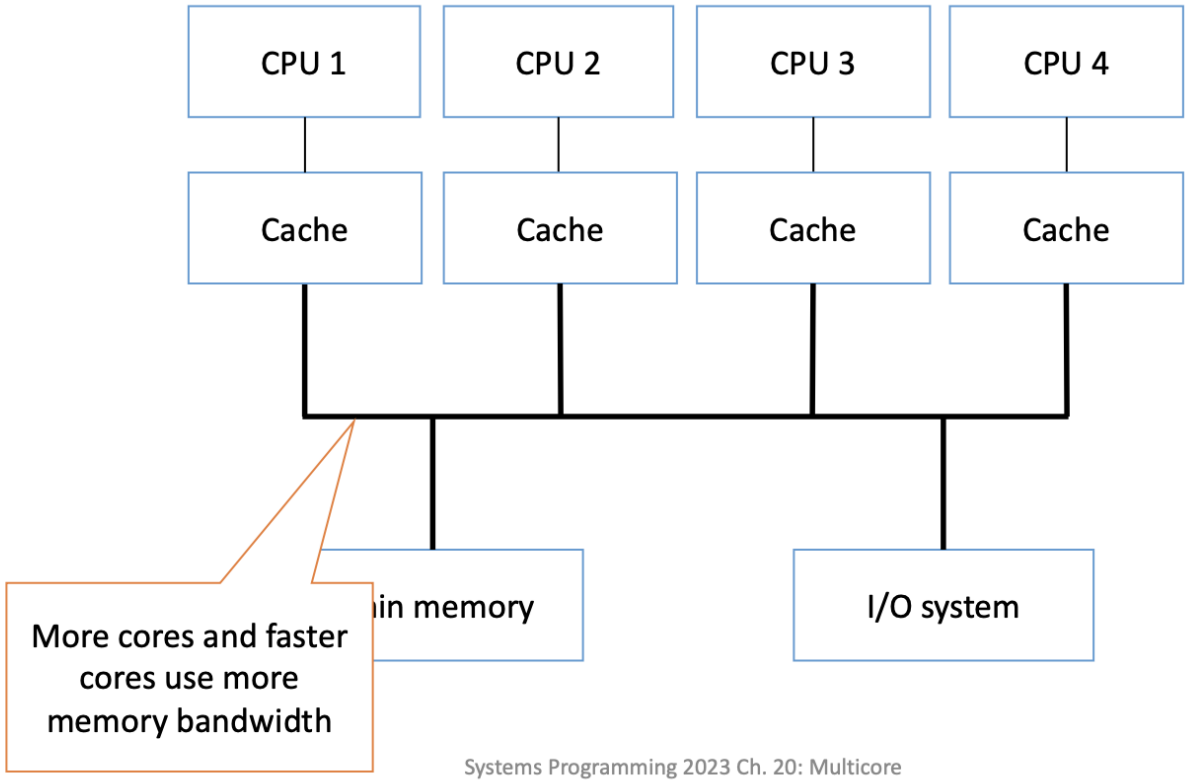
Systems@ETH zürich

SMP architecture



NUMA

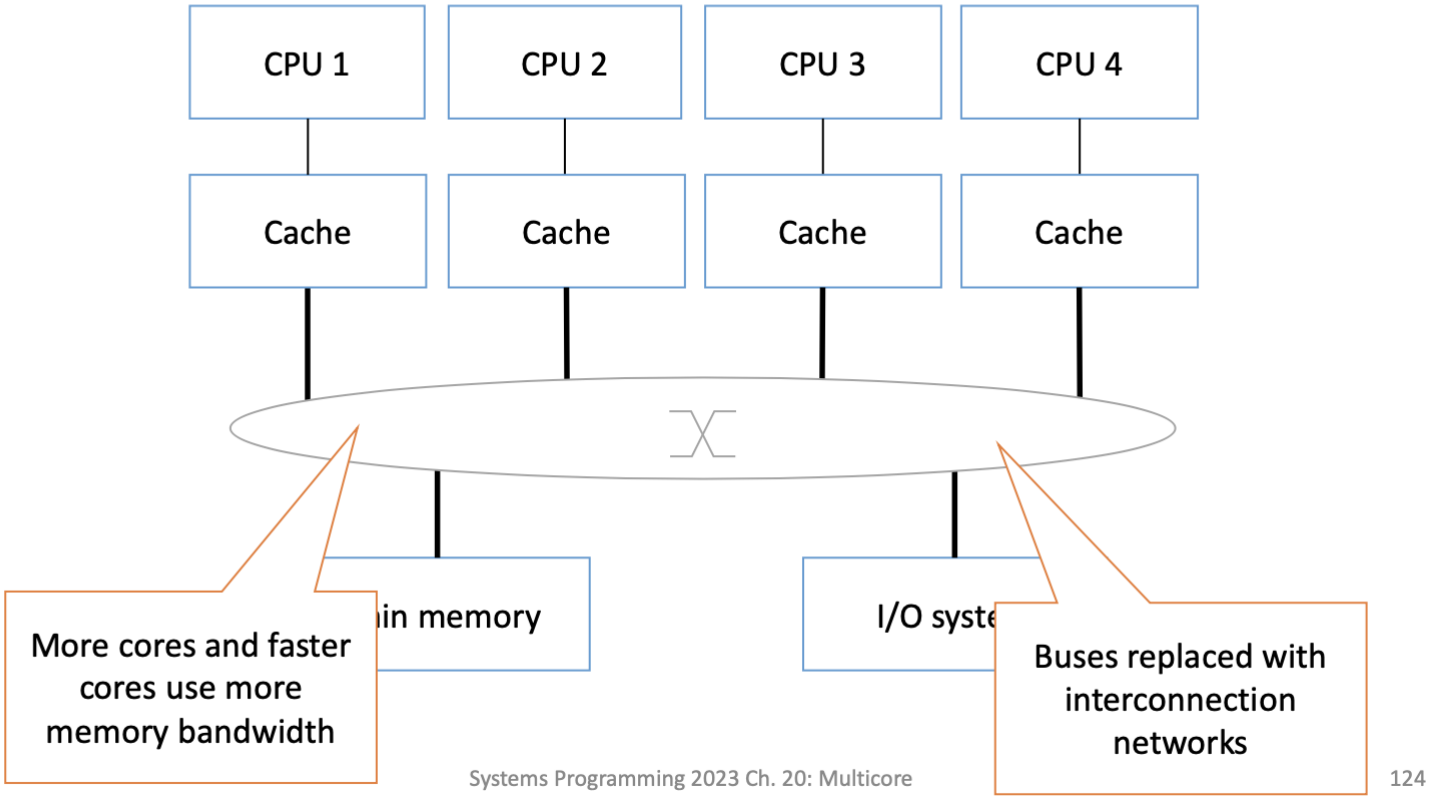
SMP architecture



- Until now we used a **BUS** (which is broadcast)

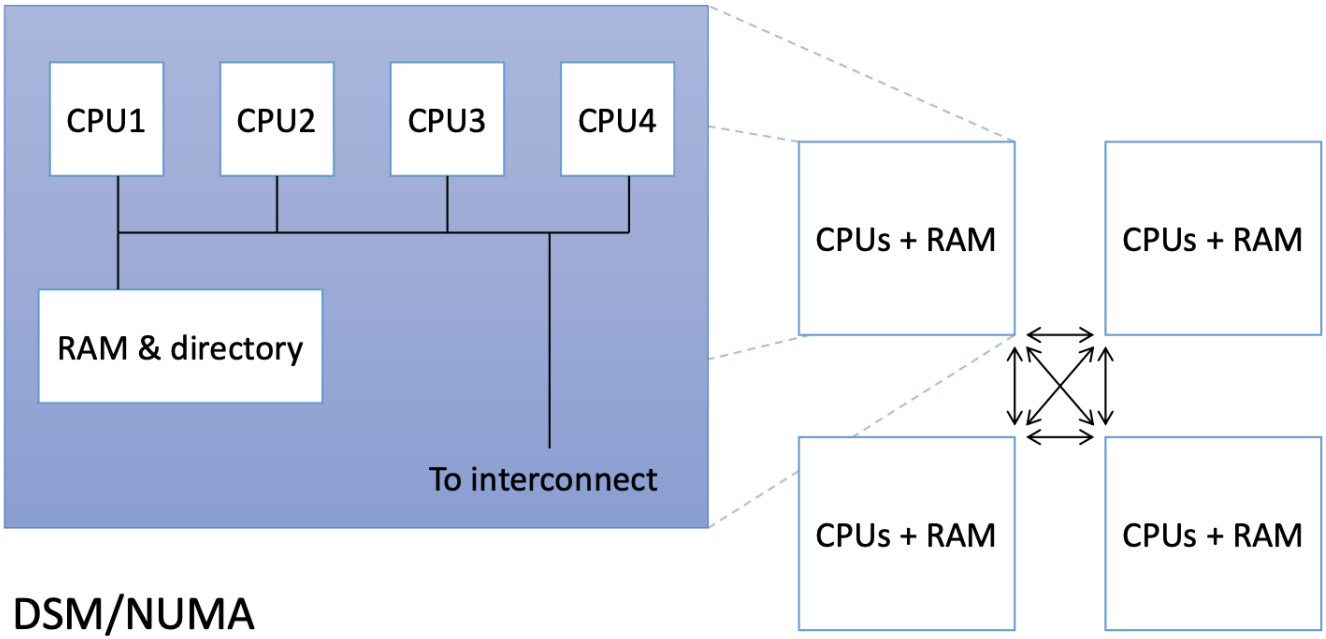
NUMA

SMP architecture



- **Idea**: Use interconnect (no broadcast)

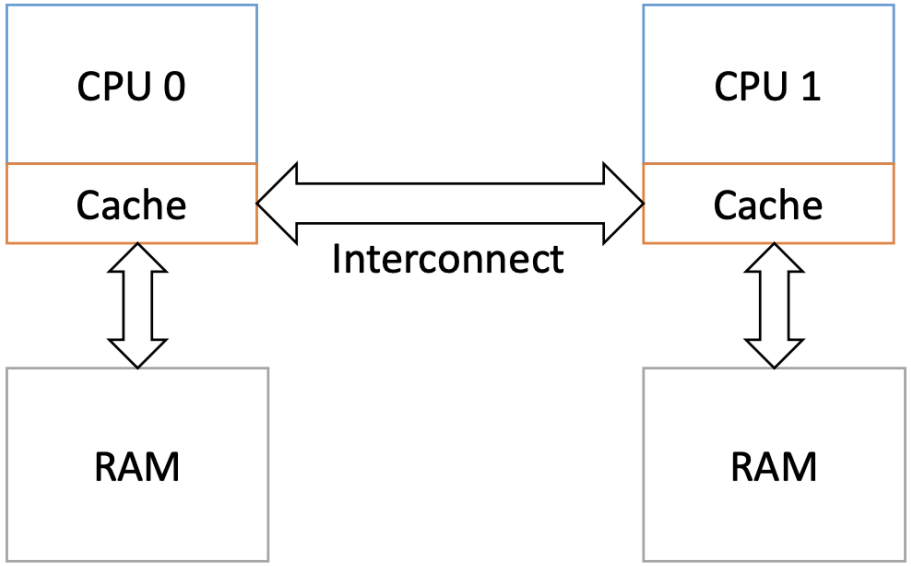
NUMA



- DSM/NUMA
- Message-passing, eg clusters
- Could scale to 100s or 1000s of cores

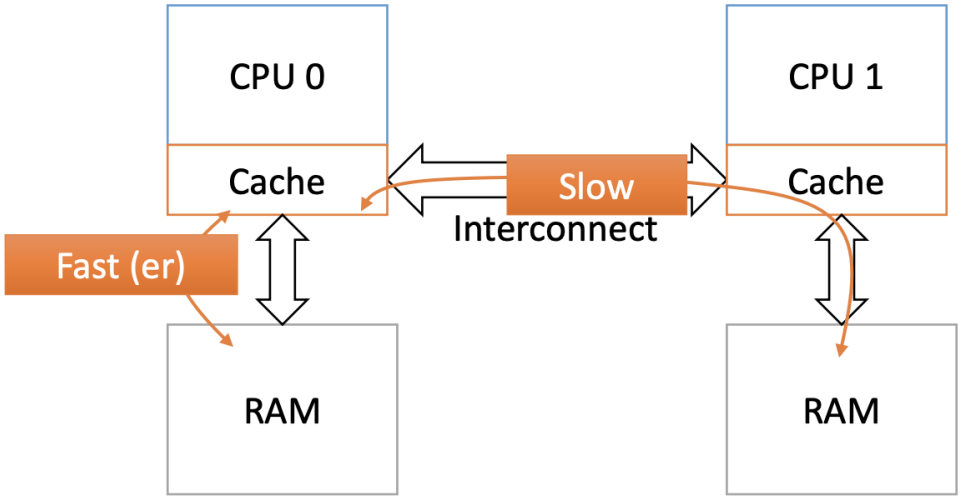
- **Idea**: Divide multiple processors into one node (NUMA Node)
- Give each **Numa Node** its **own** part of physical memory (RAM)

NUMA



- **How many CPUs** we want per node can vary
- **Here**: we only put one CPU per NUMA node

NUMA

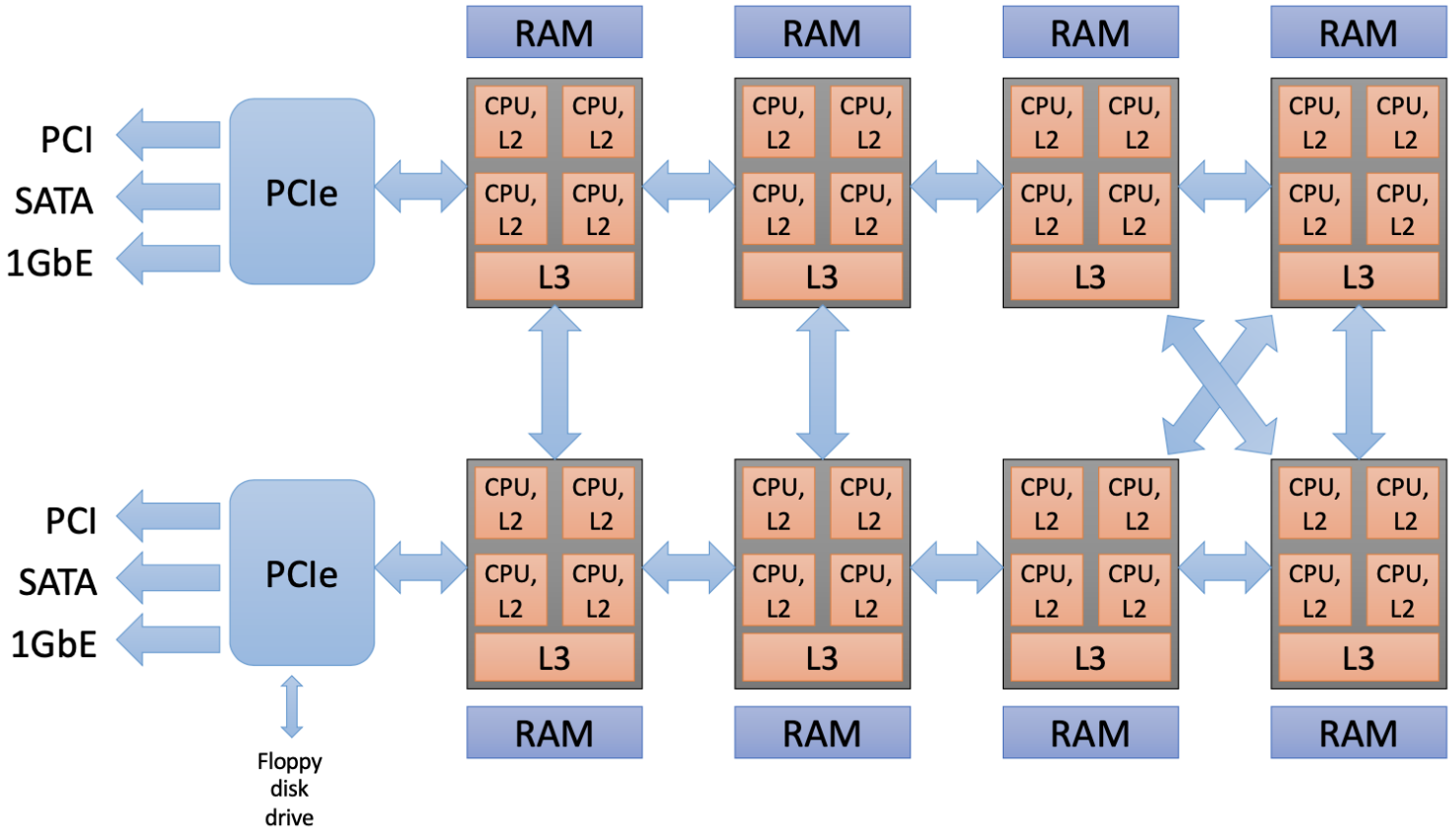


- Interconnect is not a bus any more: it's a network link
  - Carries messages between **nodes** (usually processor sockets)
  - Read/write request/response, cache invalidate, etc.
- All memory is globally addressable
  - But local is faster (to varying degrees)

NUMA

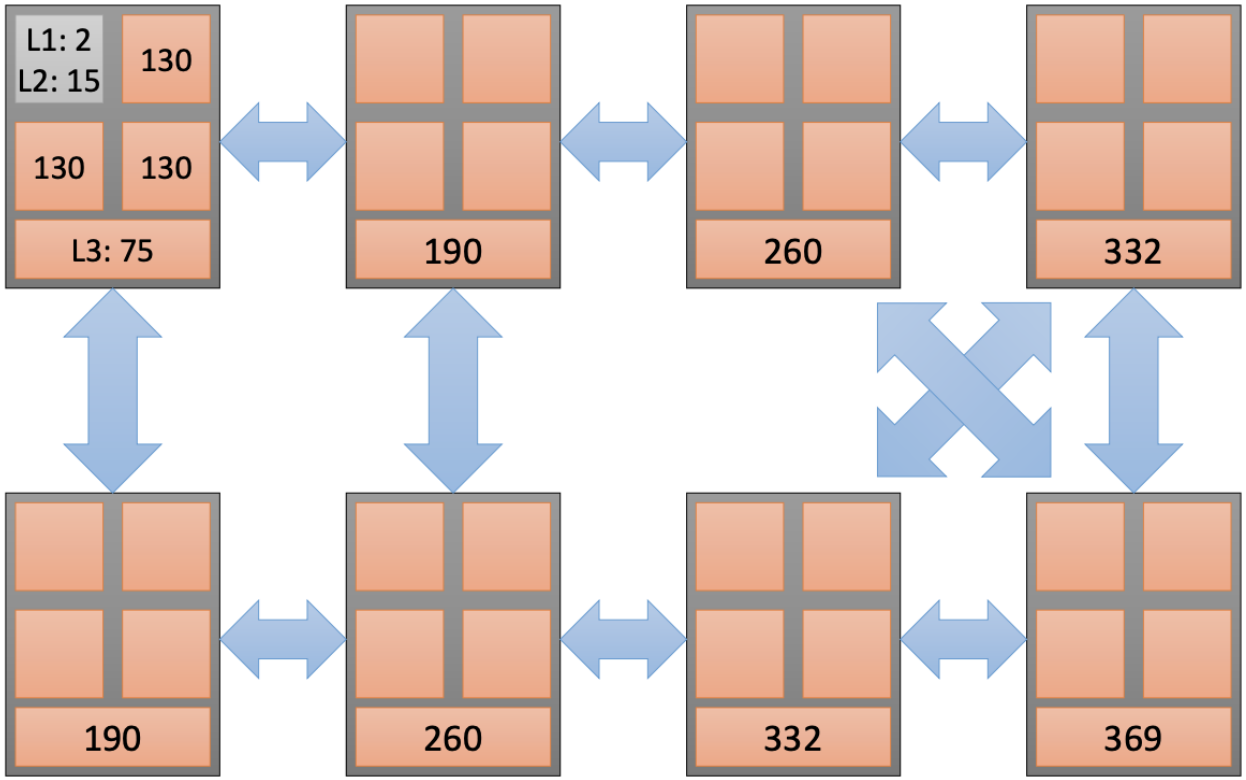
Systems@ETH zürich

8-socket 32-core AMD Barcelona (c.2007)



NUMA

Cache access latency



Memory is a bit faster than L3, but it's complicated…

NUMA

- What issue does NUMA induce?

NUMA

Systems@ETH zürich

- What issue does NUMA induce?

- Our **cache coherence protocol** have an issue: if we have a **interconnect** instead of a **BUS** things don't get broadcast anymore

- 1. **Bus emulation**
- 2. **Cache Directory**

NUMA

Systems@ETH zürich

- What issue does NUMA induce?
- Our **cache coherence protocol** have an issue: if we have a **interconnect** instead of a **BUS** things don't get broadcast anymore
- 1. **Bus emulation**
- 2. **Cache Directory**

NUMA

Systems@ETH zürich

NUMA cache coherence

Can't snoop on the bus any more: it's not a bus!

- NUMA use a message-passing interconnect

Much more complicated than it sounds…

Solution 1: Bus emulation

- Similar to snooping, but without a shared bus
- Each node sends a message to all other nodes
  - E.g. "Read exclusive"
- Waits for a reply from all nodes before proceeding
  - E.g. "Acknowledge"
- Example: AMD coherent HyperTransport

NUMA

Systems@ETH zürich

- What issue does NUMA induce?

- Our **cache coherence protocol** have an issue: if we have a **interconnect** instead of a **BUS** things don't get broadcast anymore

- 1. **Bus emulation**
- 2. **Cache Directory**

NUMA

Systems@ETH zürich

- **Idea**: the home node maintains a **directory** of the other nodes which currently have the line
  - They store **node ID** of the owner
  - **1 bit per node** indicating of presence of the line

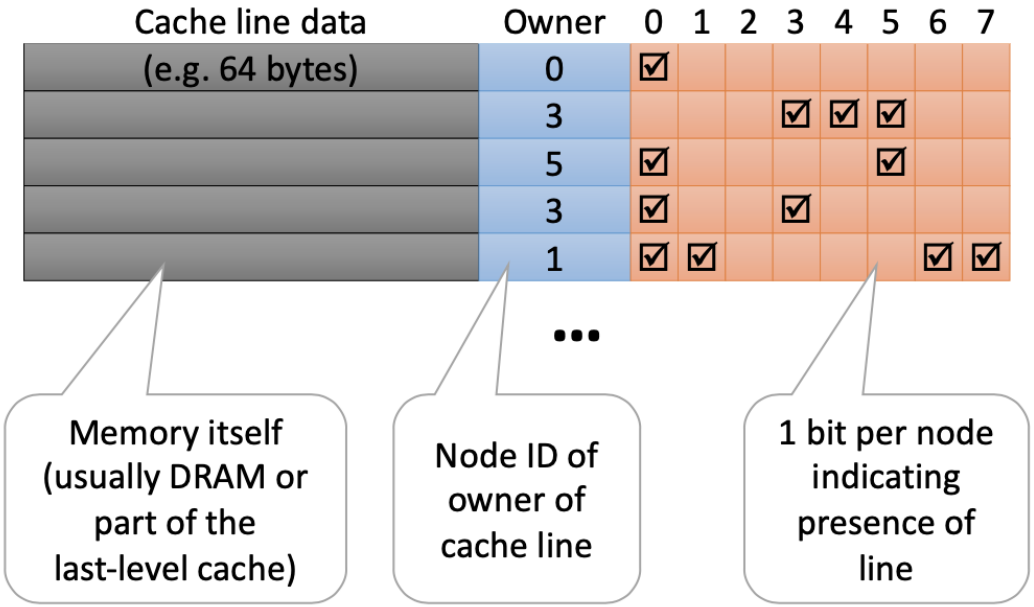
- Its like having a ”per node” cache coherence system: where each node watches out for its assigned memory

NUMA

Systems@ETH zürich

Solution 2: Cache Directory

- Augment each node's local memory with a cache directory:



NUMA: Practical

NUMA

Systems@ETH zürich

- This is **not** just some theoretical concept having no relevance for you in practice
- Let us look at some **actual processor features**

- 1. **Maximus and Euler login node**
- 2. **Piora Cluster, Swiss National Supercomputing Center**

NUMA

Systems@ETH zürich

- This is **not** just some theoretical concept having no relevance for you in practice
- Let us look at some **actual processor features**

- 1. **Maximus and Euler login node**
- 2. **Piora Cluster, Swiss National Supercomputing Center**

NUMA

Systems@ETH zürich

```
falkbe@maximus:~$ lscpu
Architecture:                    x86_64
  CPU op-mode(s):                32-bit, 64-bit
  Address sizes:                 43 bits physical, 48 bits virtual
  Byte Order:                    Little Endian
CPU(s):                          2
  On-line CPU(s) list:           0,1
Vendor ID:                       GenuineIntel
  Model name:                    Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz
    CPU family:                  6
    Model:                       85
    Thread(s) per core:          1
    Core(s) per socket:          1
    Socket(s):                   2
Caches (sum of all):
  L1d:                           64 KiB (2 instances)
  L1i:                           64 KiB (2 instances)
  L2:                            2 MiB (2 instances)
  L3:                            49.5 MiB (2 instances)
NUMA:
  NUMA node(s):                  1
  NUMA node0 CPU(s):             0,1
```

```
falkbe@maximus:~$ numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1
node 0 size: 4877 MB
node 0 free: 2532 MB
node distances:
node   0
  0:  10
```

NUMA

```
falkbe@eu-login-41:~$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   4
  On-line CPU(s) list:    0-3
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Xeon(R) CPU E3-1284L v4 @ 2.90GHz
Caches (sum of all):
  L1d:                    128 KiB (4 instances)
  L1i:                    128 KiB (4 instances)
  L2:                     1 MiB (4 instances)
  L3:                     6 MiB (1 instance)
  L4:                     128 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-3
```

```
falkbe@eu-login-41:~$ numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 32020 MB
node 0 free: 10519 MB
node distances:
node   0
  0:  10
```

NUMA

Systems@ETH zürich

- This is **not** just some theoretical concept having no relevance for you in practice
- Let us look at some **actual processor features**

- 1. **Maximus and Euler login node**
- 2. **Piora Cluster, Swiss National Supercomputing Center**

```
[bfalk@piora5 ~]$ lscpu
Architecture:                    x86_64
  CPU op-mode(s):                32-bit, 64-bit
  Address sizes:                 43 bits physical, 48 bits virtual
  Byte Order:                    Little Endian
CPU(s):                          128
  On-line CPU(s) list:           0-127
Vendor ID:                       AuthenticAMD
  Model name:                    AMD EPYC 7742 64-Core Processor
    CPU family:                  23
    Model:                       49
    Thread(s) per core:          1
    Core(s) per socket:          64
    Socket(s):                   2
Virtualization features:
  Virtualization:                AMD-V
Caches (sum of all):
  L1d:                           4 MiB (128 instances)
  L1i:                           4 MiB (128 instances)
  L2:                            64 MiB (128 instances)
  L3:                            512 MiB (32 instances)
NUMA:
  NUMA node(s):                  8
  NUMA node0 CPU(s):             0-15
  NUMA node1 CPU(s):             16-31
  NUMA node2 CPU(s):             32-47
  NUMA node3 CPU(s):             48-63
  NUMA node4 CPU(s):             64-79
  NUMA node5 CPU(s):             80-95
  NUMA node6 CPU(s):             96-111
  NUMA node7 CPU(s):             112-127
```

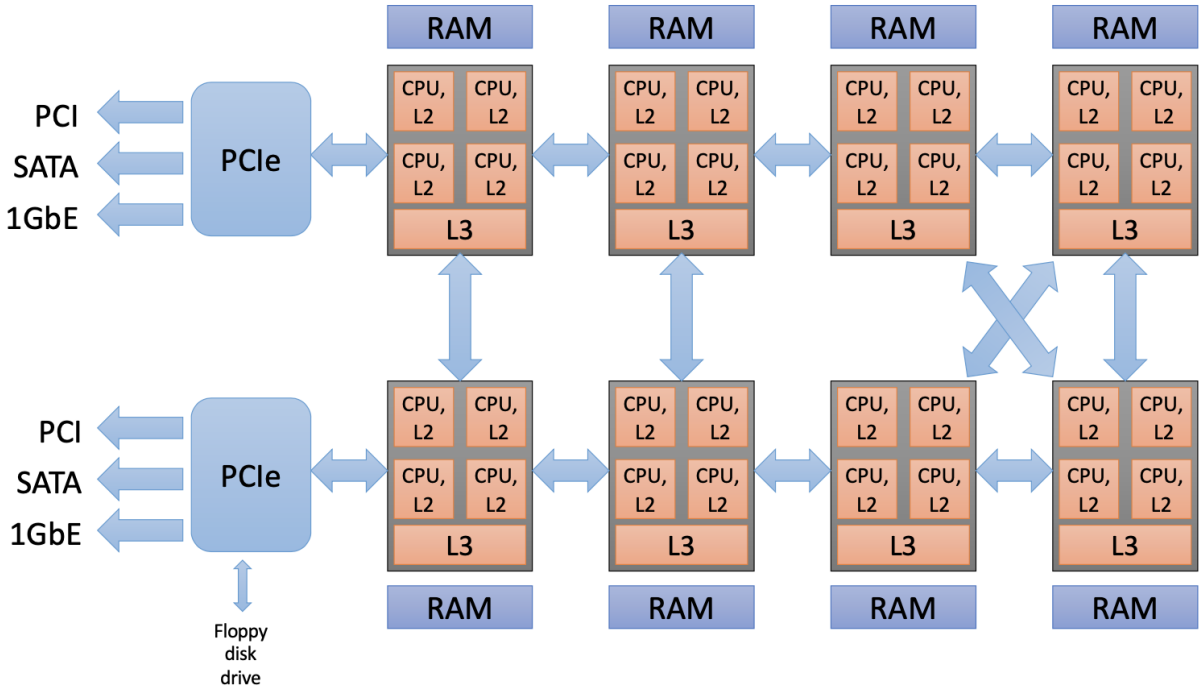
Systems@ETH zürich

```
[bfalk@piora5 ~]$ lstopo
Machine (504GB total)
  Package L#0
    Group0 L#0
      NUMANode L#0 (P#0 63GB)
      L3 L#0 (16MB)
        L2 L#0 (512KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
        L2 L#1 (512KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
        L2 L#2 (512KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
        L2 L#3 (512KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)
      L3 L#1 (16MB)
        L2 L#4 (512KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4 + PU L#4 (P#4)
        L2 L#5 (512KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5 + PU L#5 (P#5)
        L2 L#6 (512KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6 + PU L#6 (P#6)
        L2 L#7 (512KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7 + PU L#7 (P#7)
      L3 L#2 (16MB)
        L2 L#8 (512KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8 + PU L#8 (P#8)
        L2 L#9 (512KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9 + PU L#9 (P#9)
        L2 L#10 (512KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10 + PU L#10 (P#10)
        L2 L#11 (512KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11 + PU L#11 (P#11)
      L3 L#3 (16MB)
        L2 L#12 (512KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12 + PU L#12 (P#12)
        L2 L#13 (512KB) + L1d L#13 (32KB) + L1i L#13 (32KB) + Core L#13 + PU L#13 (P#13)
        L2 L#14 (512KB) + L1d L#14 (32KB) + L1i L#14 (32KB) + Core L#14 + PU L#14 (P#14)
        L2 L#15 (512KB) + L1d L#15 (32KB) + L1i L#15 (32KB) + Core L#15 + PU L#15 (P#15)
      HostBridge
        PCIBridge
          PCI 62:00.0 (Ethernet)
            Net "enp98s0f0"
          PCI 62:00.1 (Ethernet)
            Net "enp98s0f1"
        PCIBridge
          PCIBridge
            PCI 65:00.0 (VGA)
```

NUMA

- This is the structure **we have just seen** (note its not the same processor, but the concepts are the same)

8-socket 32-core AMD Barcelona (c.2007)

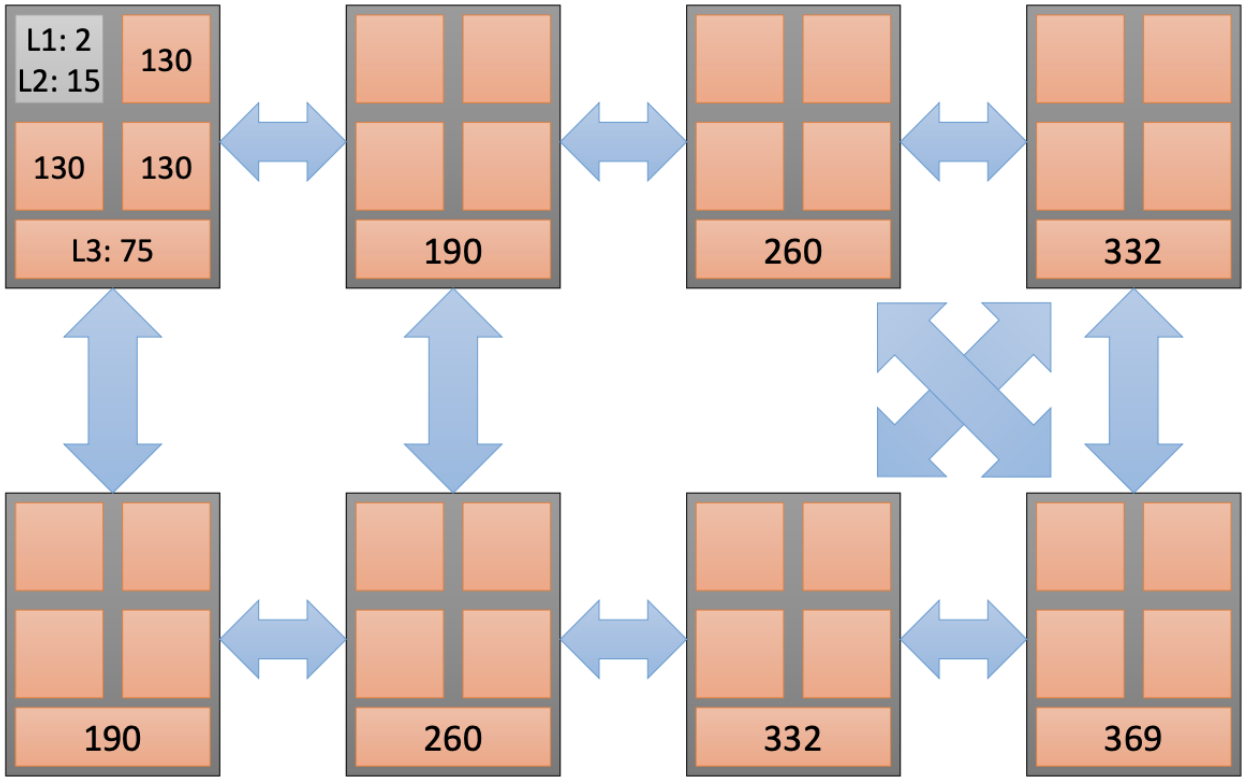


```
[bfalk@piora5 ~]$ numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
node 0 size: 64323 MB
node 0 free: 58103 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
node 1 size: 64465 MB
node 1 free: 62236 MB
node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 2 size: 64506 MB
node 2 free: 62096 MB
node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
node 3 size: 64494 MB
node 3 free: 62380 MB
node 4 cpus: 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
node 4 size: 64506 MB
node 4 free: 46135 MB
node 5 cpus: 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
node 5 size: 64506 MB
node 5 free: 62674 MB
node 6 cpus: 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
node 6 size: 64506 MB
node 6 free: 63770 MB
node 7 cpus: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
node 7 size: 64502 MB
node 7 free: 61637 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  12  12  12  32  32  32  32
  1:  12  10  12  12  32  32  32  32
  2:  12  12  10  12  32  32  32  32
  3:  12  12  12  10  32  32  32  32
  4:  32  32  32  32  10  12  12  12
  5:  32  32  32  32  12  10  12  12
  6:  32  32  32  32  12  12  10  12
  7:  32  32  32  32  12  12  12  10
```

Systems@ETH zürich

NUMA

Cache access latency



Memory is a bit faster than L3, but it's complicated…

Overview

Systems@ETH zürich

- Part of Assignment 10
- Recap Cache Coherence
- Hints for Assignment 11
- Quiz

Overview

- Part of Assignment 10
- Recap Cache Coherence
- Hints for Assignment 11
- Quiz

Question 4)

Systems@ETH zürich

- Exam Hint:
  - Do some pre-processing of the given values
  - Bit numbers of: VPN, VPO, PPN, PPO
  - TLB: Number of sets / entries per set
  - Cache: Offset, Index and Tag bits

Question 4) Walk-through

Systems@ETH zürich

Assumptions (we have a TLB and L1 Cache)

- Byte addressable memory
- The page size is **64** bytes
- Virtual addresses are **14** bits wide
- Physical addresses are **12** bits wide
- TLB is **2**-way associative with **8** total entries
- The L1 (data) cache is **physically** addressed, direct mapped, and has a **4**-byte block size; there are **16** sets.

Conclusions

⇒ #offset bits = **6**

⇒ #VPN bits = **14** – **6** = **8**

⇒ #PPN bits = **12** – **6** = 6

⇒ TLB has **4** sets

#TLBI bits = **2**

#TLBT bits = **8** – **2** = 6

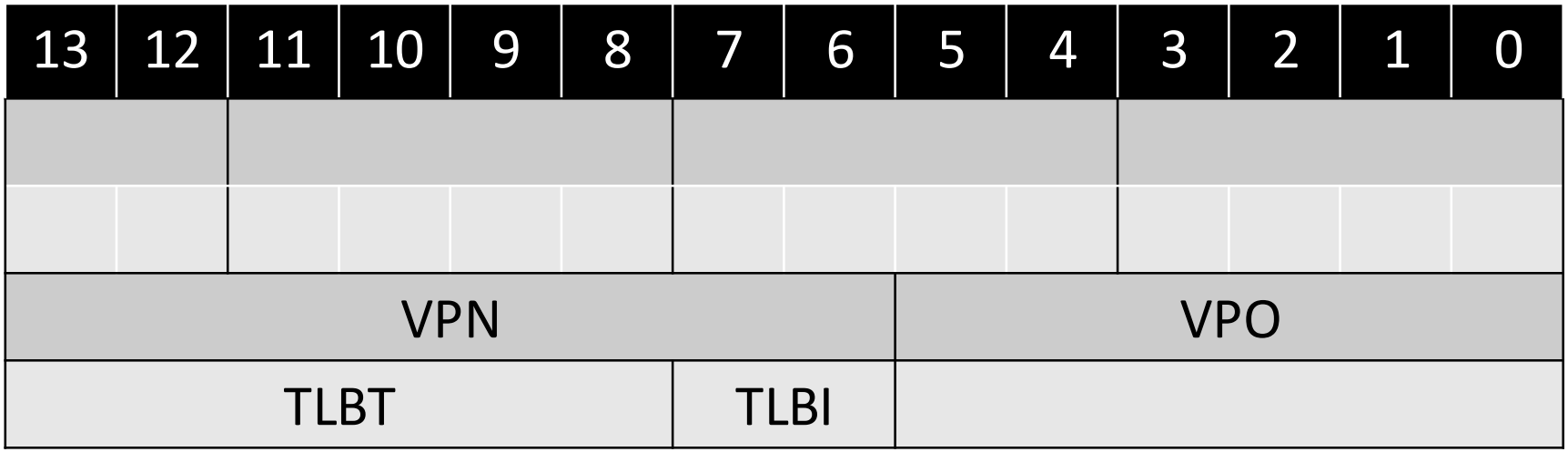
⇒ #CO bits = **2**

#CI bits = **4**

#CT bits = **12** – **2** – **4** = 6

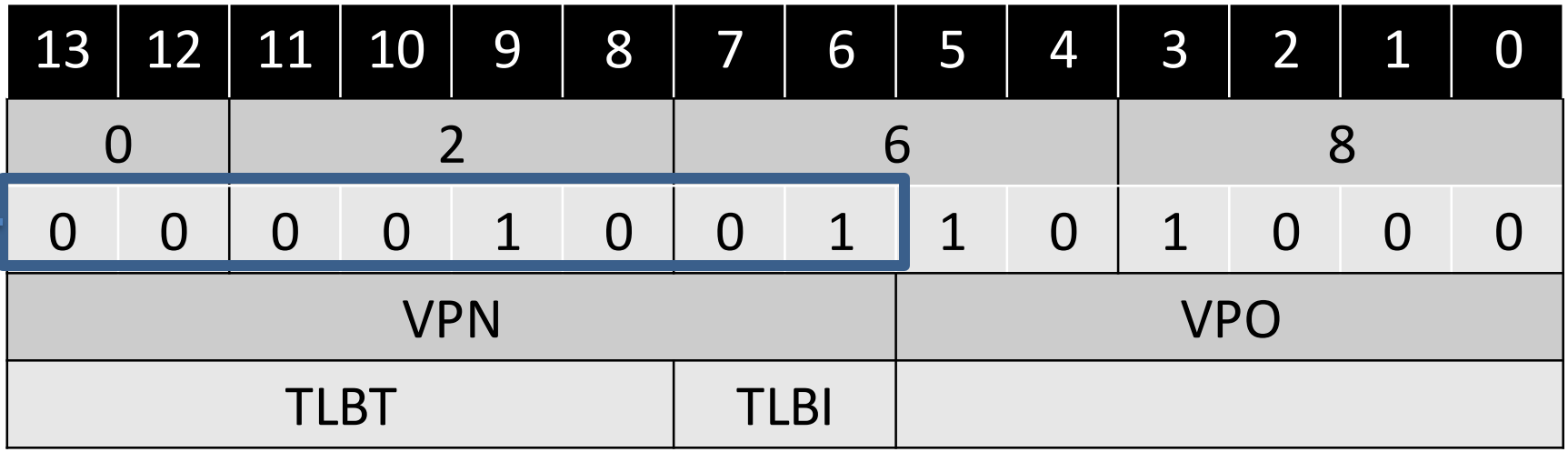
Question 4a) Walk-through

- VA=0x268



Question 4a) Walk-through

- VA=0x268



Page table: (all values are hexadecimal)

VPN	PPN	Valid
00		
01	02	1
02	03	1
03		
04		
05		
06	22	1
07	22	1

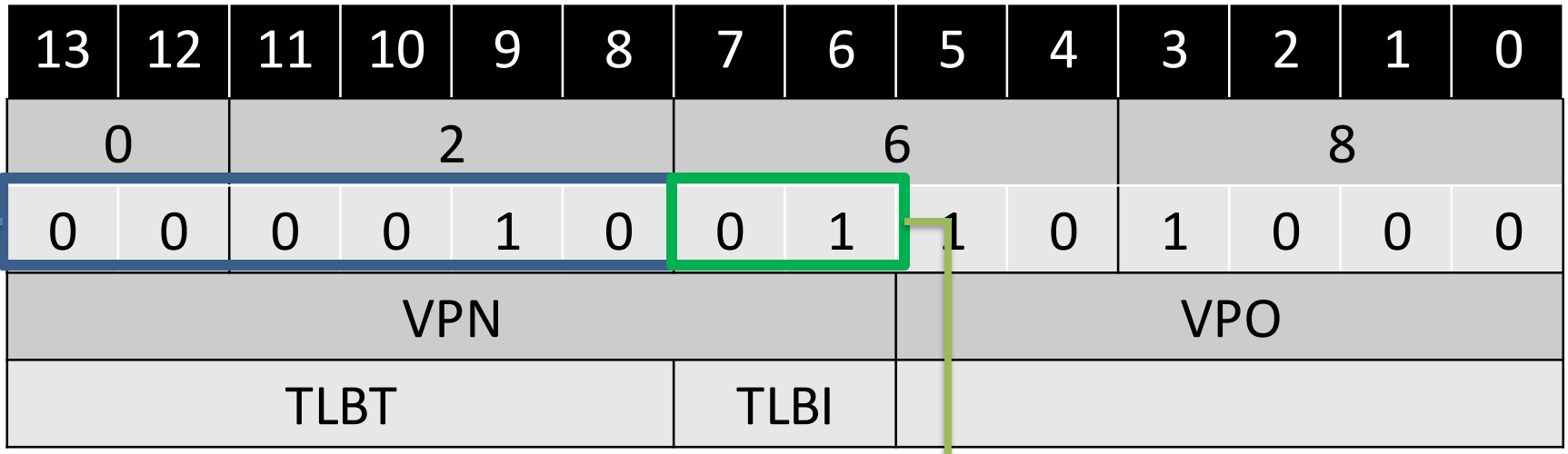
VPN = 0x9

VPN	PPN	Valid
08		
09	20	1
0a		
0b		
0c		
0d	04	1
0e		
0f		

HIT

Question 4a) Walk-through

- VA=0x268



TLBT= 0x2

TLBI= 0x1

TLB: (All values are hexadecimal)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	0	22	1	07		0
3	01	22	1	02		0

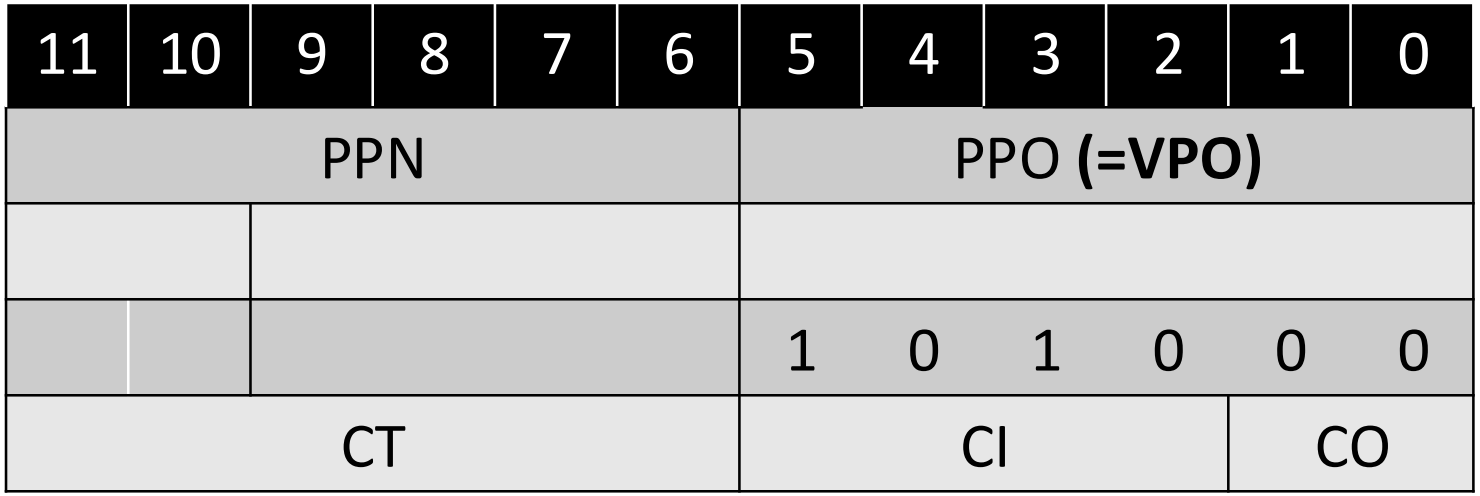
HIT

Question 4a) Walk-through

- VA=0x268

TLB: (all values are hexadecimal)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	01	22	1	07		0
3	01	22	1	02		0

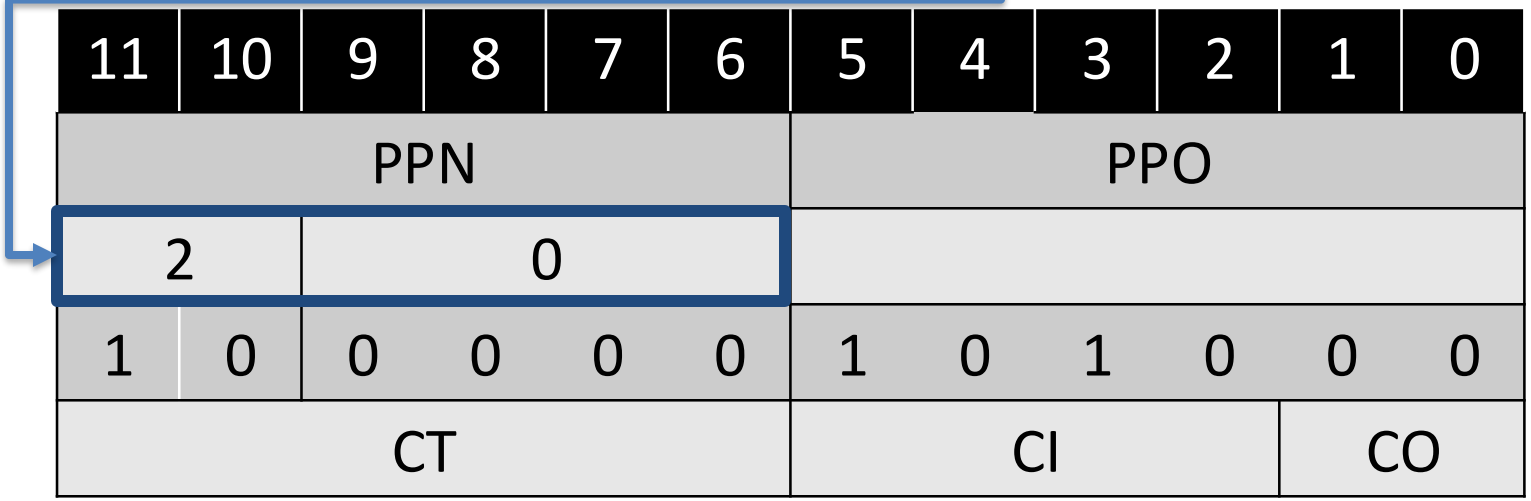


Question 4a) Walk-through

- VA=0x268

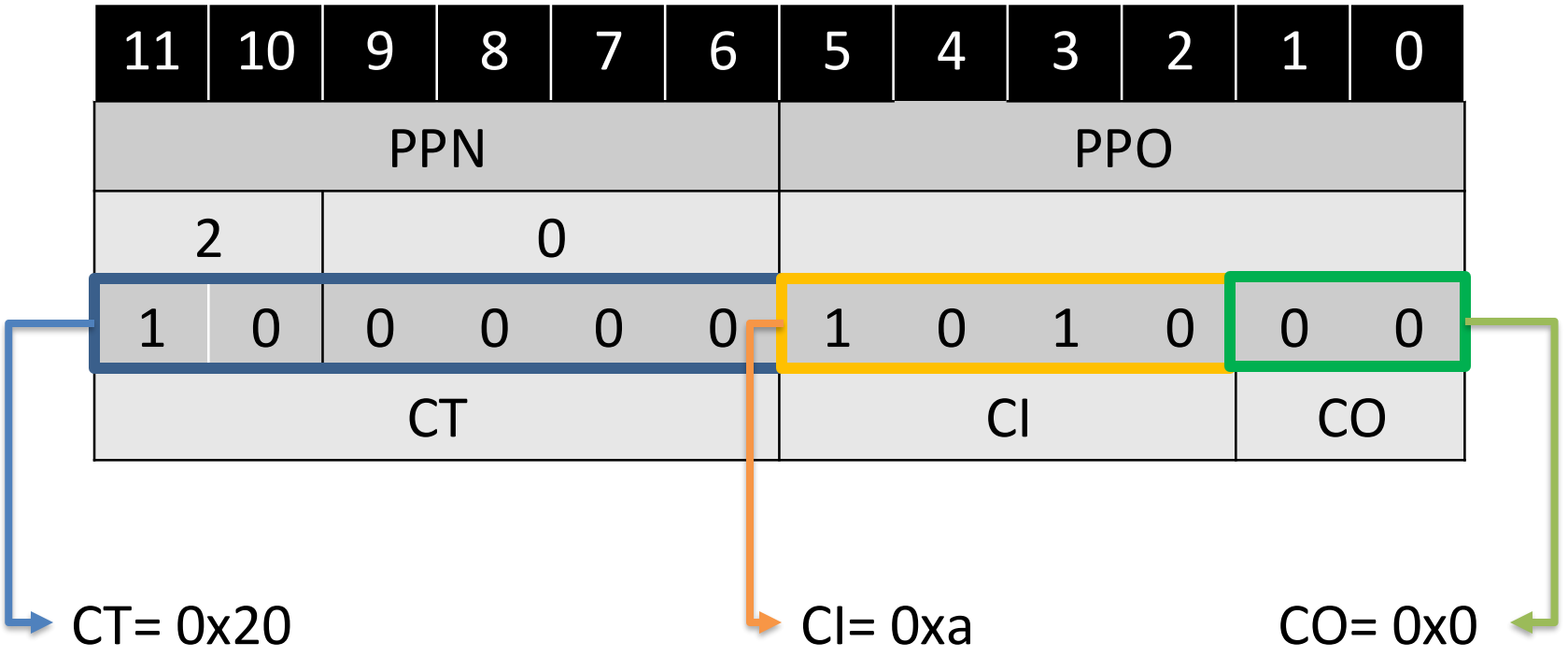
TLB: (all values are hexadecimal)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	01	22	1	07		0
3	01	22	1	02		0



Question 4a) Walk-through

- VA=0x268



Question 4a) Walk-through

- VA=0x268

Cache: (all values are hexadecimal)

Index	Tag	Valid	Block[0]	Block[1]	Block[2]	Block[3]
00	00	1	de	ad	fa	ce
01	31	0				
02	24	1	02	13	e1	de
03	22	1	22	23	e2	2e
04	21	0				
05	22	0				
06	18	0				
07	22	1	9a	01	00	de
08	20	0				
09	32	1	83	1a	09	ce
0a	20	1	0d	1f	f1	d0
0b	3a	0				
0c	3f	0				
0d	24	1	be	fb	57	02
0e	23	0				
0f	22	1	cf	7a	9b	a0

CI= 0xa

CT= 0x20

CO= 0x0

HIT

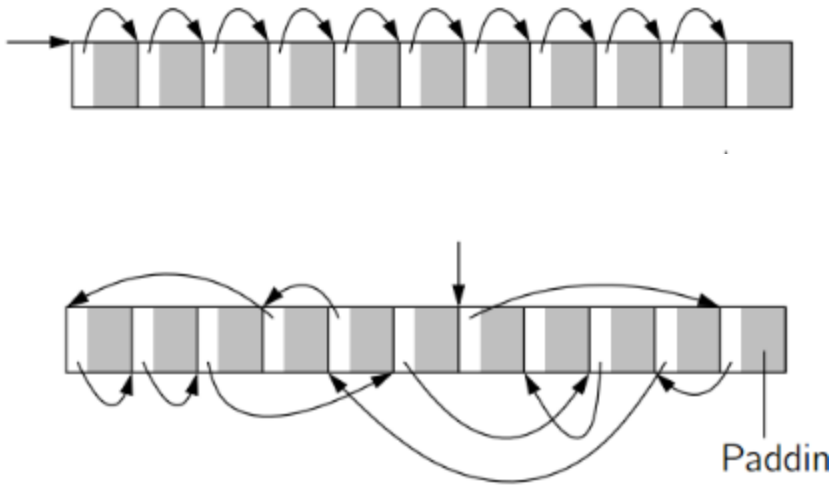
Systems@ETH Zürich

Interlude: Reverse Engineering Caches

Getting the Cache Line Size

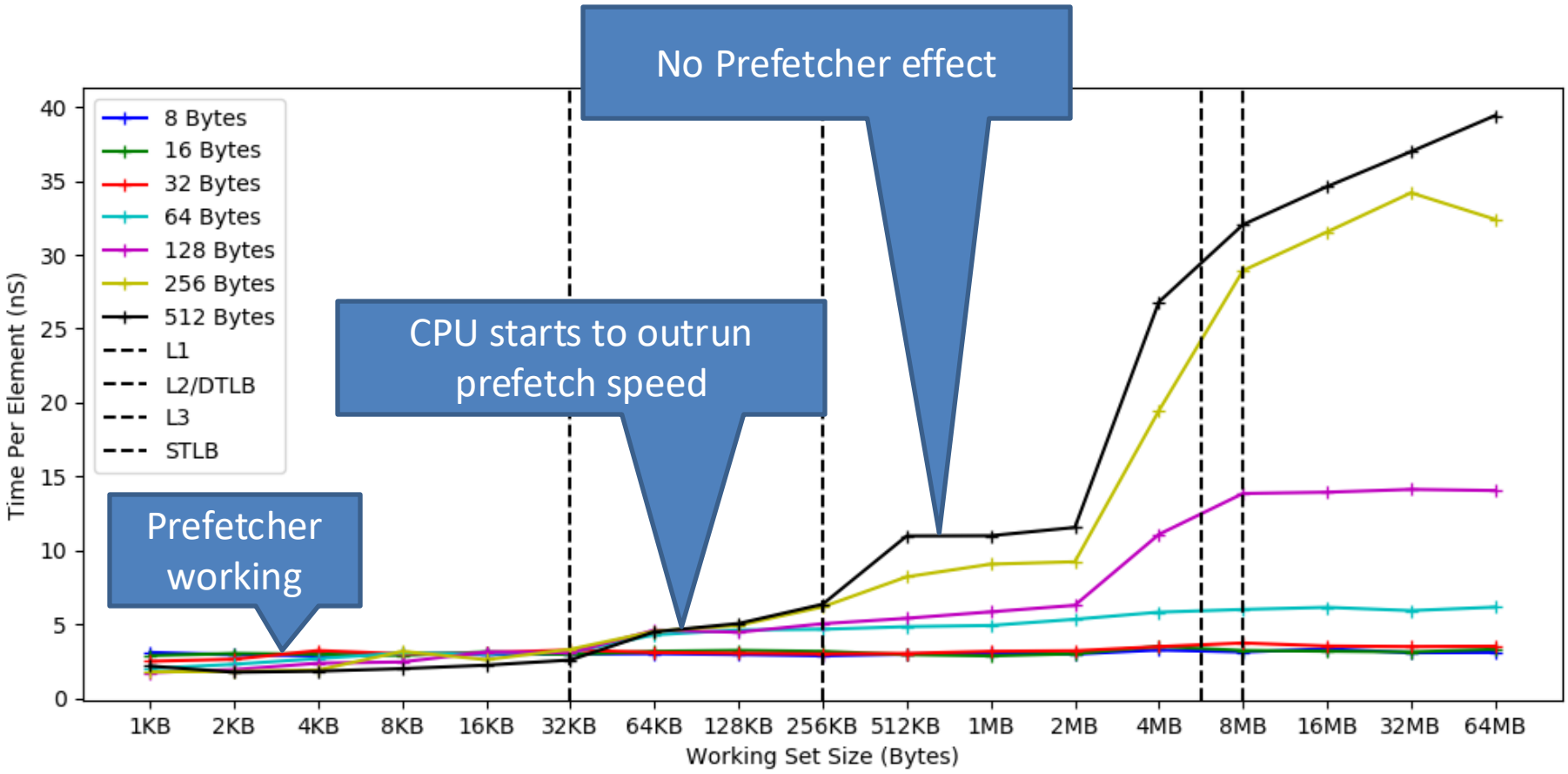
- Experiment: Traversal of a linked list with varying element size

```
1. struct listelem {
2.     struct listelem *next;
3.     uint64_t padding[NUMPAD];
4. }
```

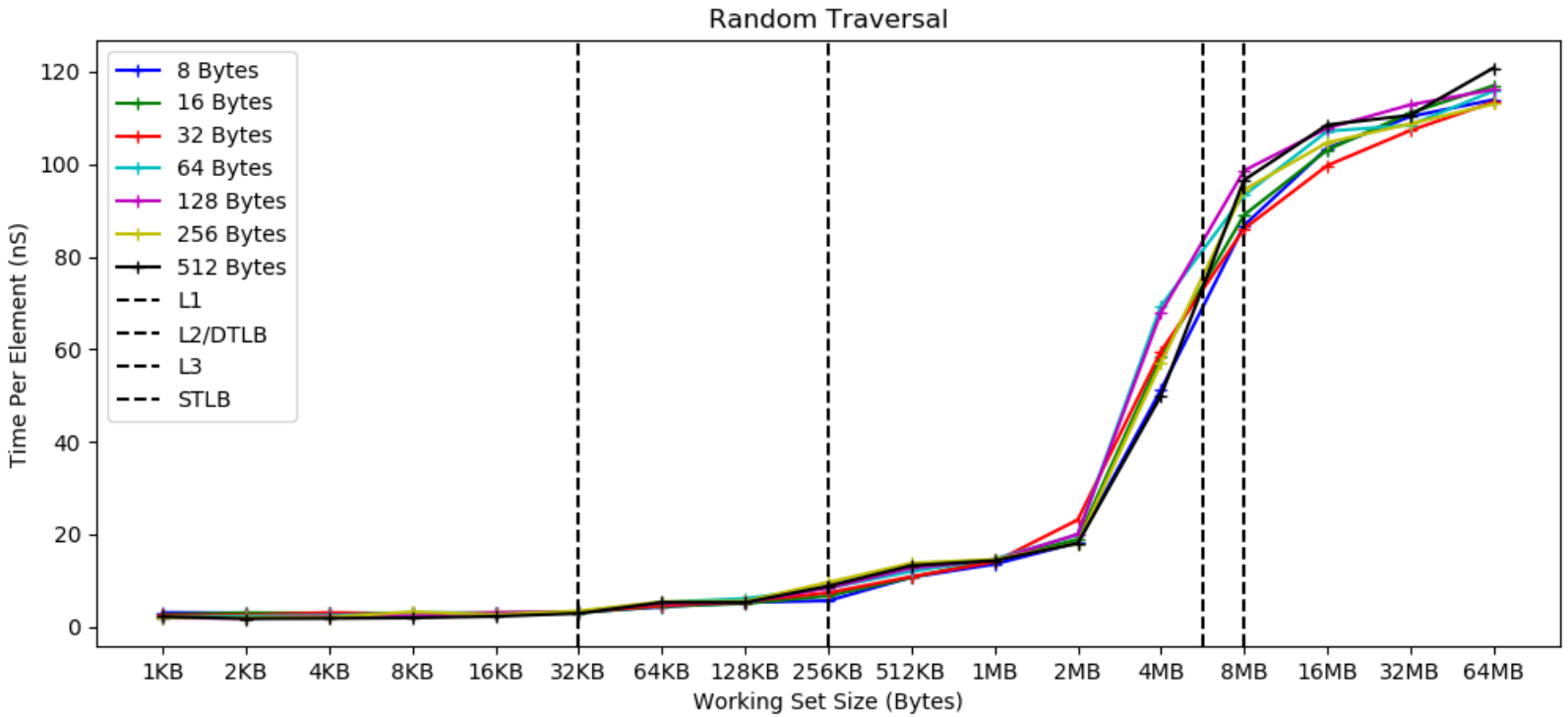


Source: Jens Teubner, Data Processing on Modern Hardware, Fall 2012

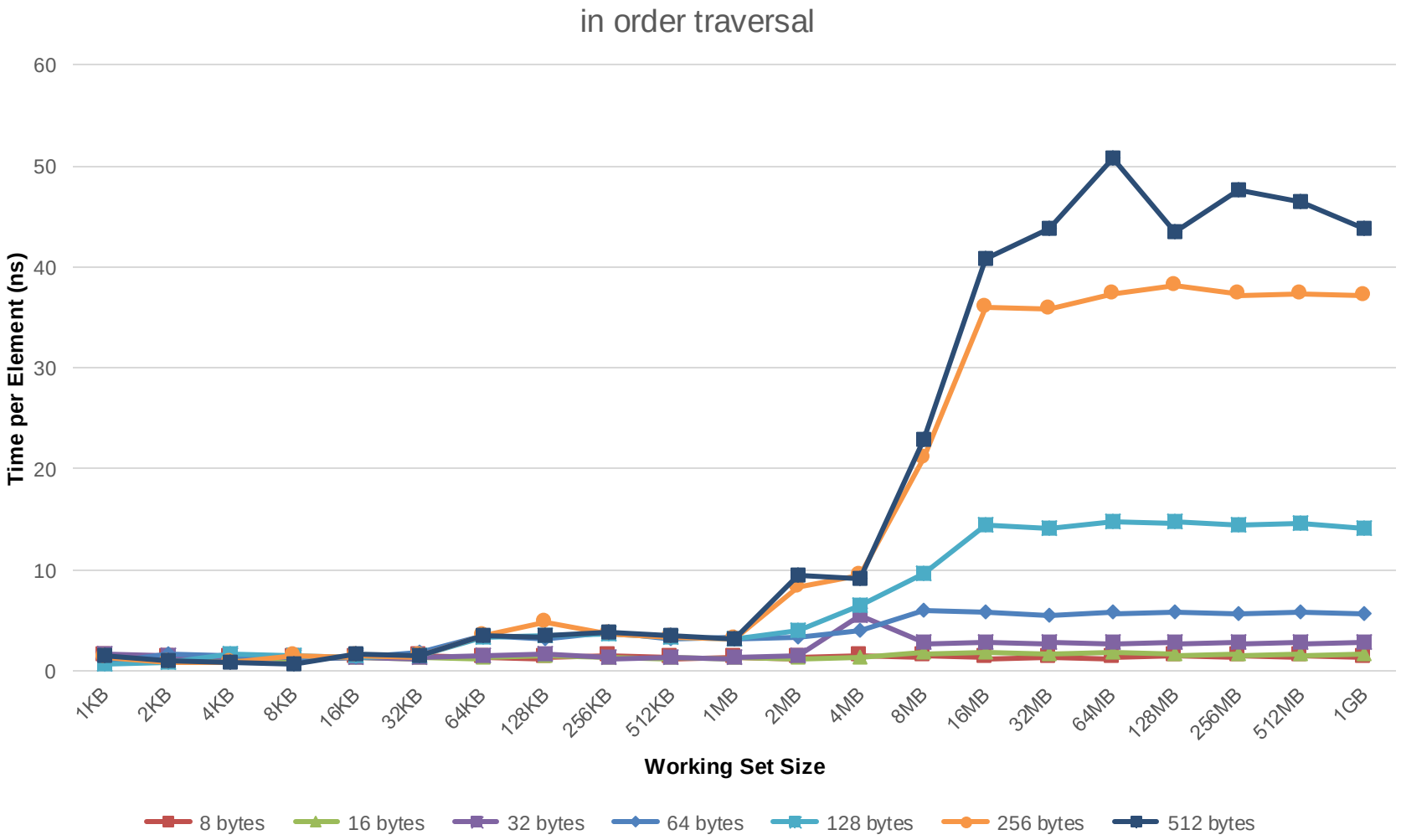
Experiment Results: Sequential



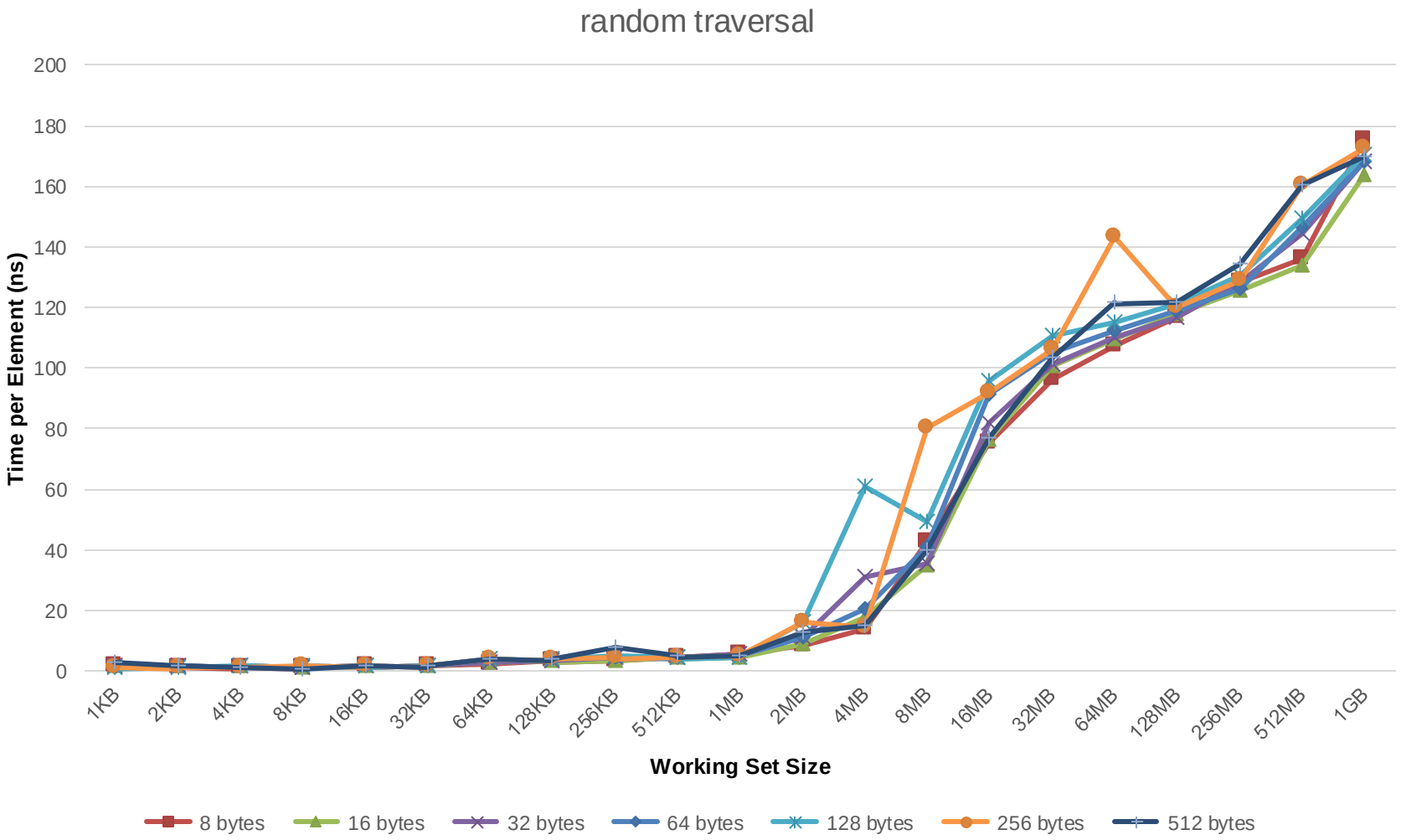
Experiment Results: Random



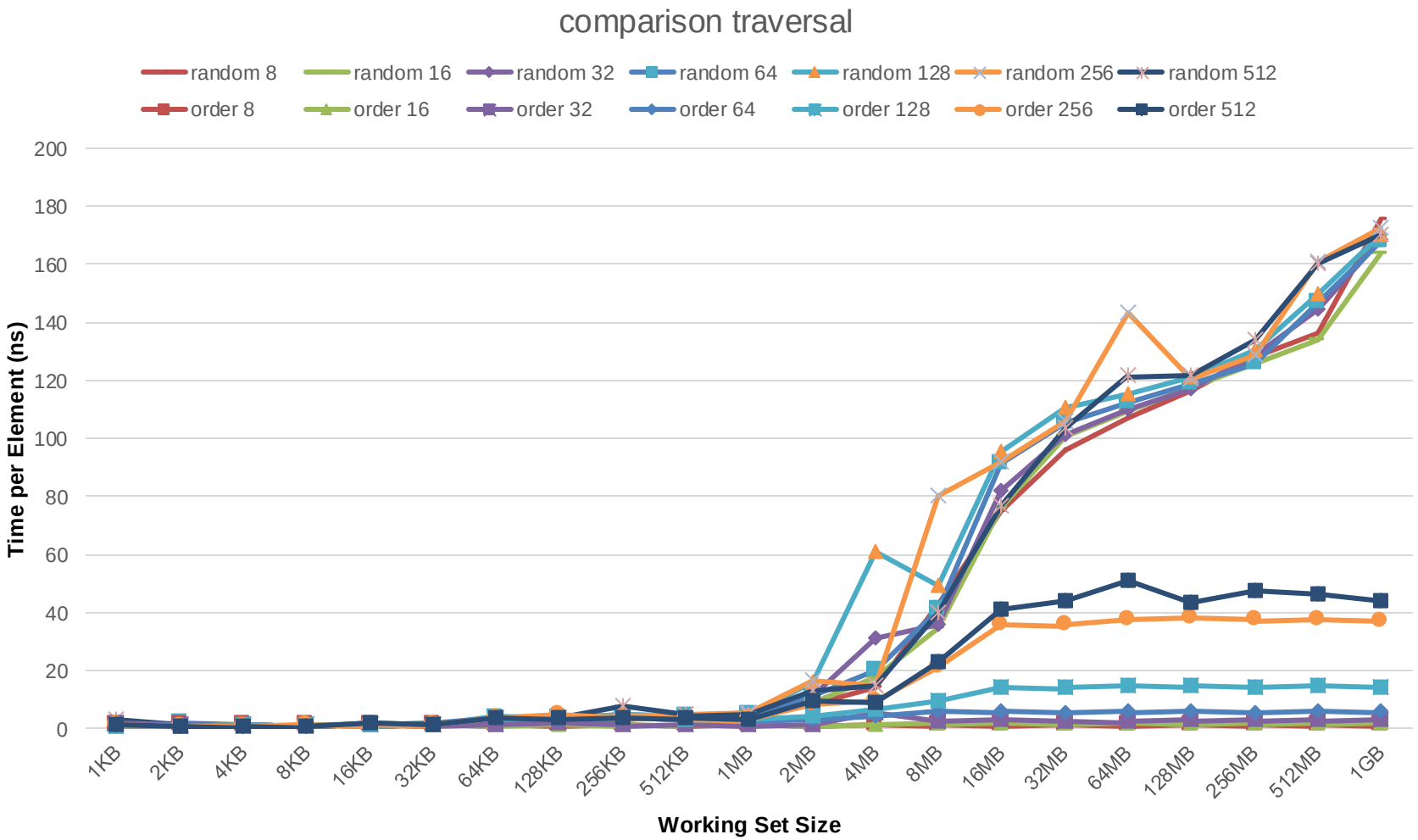
Sequential (i5-1135G7 2.42GHz)



Random (i5-1135G7 2.42GHz)



Comparison (i5-1135G7 2.42GHz)



Getting Information about the Cache

Systems@ETH zürich

Linux:

Read files in /sys/devices/system/cpu/cpu*/cache/index*/

size

ways_of_associativity

number_of_sets

Windows:

GetLogicalProcessorInformationEx

X86 in general:

cpuid instruction

Suggested Reading

- “What Every Programmer Should Know About Memory”
- http://www.akkadia.org/drepper/cpumemory.pdf

Simulating Caches

Systems@ETH zürich

Insominac Games: Cachesim
https://github.com/InsomniacGames/ig-cachesim

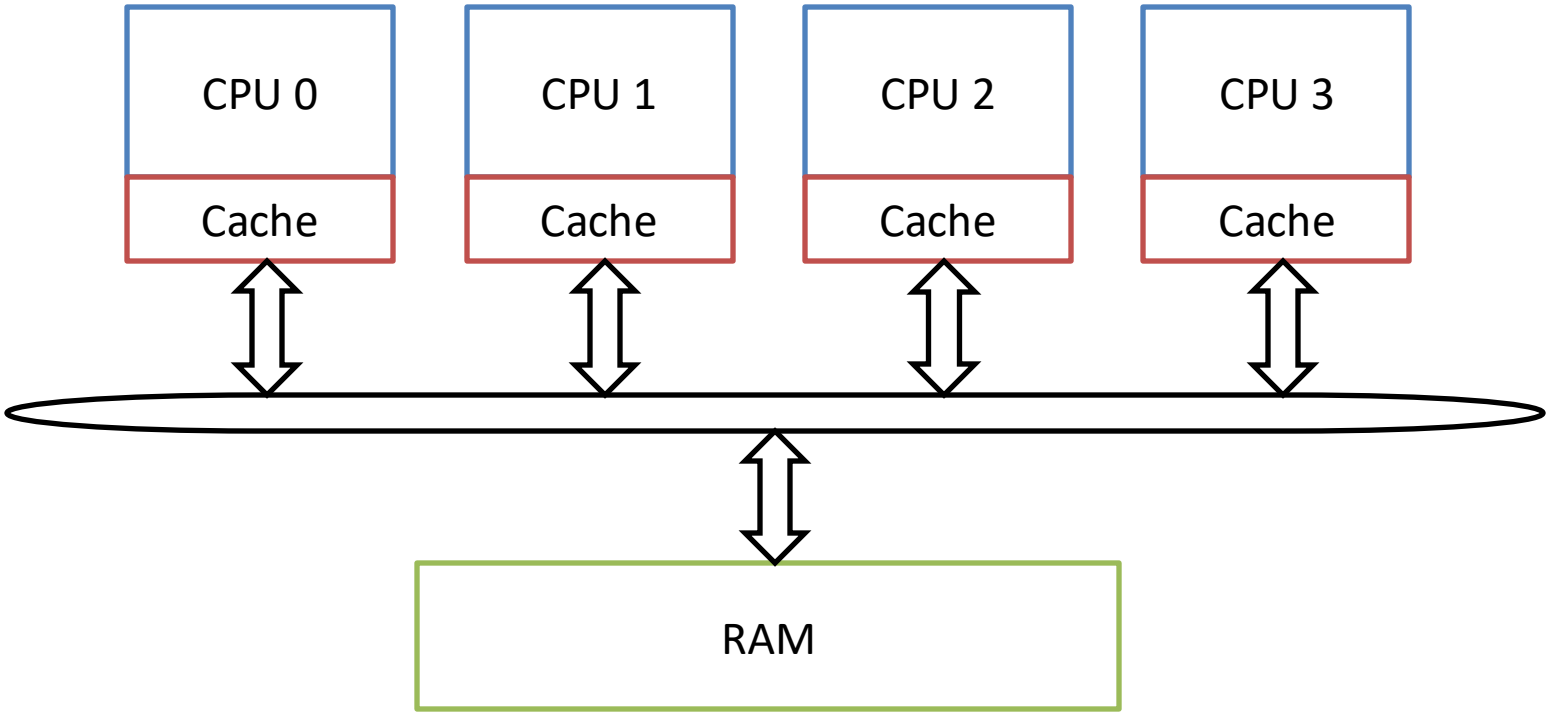
Cache Grind:
https://valgrind.org/docs/manual/cg-manual.html

Overview

Systems@ETH zürich

- Part of Assignment 10
- Recap Cache Coherence
- Hints for Assignment 11
- Quiz

Symmetric multiprocessing (SMP)



SMP only works because of caches!

- Shared memory rapidly becomes bottleneck

Coherency and Consistency

Systems@ETH zürich

- As with DMA, memory can change under a cache
  - Writes from other processors to memory
  - Leads to 2 important concepts:

1. **Coherency**:
   - Values in caches all match each other
   - Processors all see a coherent view of memory
2. **Consistency**:
   - The order in which changes to memory are seen by different processors

Sequential Consistency with a snoopy cache

Systems@ETH zürich

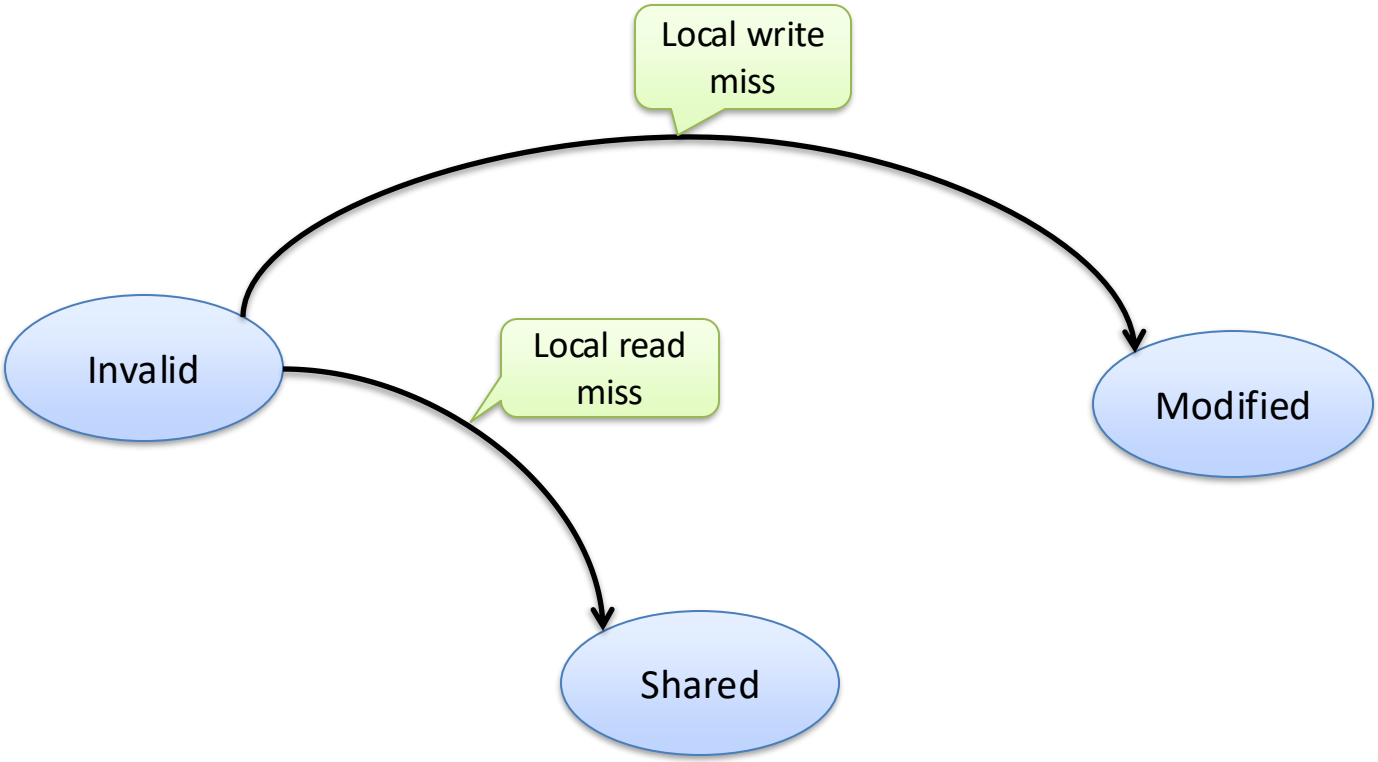
- Cache “snoops” on reads/writes from other processors
- If a line is valid in local cache:
  - Remote (other processor) write to line
    $\Rightarrow$ invalidate local line
- Requires a write-through cache!
  - But coherency mechanism $\Rightarrow$ sequential consistency
- Line can be valid in many caches, until a write

What about write-back caches?

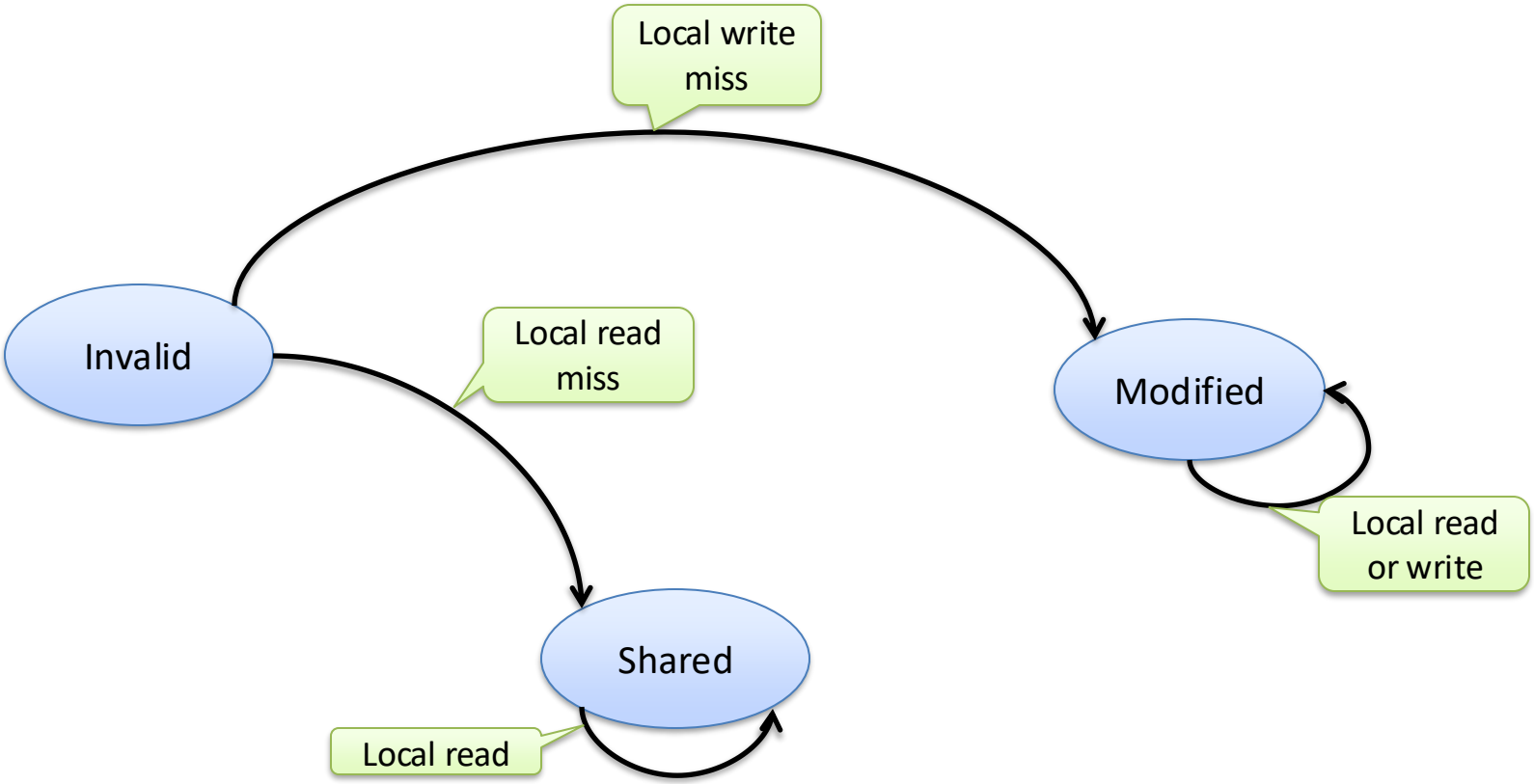
Systems@ETH zürich

- Cache lines can now be “dirty” (modified)
- Requires a **cache coherency protocol**
- Simplest protocol: MSI
  - Each line has 3 states: Modified, Shared, Invalid
  - Line can only be dirty in one cache
- Cache logic must respond to:
  - Processor reads and writes
  - Remote bus reads and writes
- and must:
  - Change cache line state
  - Write back data (flush) if required

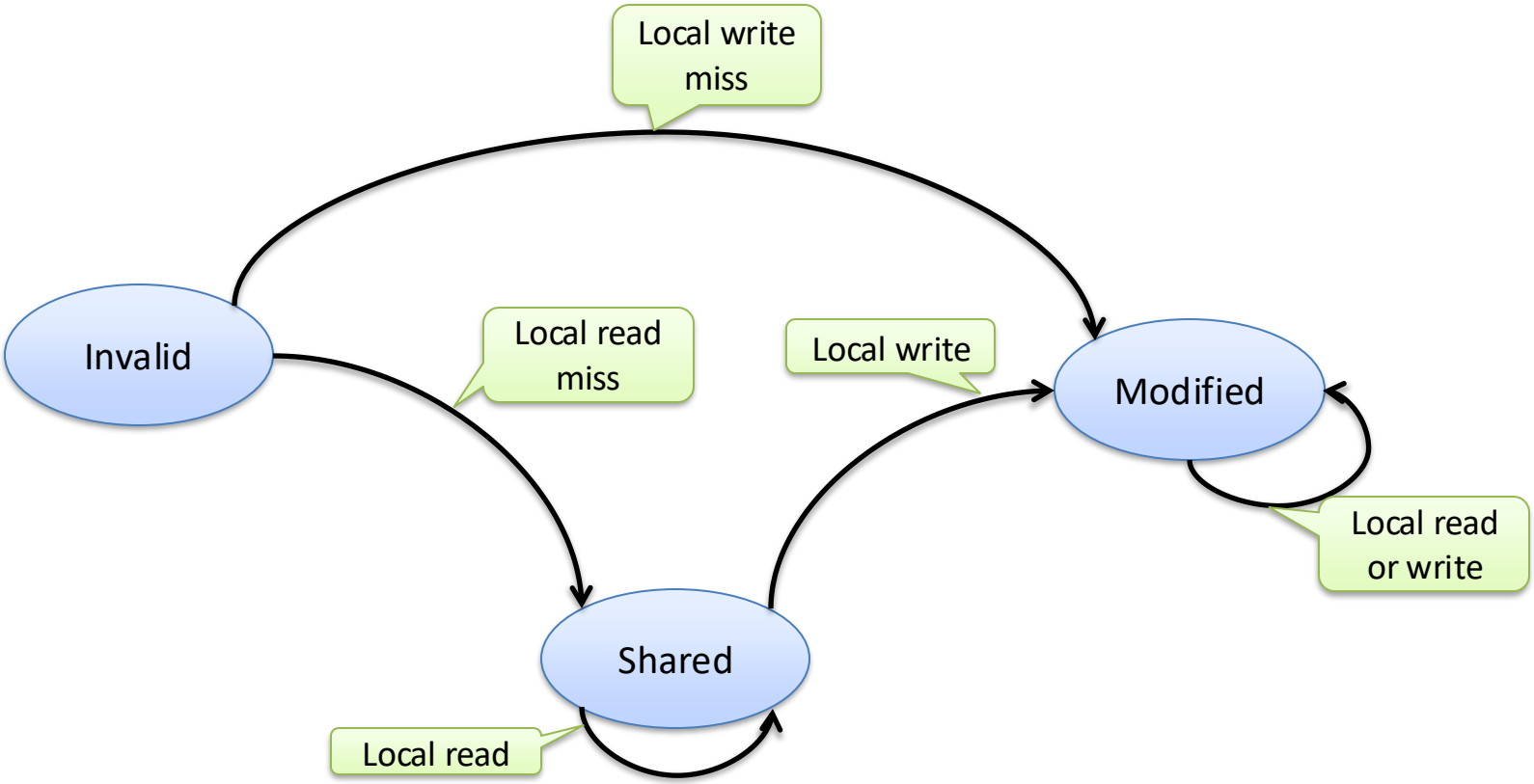
MSI state machine: local (processor) transitions



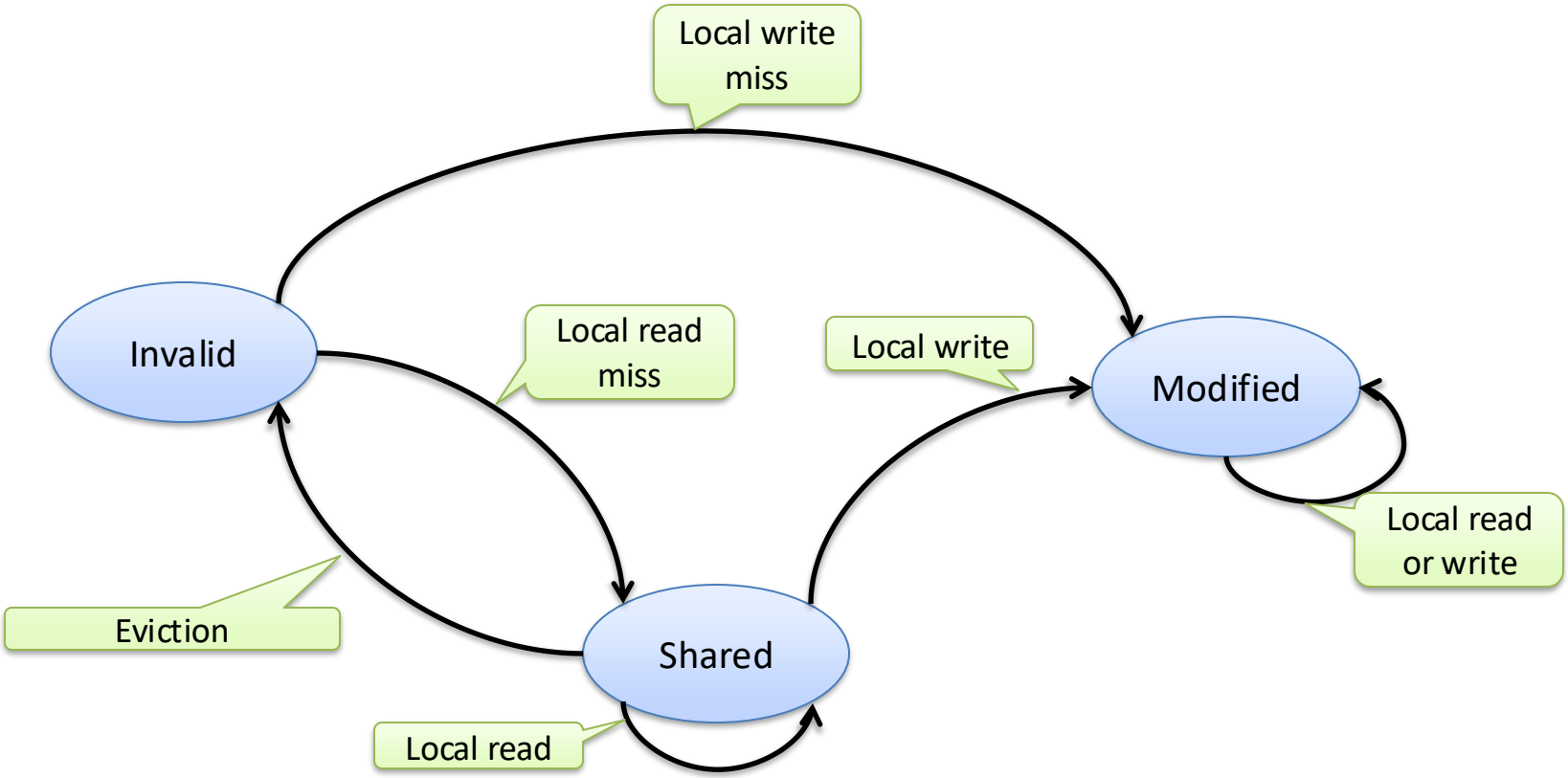
MSI state machine: local (processor) transitions



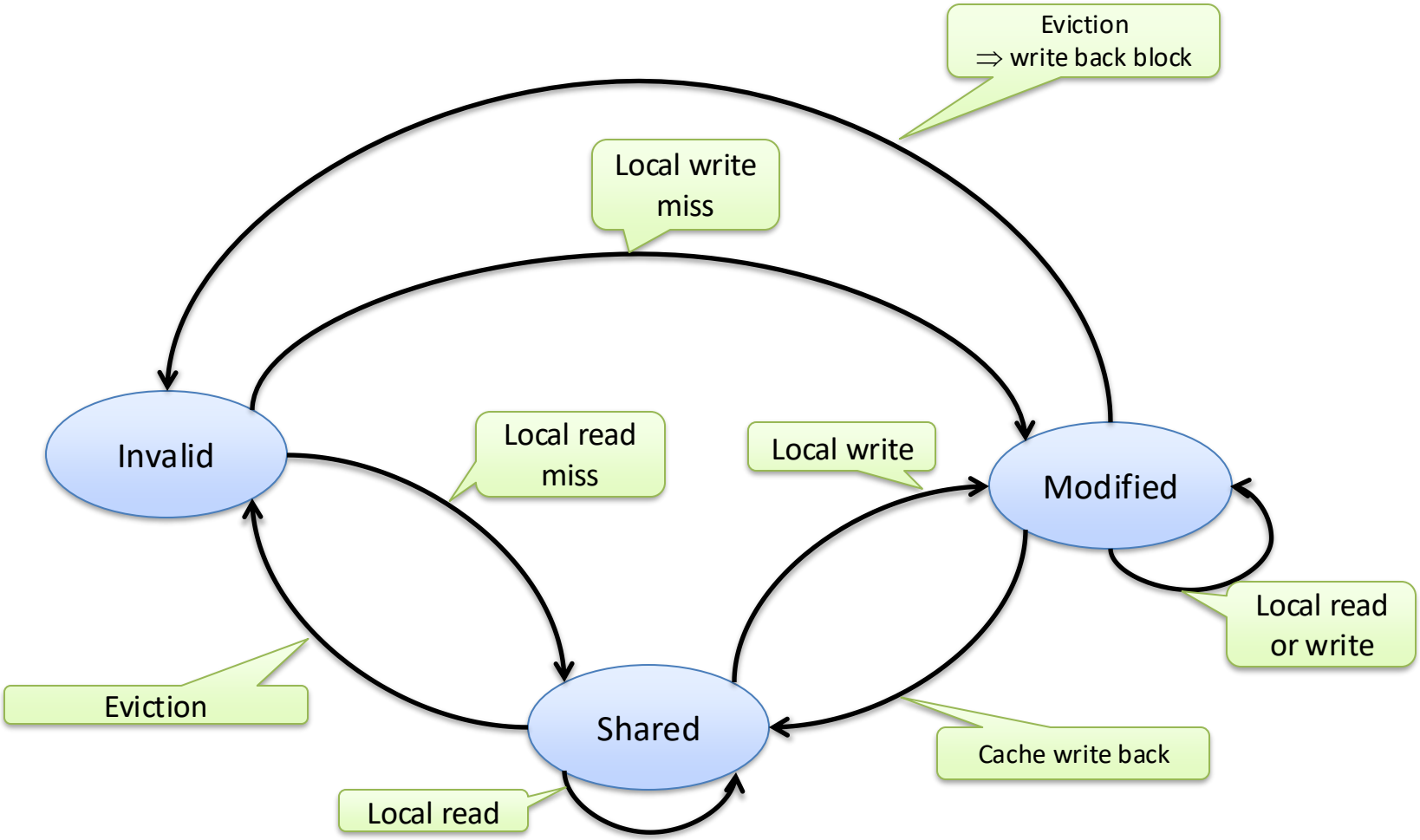
MSI state machine: local (processor) transitions



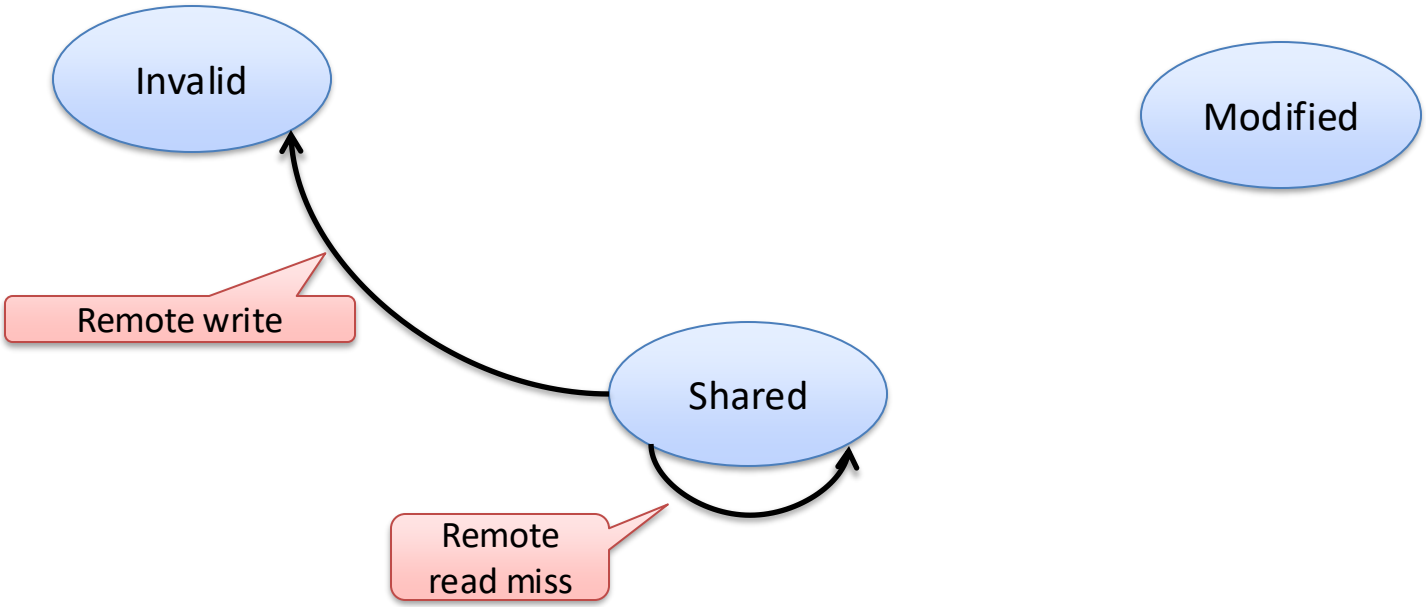
MSI state machine: local (processor) transitions



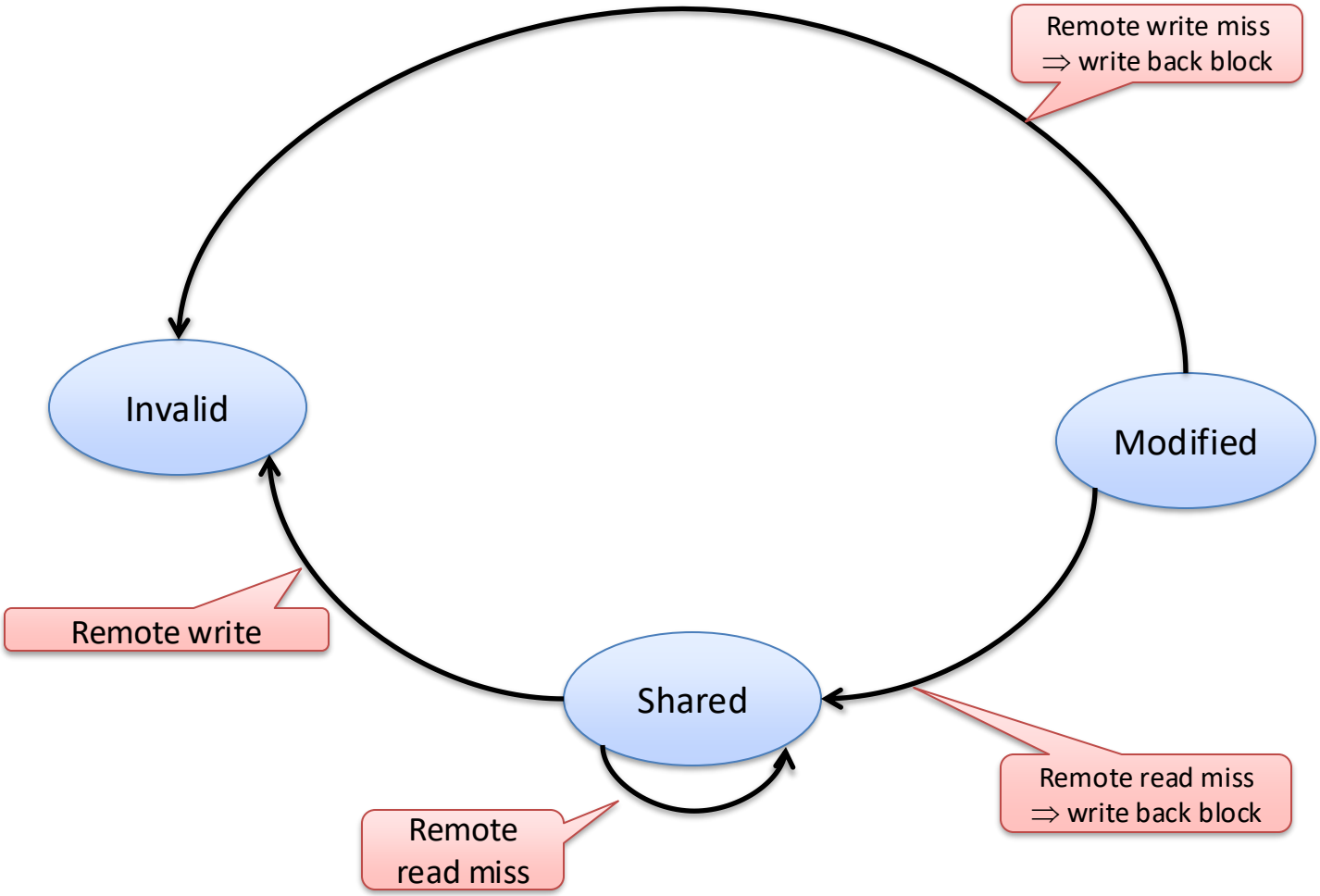
MSI state machine: local (processor) transitions



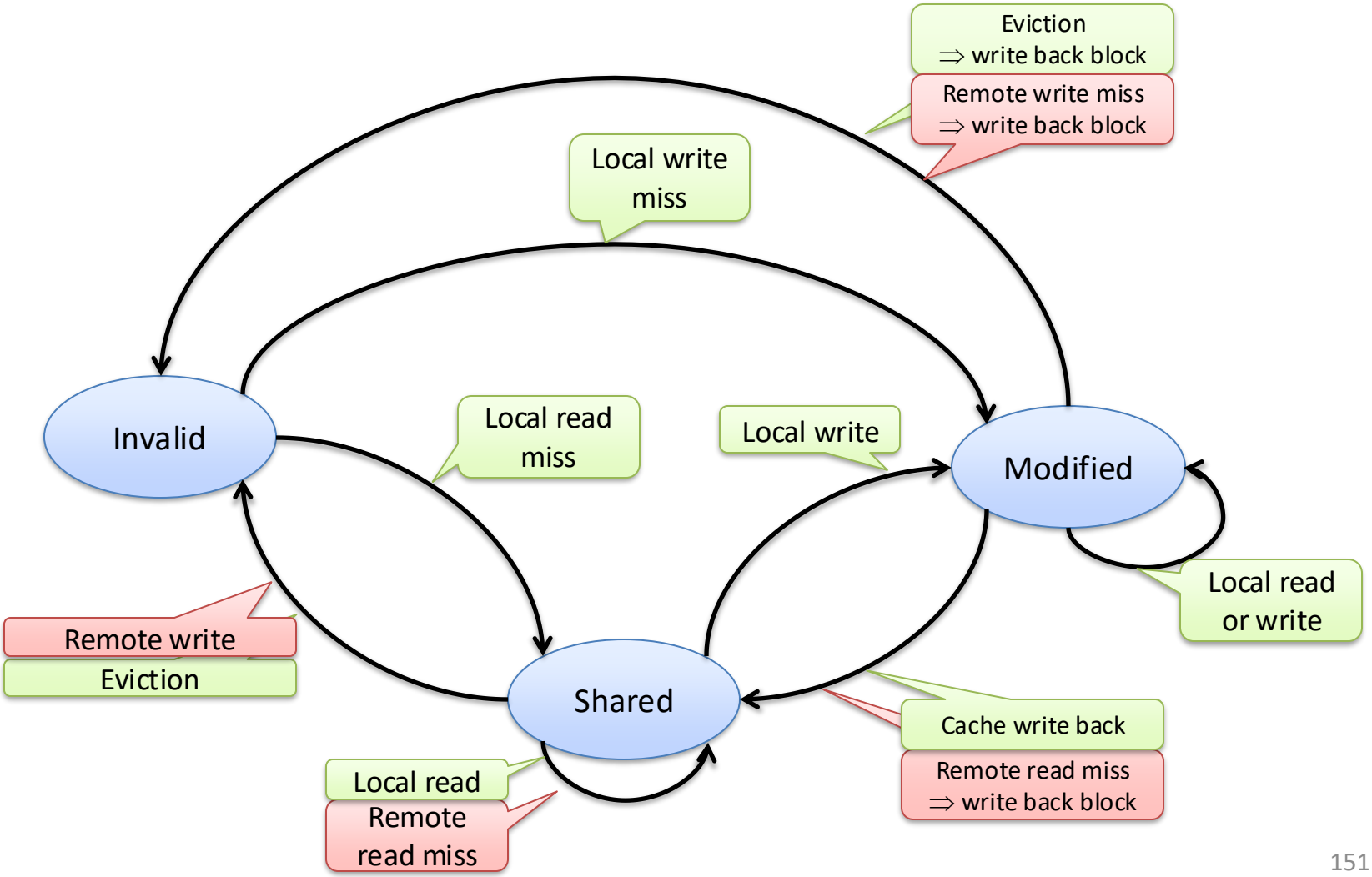
MSI state machine: remote (snooped) transitions



MSI state machine: remote (snooped) transitions



MSI state machine: all transitions



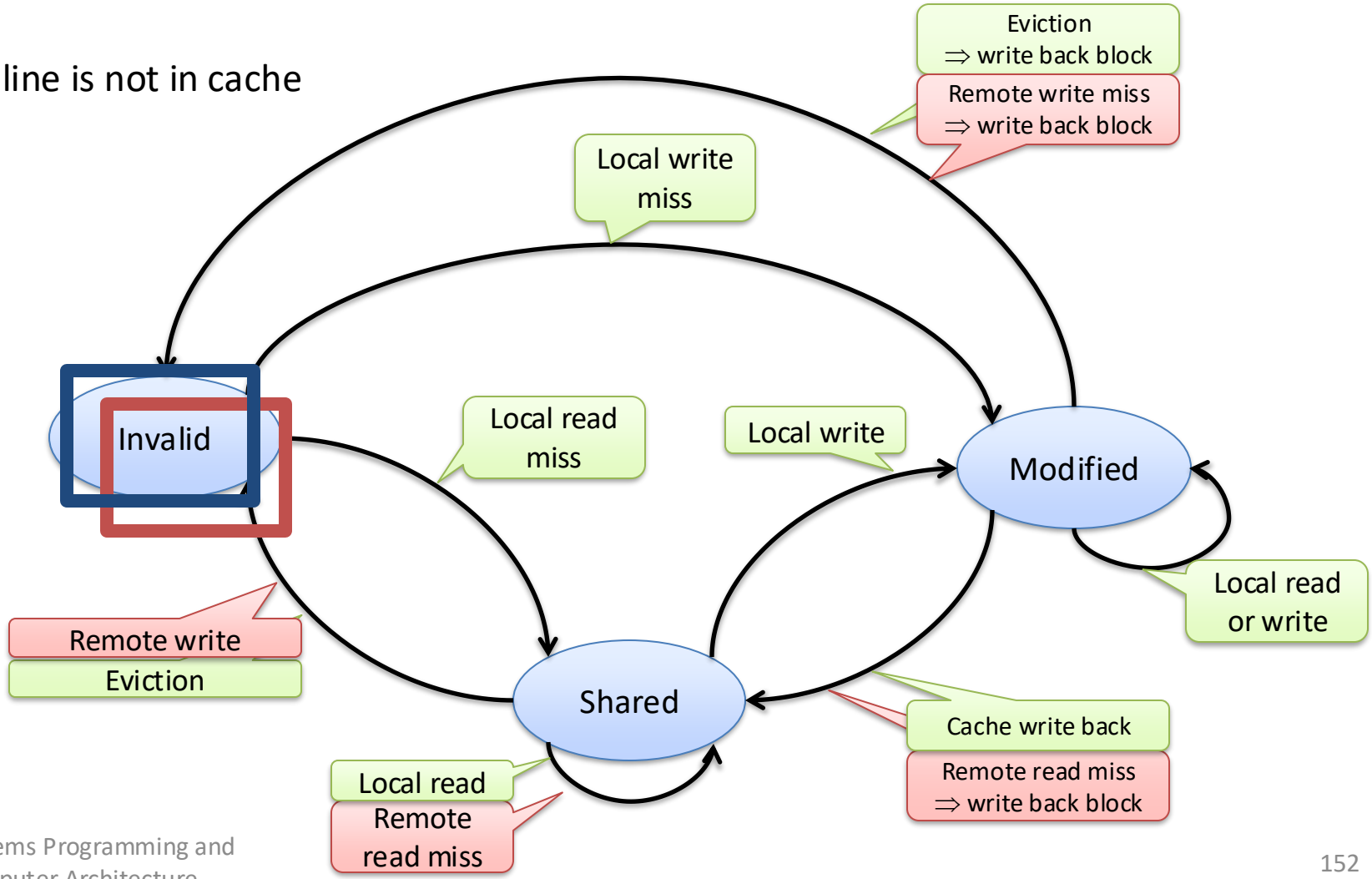
Core 0

Core 1

MSI Model

Systems@ETH zürich

Cache line is not in cache



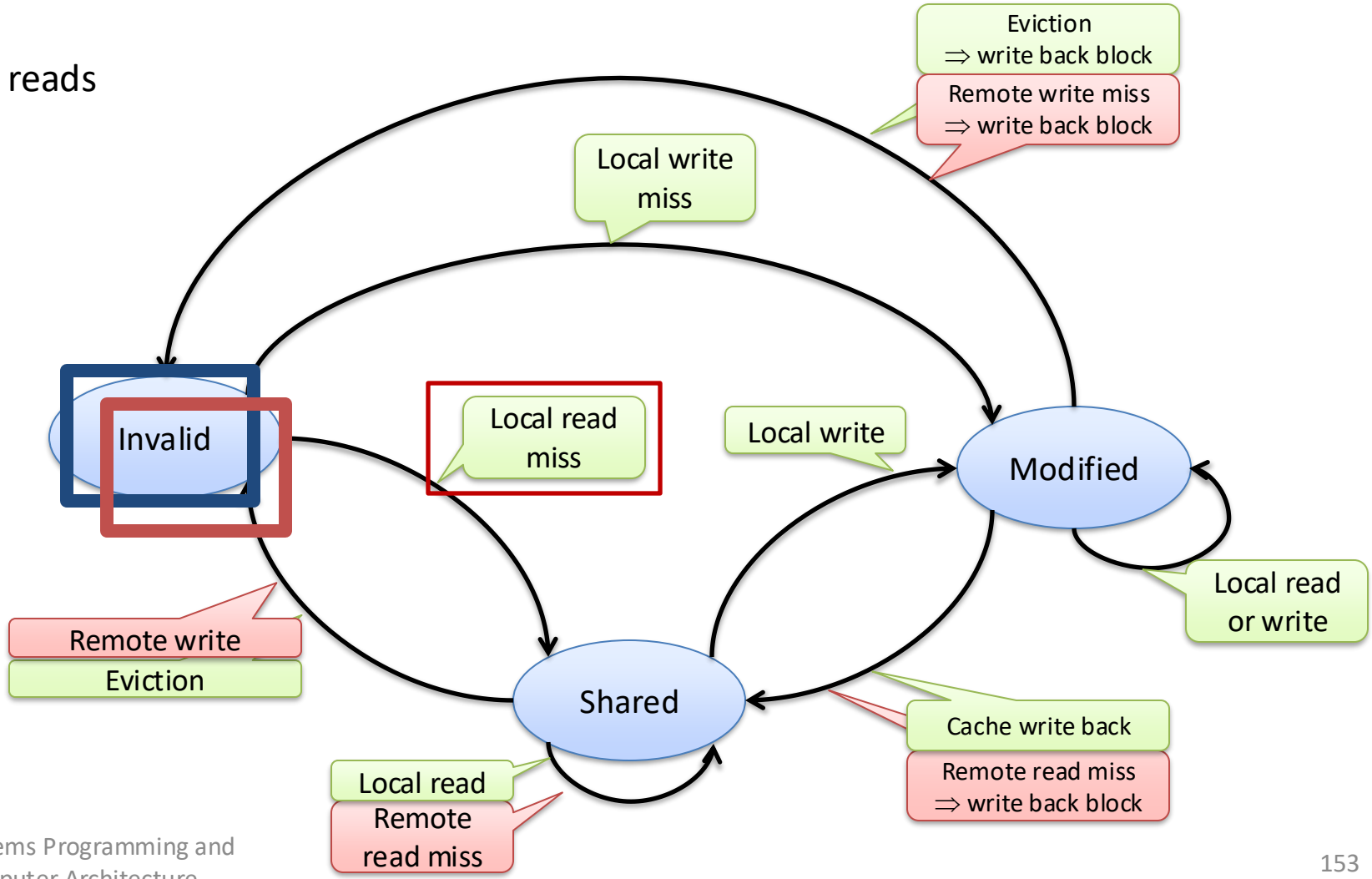
Core 0

Core 1

MSI Model

Systems@ETH zürich

Core 0 reads



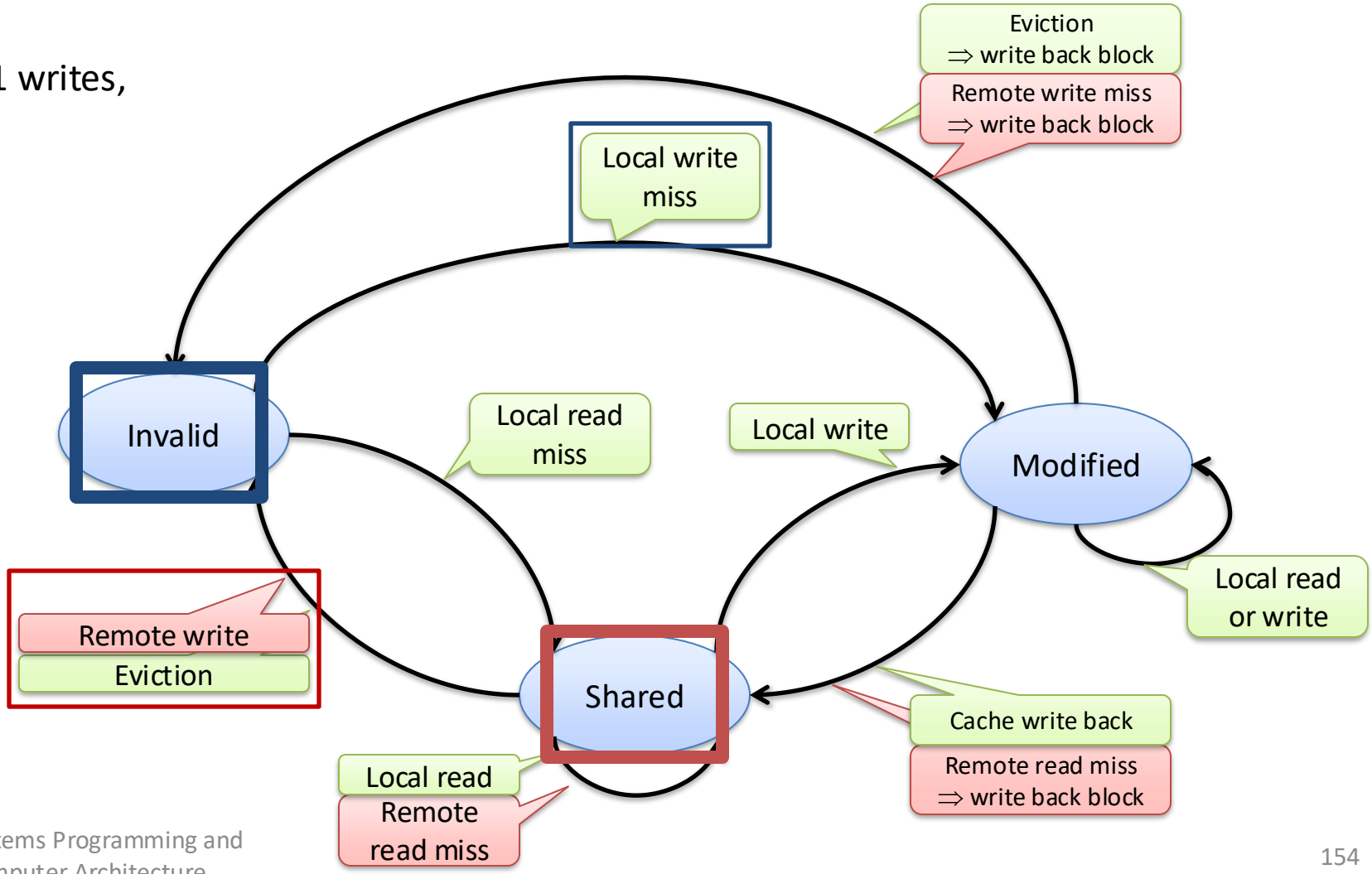
Core 0

Core 1

MSI Model

Systems@ETH zürich

Core 1 writes,



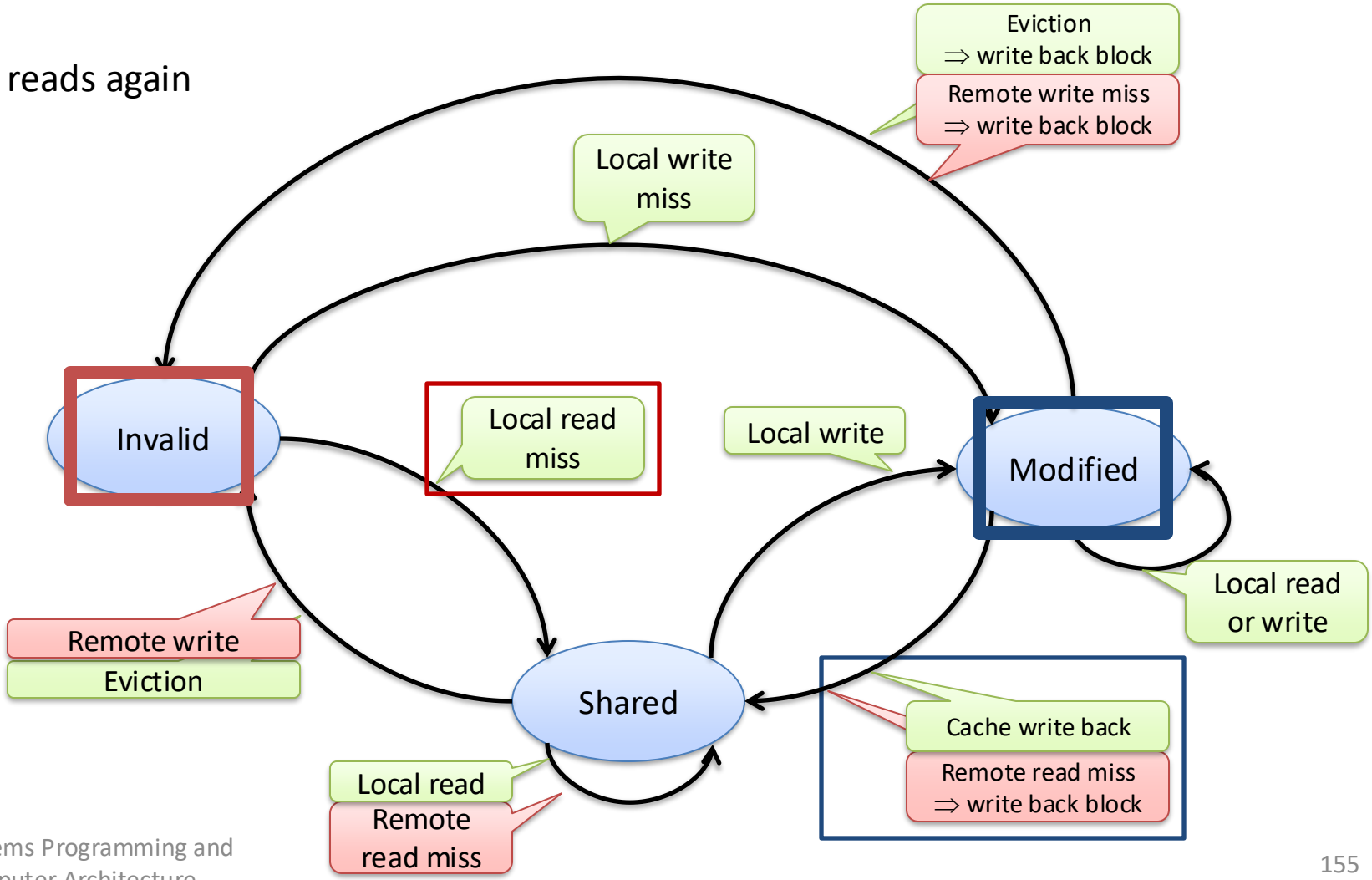
Core 0

Core 1

MSI Model

Systems@ETH zürich

Core 0 reads again



Core 0

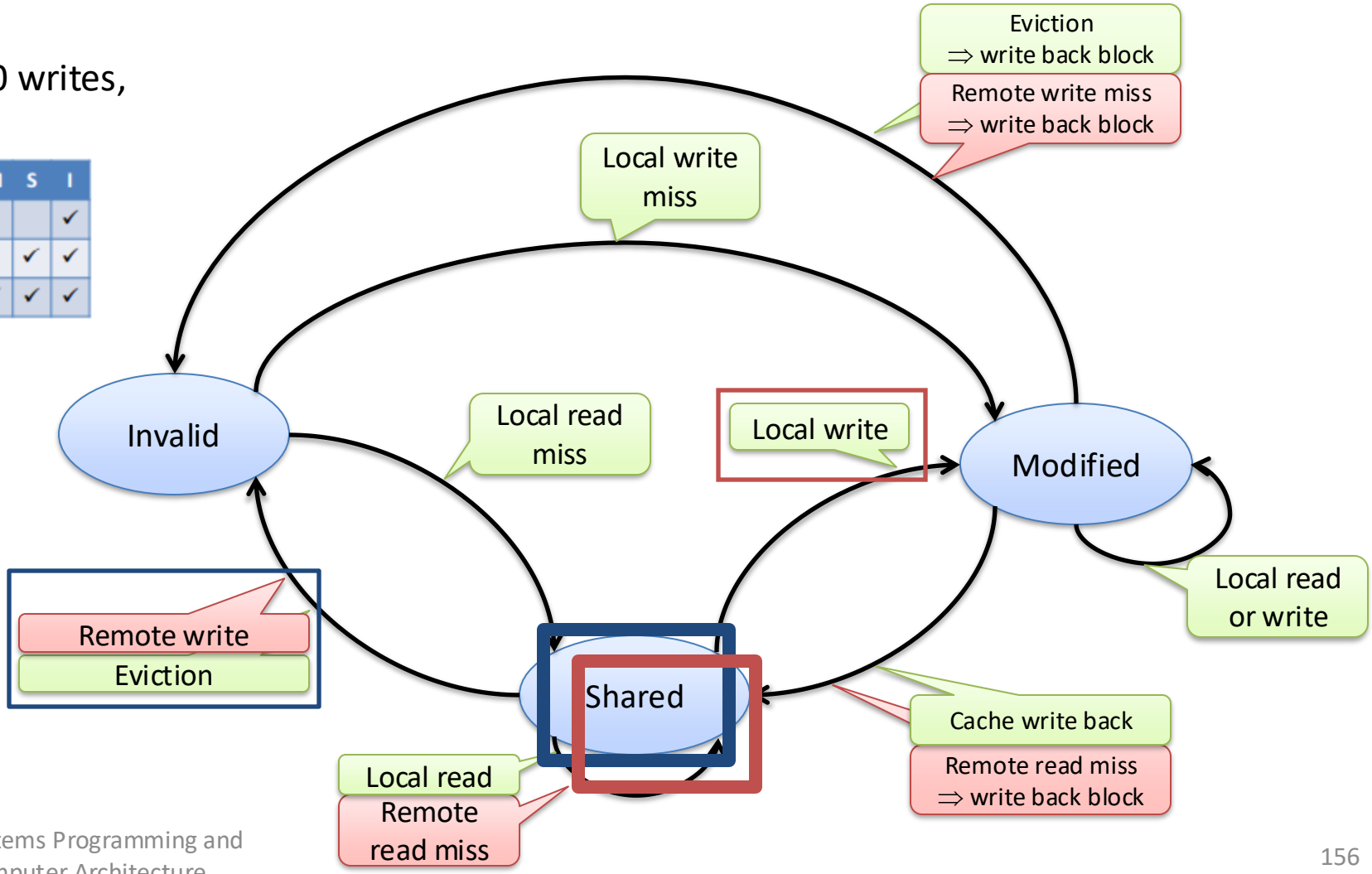
Core 1

MSI Model

Systems@ETH zürich

Core 0 writes,

	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓



Core 0

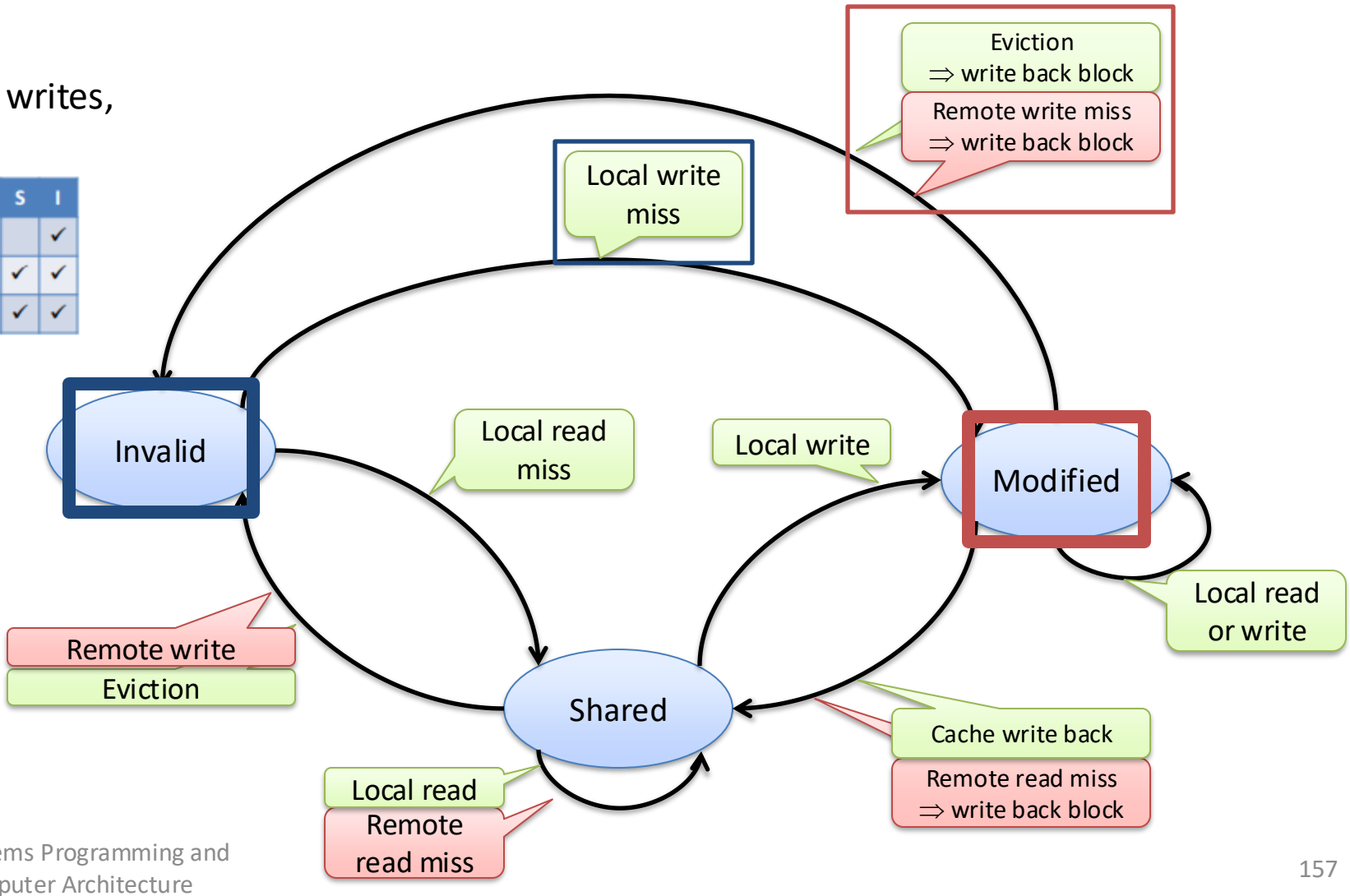
Core 1

MSI Model

Systems@ETH zürich

Core 1 writes,

	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓



MSI issues

- Assume: Read then Write
- On MSI: Invalid->Shared, Shared->Modified. ***Two*** bus transactions.

- Idea: Introduce exclusive state to perform the Shared->Modified transition without a bus transaction.

MESI protocol

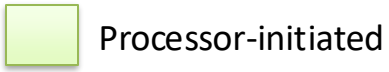
Systems@ETH zürich

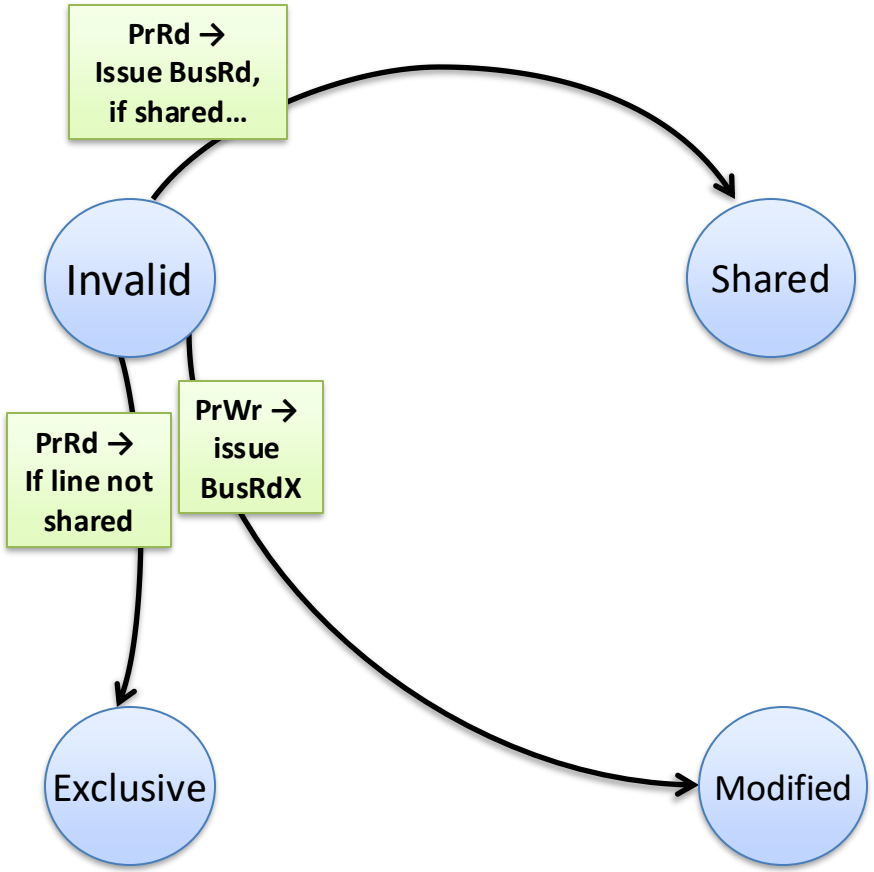
- Add a new line state: “exclusive”
  - **Modified**: This is the only copy, it’s dirty
  - **Exclusive**: This is the only copy, it’s clean
  - **Shared**: This is one of several copies, all clean
  - **Invalid**
- Add a new bus signal: **RdX**
  - “Read exclusive”
  - Cache can load into either “shared” or “exclusive” states
  - Other caches can see the type of read
- Also: **HIT** signal
  - Signals to a remote processor that its read hit in local cache.
- First x86 appearance in the Pentium

MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

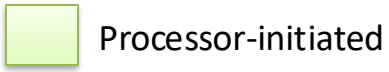


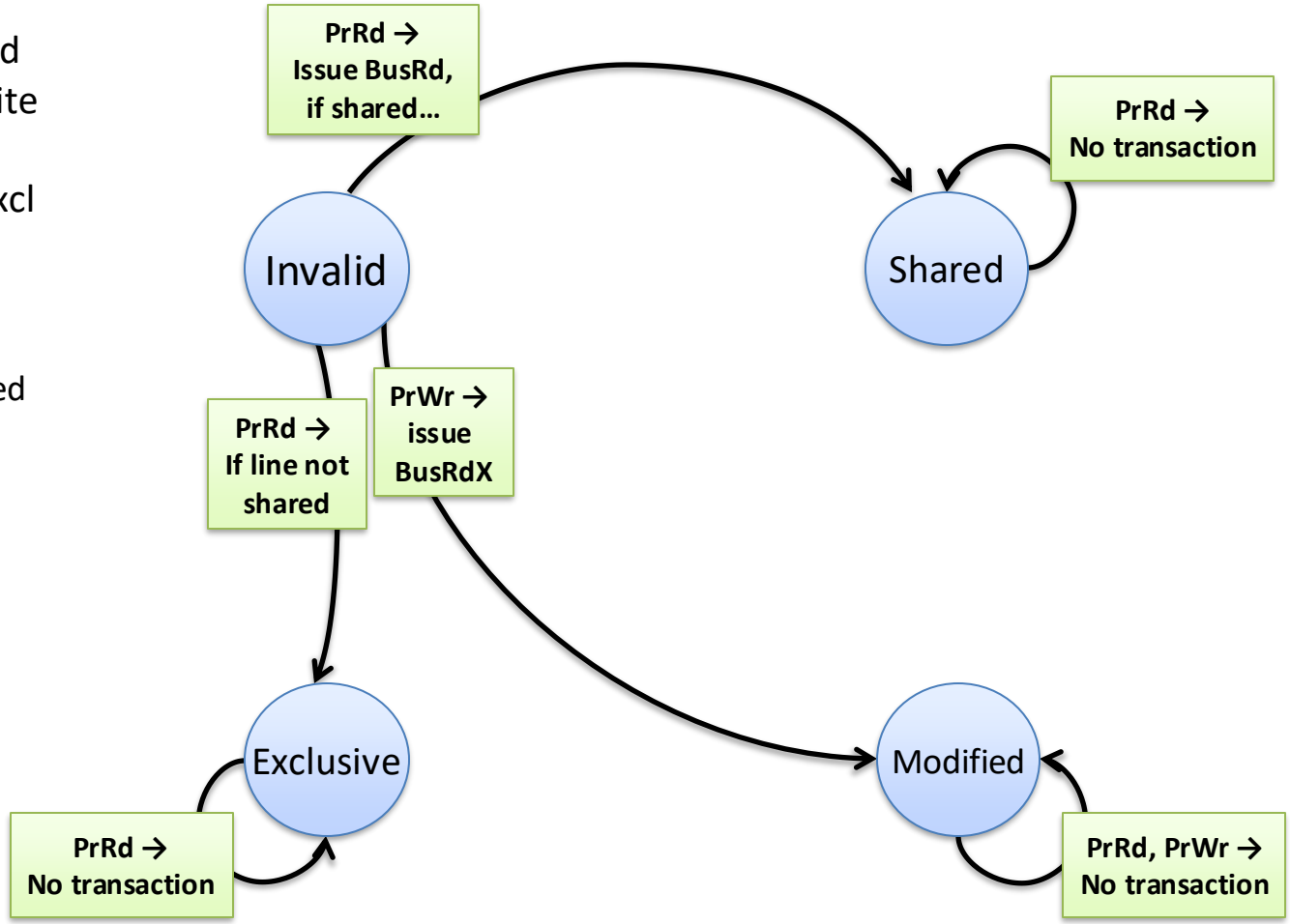


MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



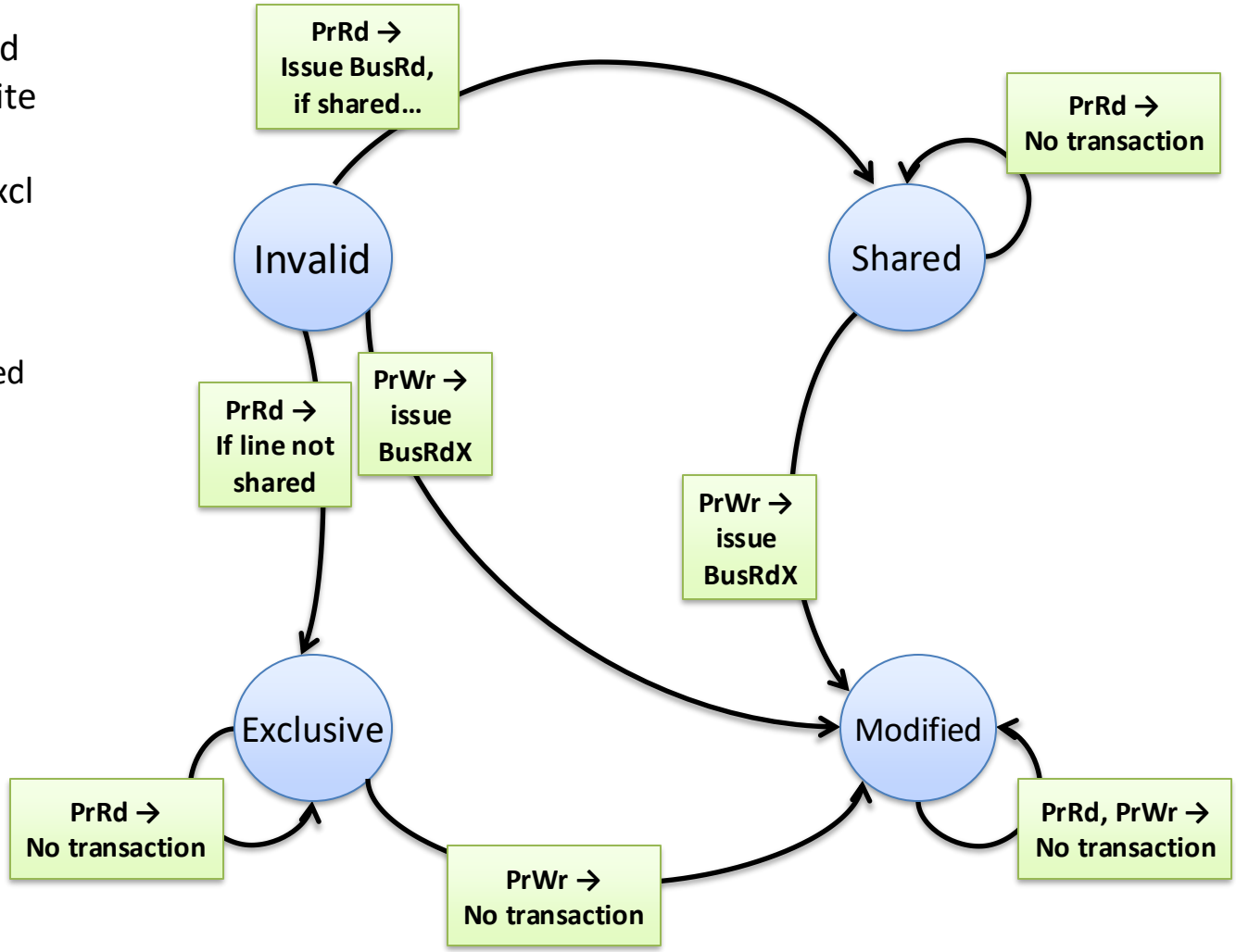


MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

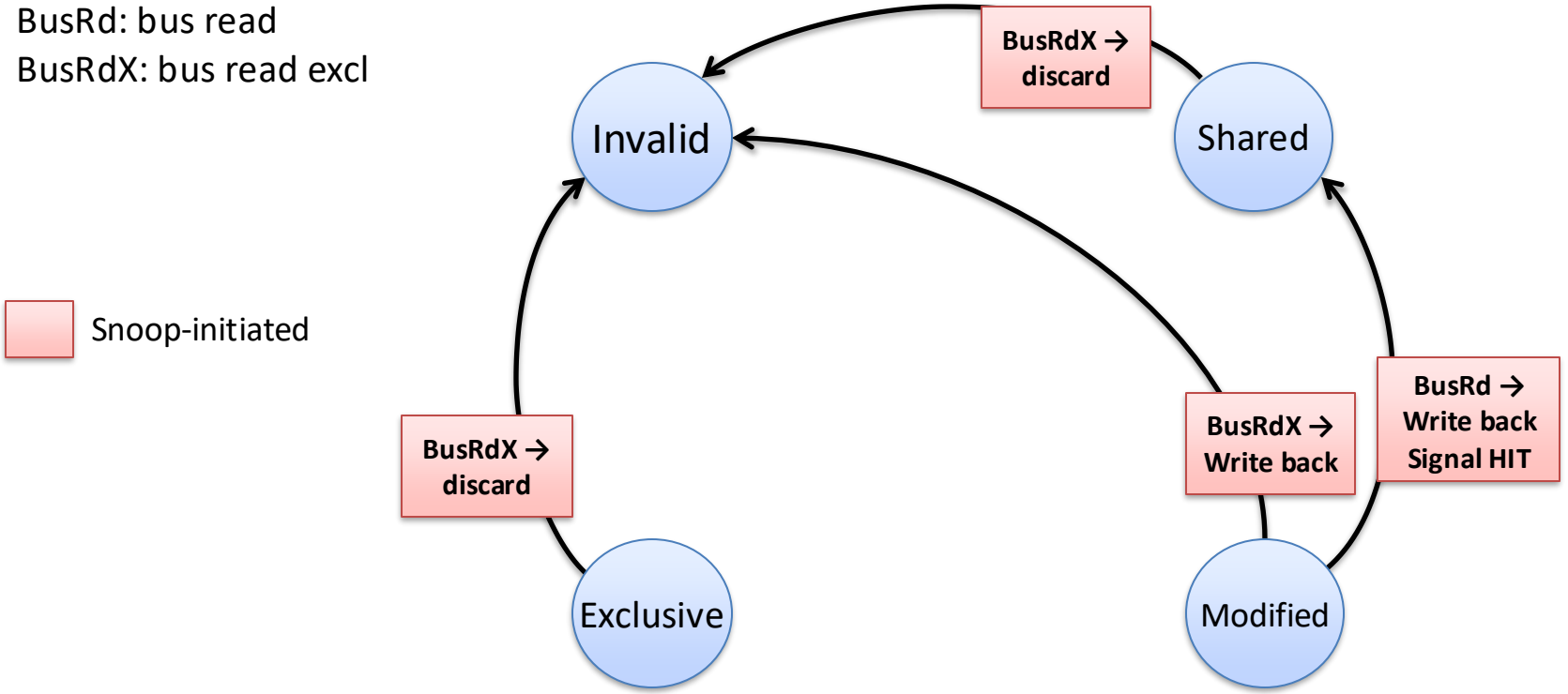
Processor-initiated



MESI state machine

Terminology:

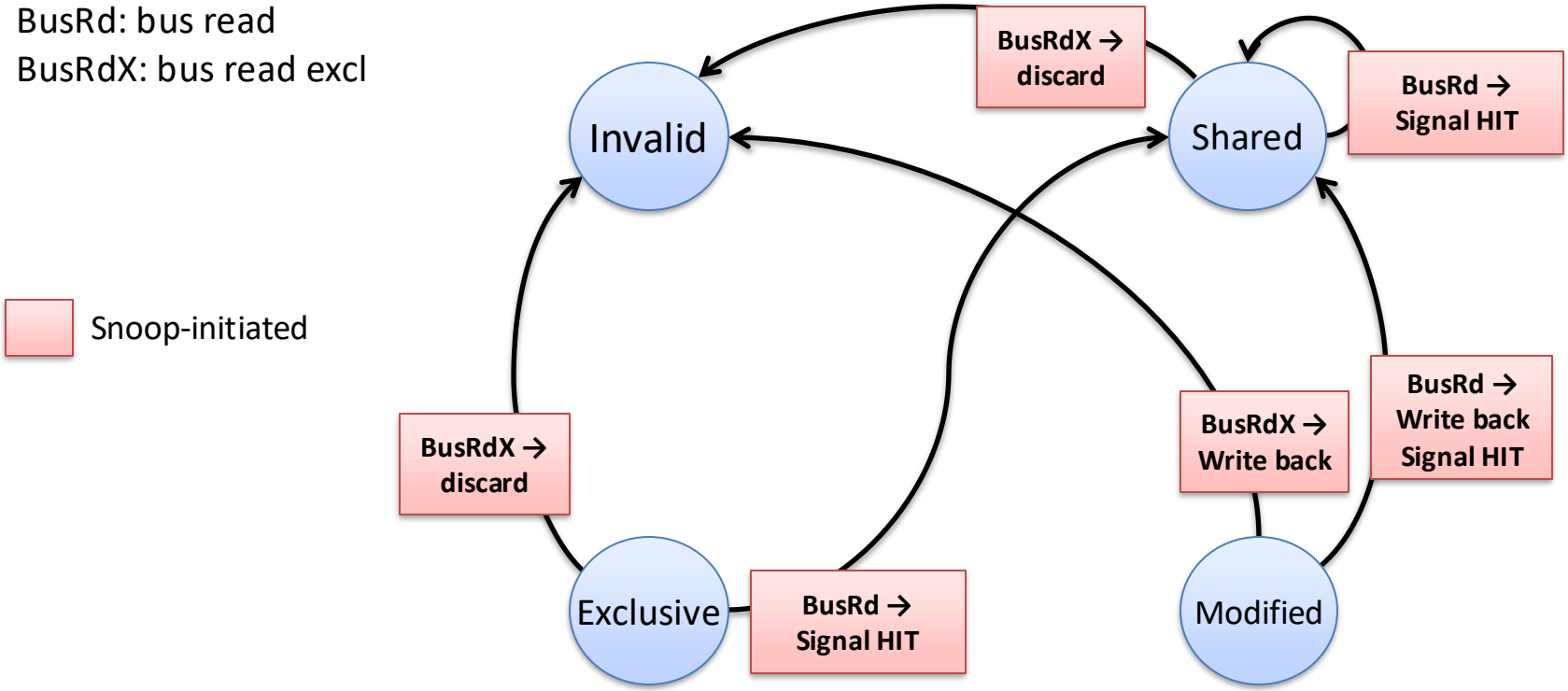
- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



MESI state machine

Terminology:

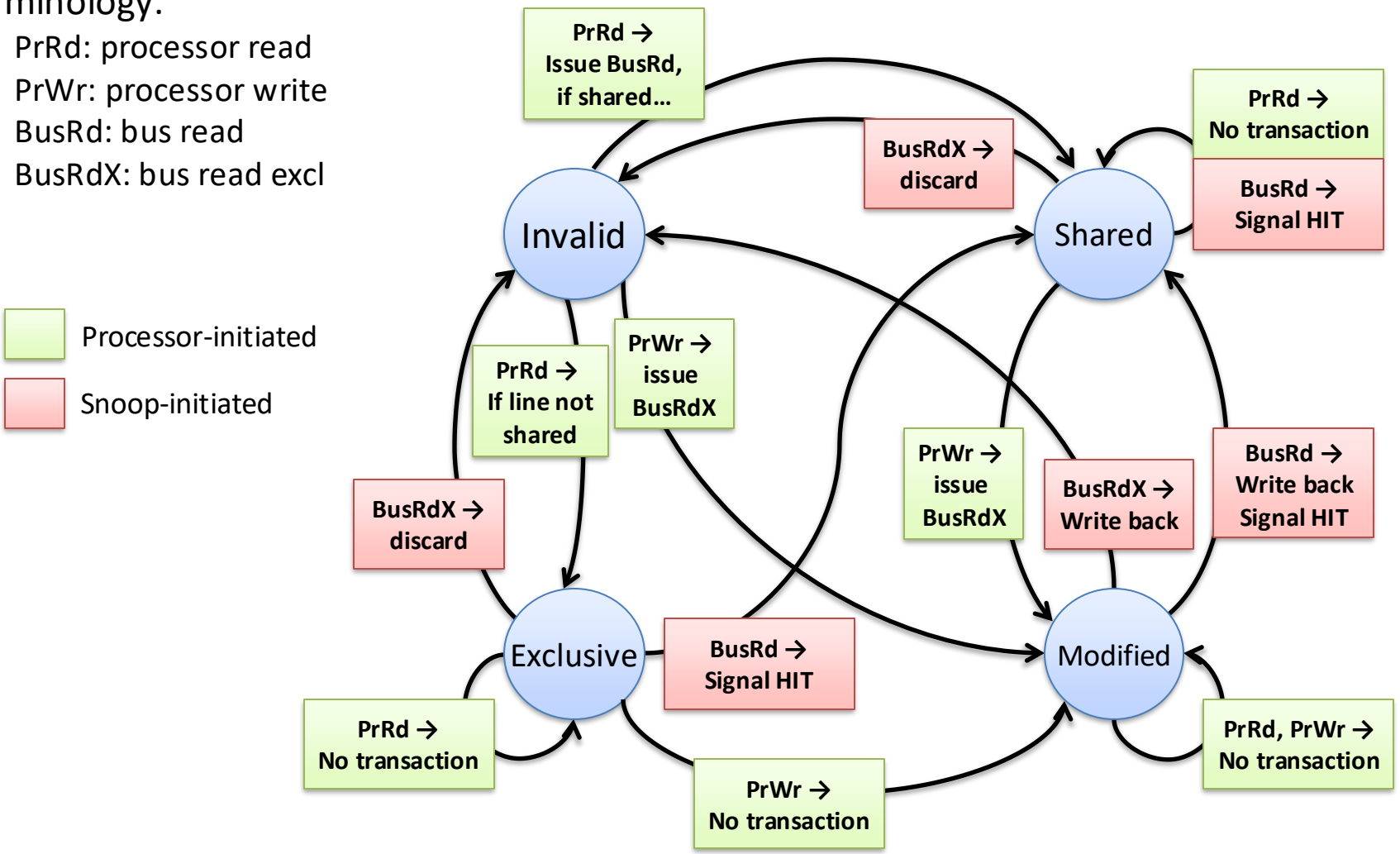
- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



MESI state machine

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



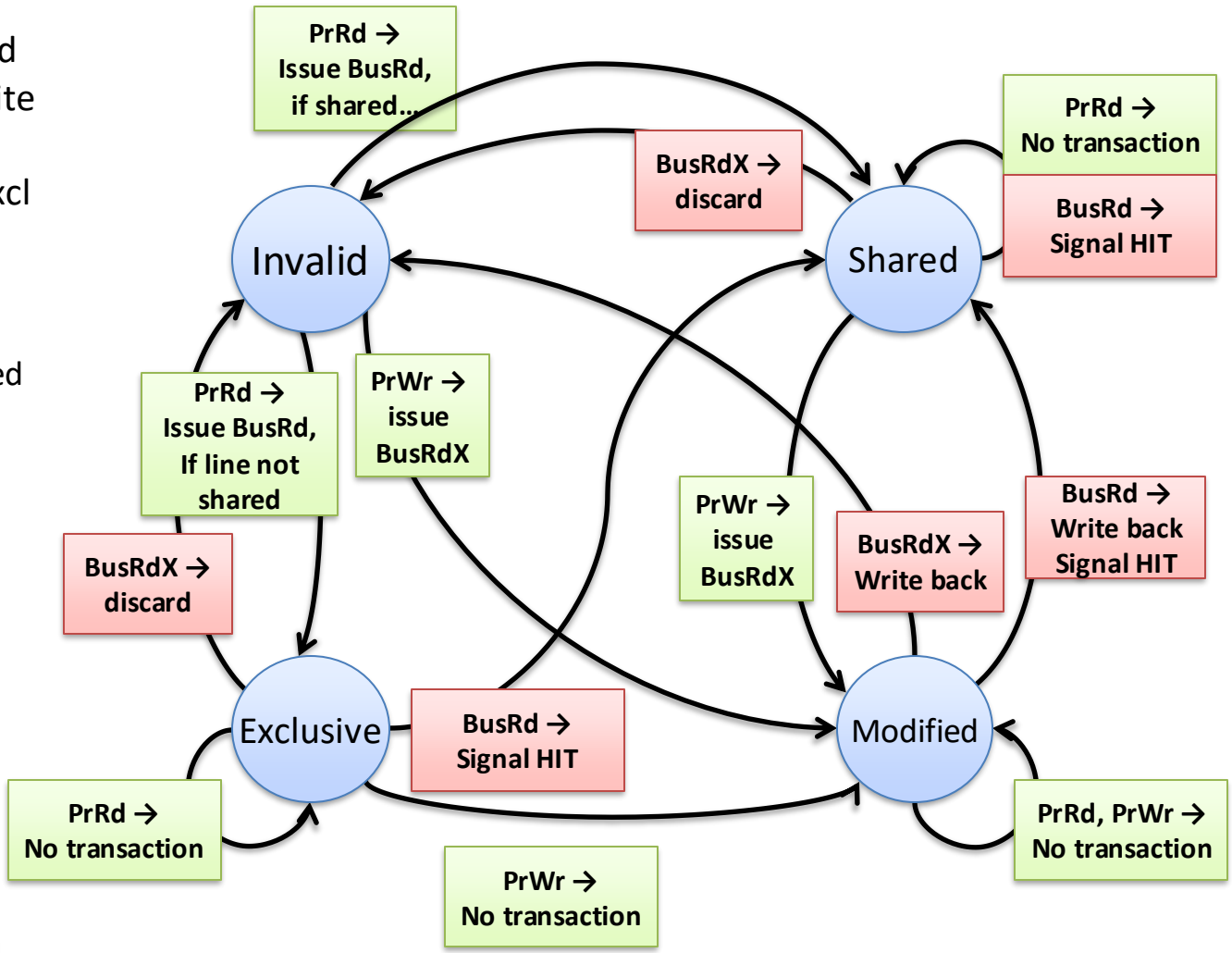
Core 0

Core 1

MESI

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



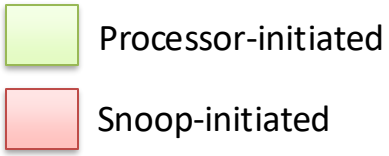
Core 0

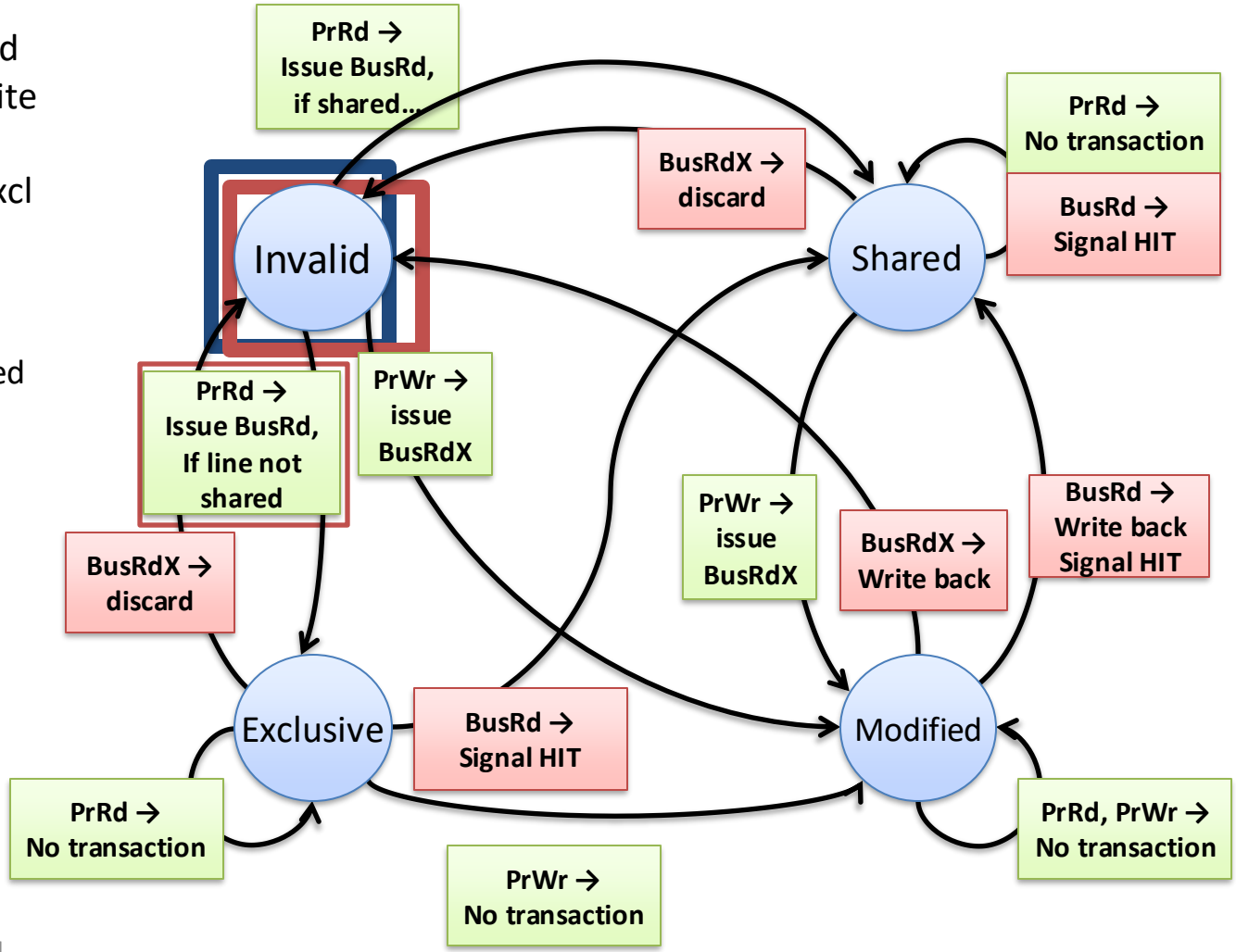
Core 1

MESI

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl





Core 0

Core 1

MESI

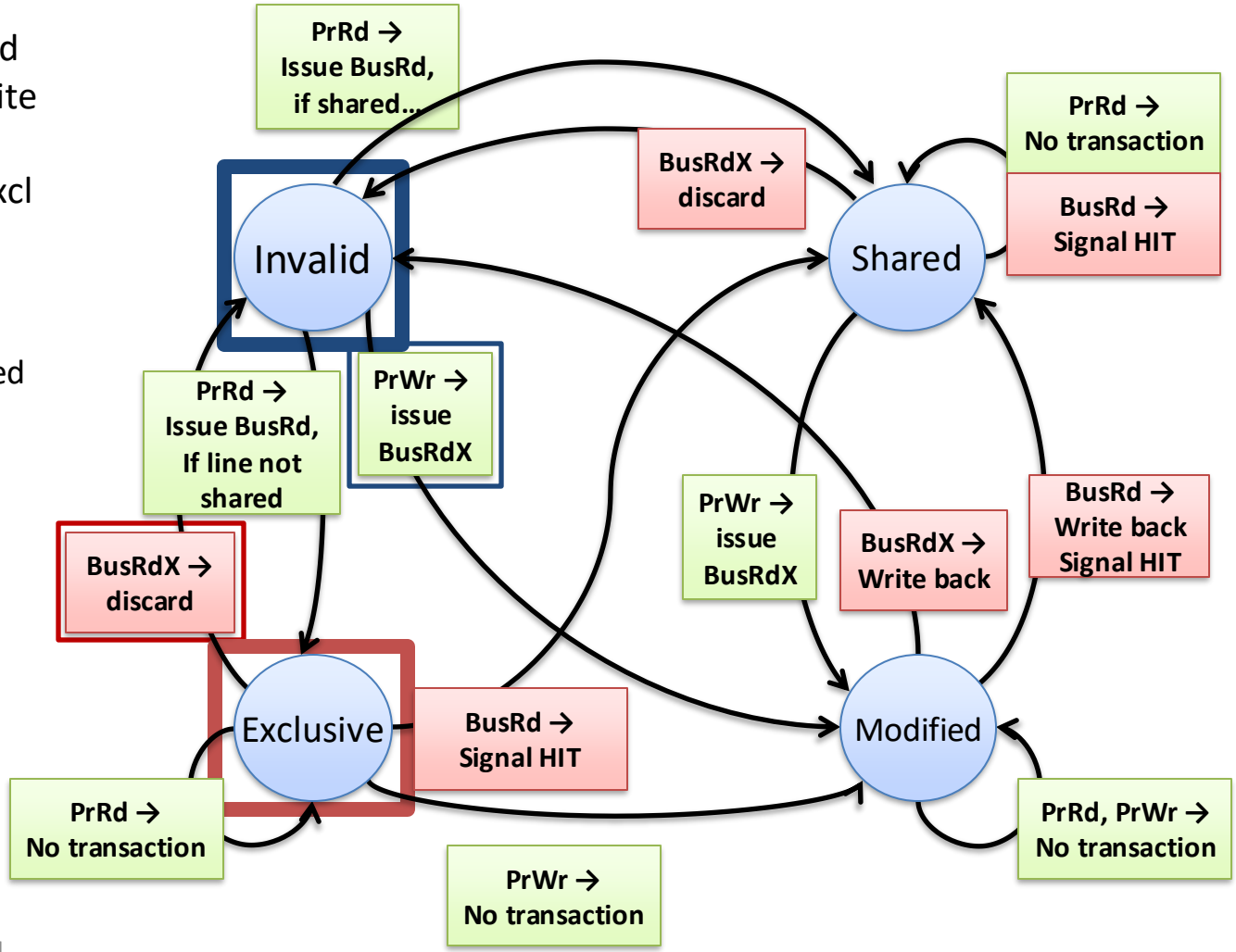
Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

Processor-initiated

Snoop-initiated

Core 1: write



Core 0

Core 1

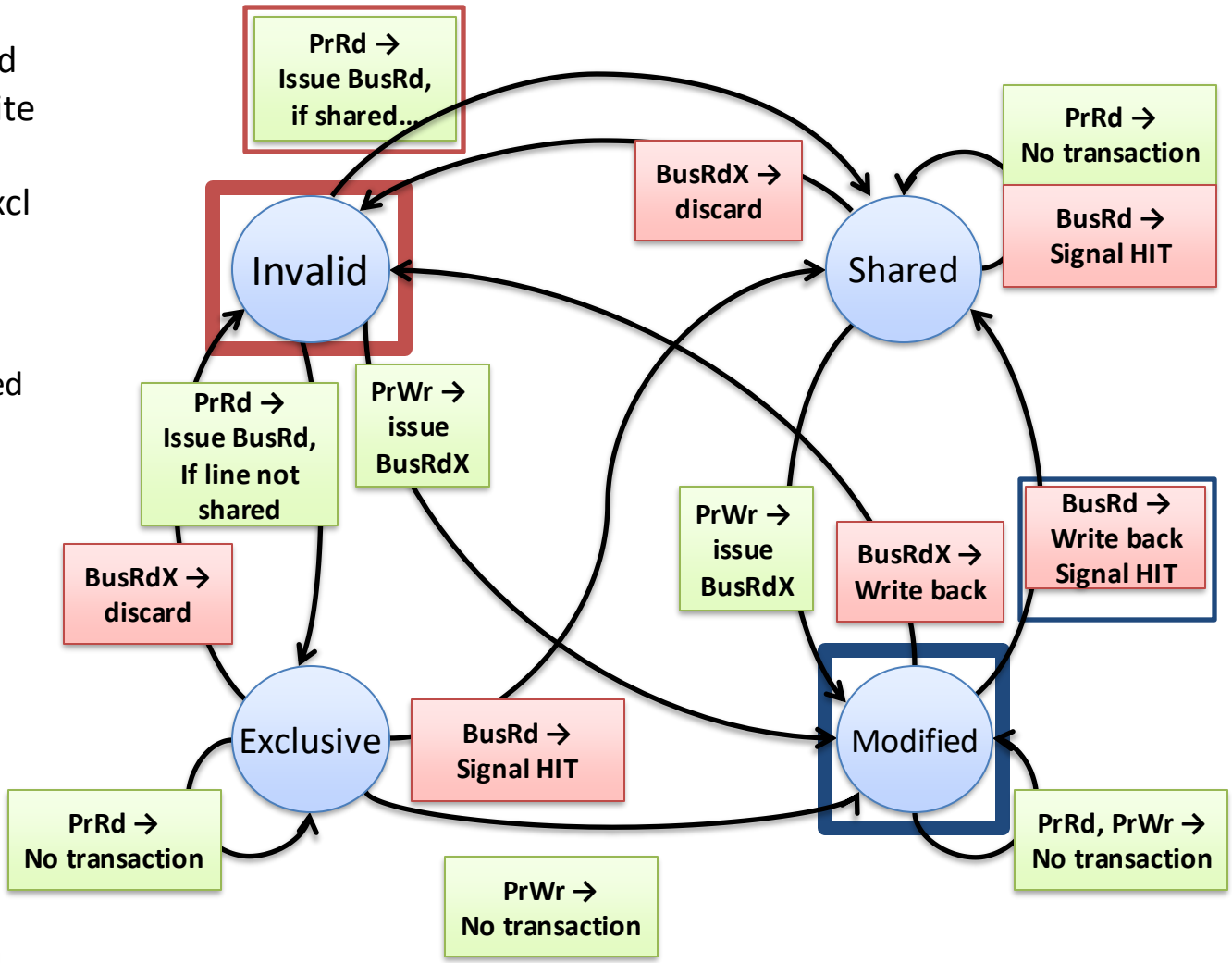
MESI

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

Processor-initiated

Snoop-initiated



Core 0: read

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

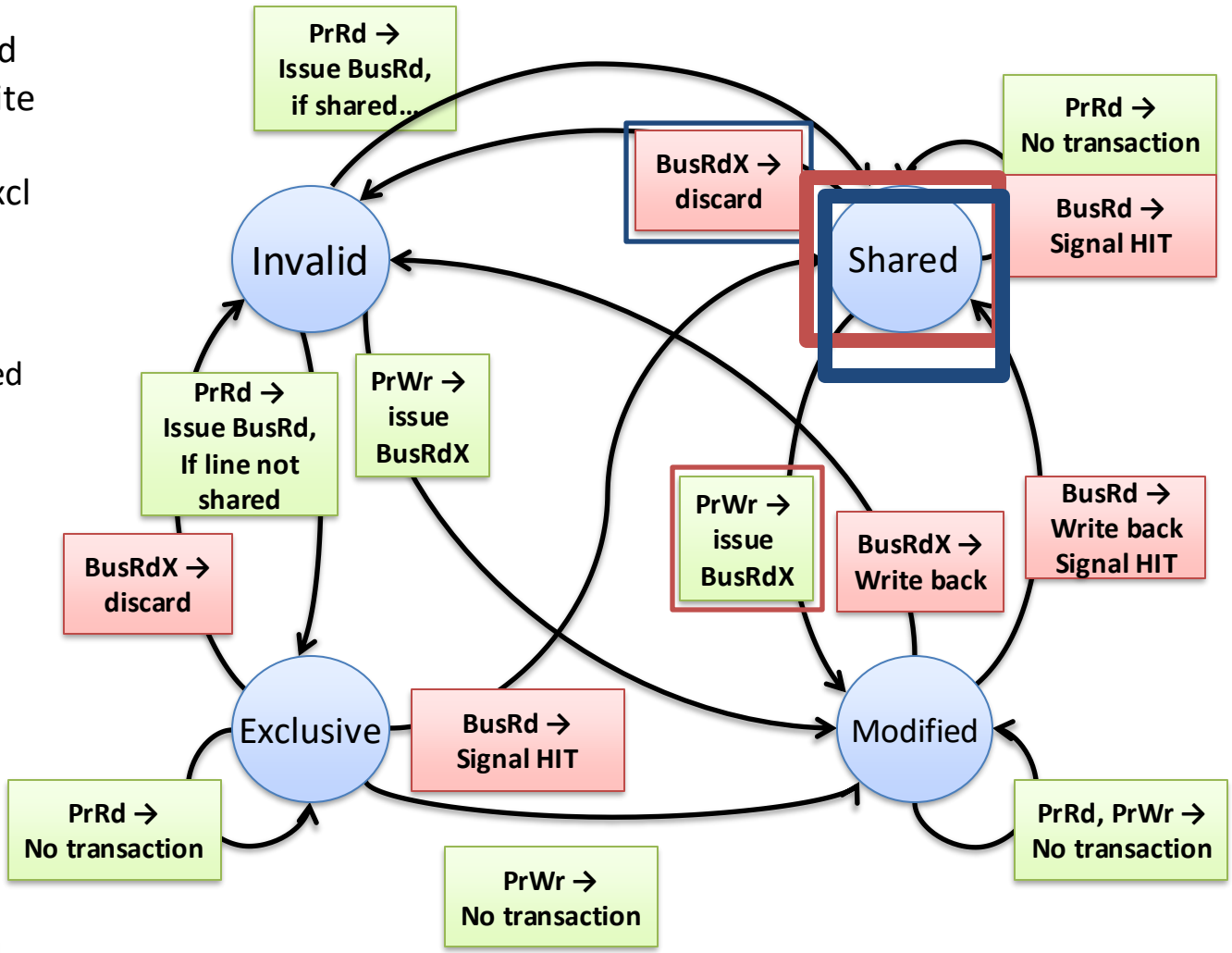
Core 0

Core 1

MESI

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl



Processor-initiated

Snoop-initiated

Core 0: write

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

Core 0

Core 1

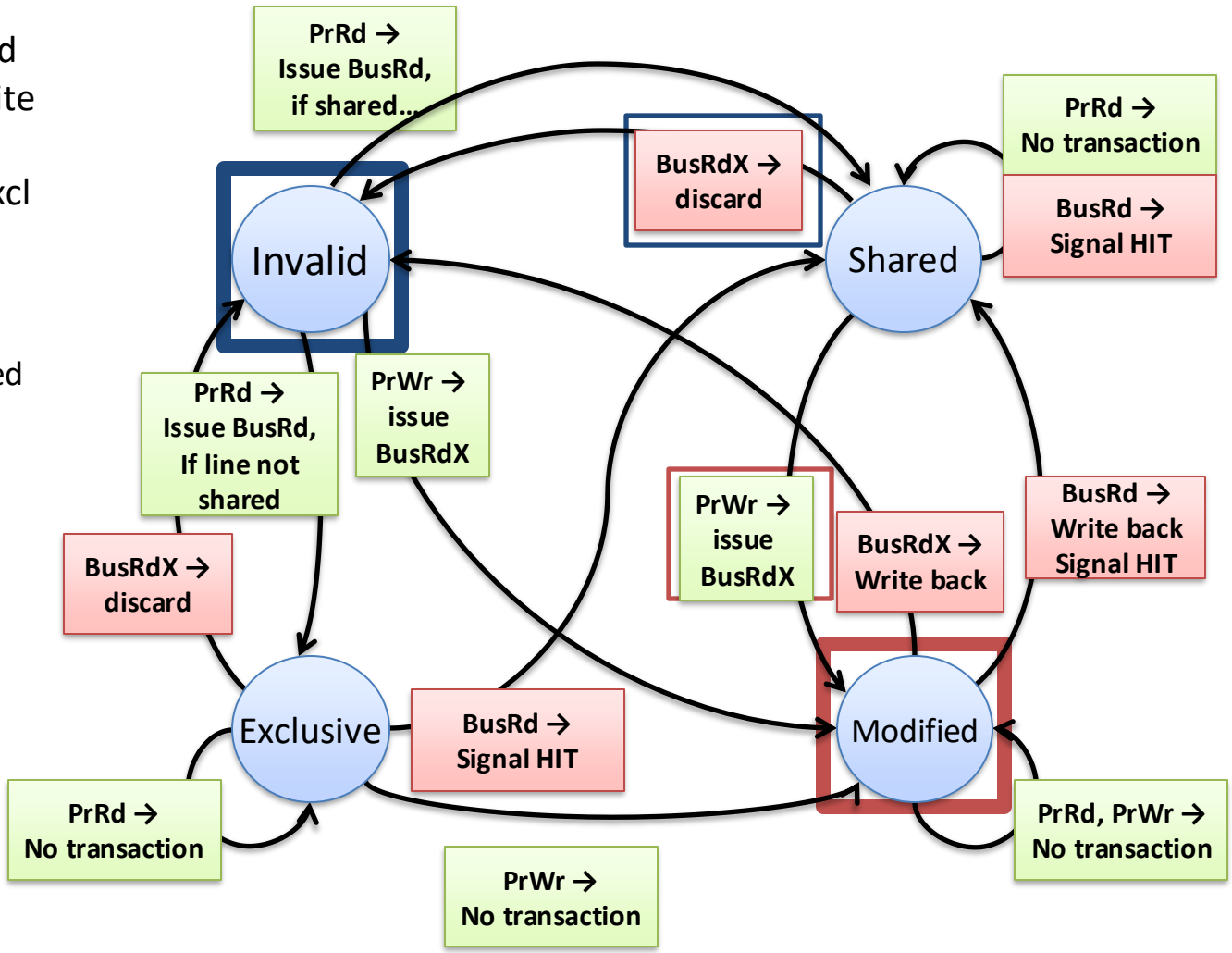
MESI

Terminology:

- PrRd: processor read
- PrWr: processor write
- BusRd: bus read
- BusRdX: bus read excl

Processor-initiated

Snoop-initiated



Core 0: write

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

MESI

- Problems:
  - Need to write back dirty data (no cache-cache transfer)
  - Either clean or dirty in (exactly) one cache

Overview

- Part of Assignment 10
- Recap Cache Coherence
- Hints for Assignment 11
- Quiz

Assignment 11

Systems@ETH zürich

- **Part 1: Pen & Paper**
  - Understanding cache coherence protocols (MSI and MESI)
- **Part 2: Programming Part**
  - Implement page table

Check correctness

Systems@ETH zürich

- **`$ ./correctness`**
  - Executes the executable of your pagetables.c (**`pt`**)
  - **`dropAddresses`** removes all dynamic address from the page-table entries in your output
  - Compares your page-table entries to **`solution`**
    - Match
    - Not match: save **`diff`** into a temporary file

Hints

Systems@ETH zürich

- Makefile
  - **`$ make pt`**
  - **`gcc $(CFLAGS) pagetables.c $(LIBS) -o pt`**
- Functions
  - Links a static library libdump.a (providing function dump_pagetable(pdbr);)
  - Generates executable **`pt`** (needed for **`./correctness`** script)

Hints

Systems@ETH zürich

- posix_memalign
  - The function **posix_memalign**() allocates *size* bytes and places the address of the allocated memory in **memptr*. The address of the allocated memory will be a multiple of *alignment*, which must be a power of two and a multiple of *sizeof(void *)*.

Overview

- Part of Assignment 10
- Recap Cache Coherence
- Hints for Assignment 11
- Quiz (following slides cover solutions only)

Question 1

Systems@ETH zürich

- VA: Virtual Address
- PA: Physical Address
- VPN: Virtual Page Number
- VPO: Virtual Page Offset
- PPN: Physical Page Number
- PPO: Physical Page Offset
- TLB: Translation Lookaside Buffer
- TLBI: TLB Index

Question 1

- TLBT: TLB Tag
- CT: Cache Tag
- CI: Cache Index
- CO: Cache Offset

Question 2

- Byte addressable
- VA = 14 bits
- PA = 12 bits
- Page is 64 bytes
- TLB: 2-way & 8 total entries
- Cache physically addr. direct mapped; 4-byte block & 16 sets

a) How much memory can a process address?

2^VA = 2^14 = 16 KB

b) How much memory can the processor address?

2^PA = 2^12 = 4 KB

c) How large are the VPN, VPO, PPN, PPO in bits?

VPO/PPO = 6 bits, VPN = 8 bits, PPN = 6 bits

d) How many pages can be referenced by a virtual address?

2^VPN = 2^8 = 256

Question 2

- Byte addressable
- VA = 14 bits
- PA = 12 bits
- Page is 64 bytes
- TLB: 2-way & 8 total entries
- Cache physically addr. direct mapped; 4-byte block & 16 sets

e) How many physical pages can the page table address?

2^PPN = 2^6 = 64

f) How many sets does the TLB have?

4 sets (8 total entries / 2 due to associativity)

g) How large are the TLBI and TLBT in bits?

TLBI = 2 bits (4 sets), TLBT = VPN – TLBI = 8 – 2 = 6

h) How large are the CT, CI and CO?

CO = 2 bits <= 4-byte blocks

CI = 4 bits, <= 16 direct mapped sets

CT = 6 bits <= PA – CO – CI = 12 – 2 – 4

Question 2

- Byte addressable
- VA = 14 bits
- PA = 12 bits
- Page is 64 bytes
- TLB: 2-way & 8 total entries
- Cache physically addr. direct mapped; 4-byte block & 16 sets

i) What would it mean if the cache were virtually addressed?

If the cache were virtually addressed, the cache would be used before the translation of virtual to physical addresses. This would mean that the cached values would only be valid for a single process and its virtual address space

j) How large would CT, CI and CO be in that case?

CO = 2 bits	**<= 4-byte blocks**
CI = 4 bits,	**<= 16 direct mapped sets**
CT = 8 bits	**<= VA – CO – CI = 14 – 2 – 4**

Question 2

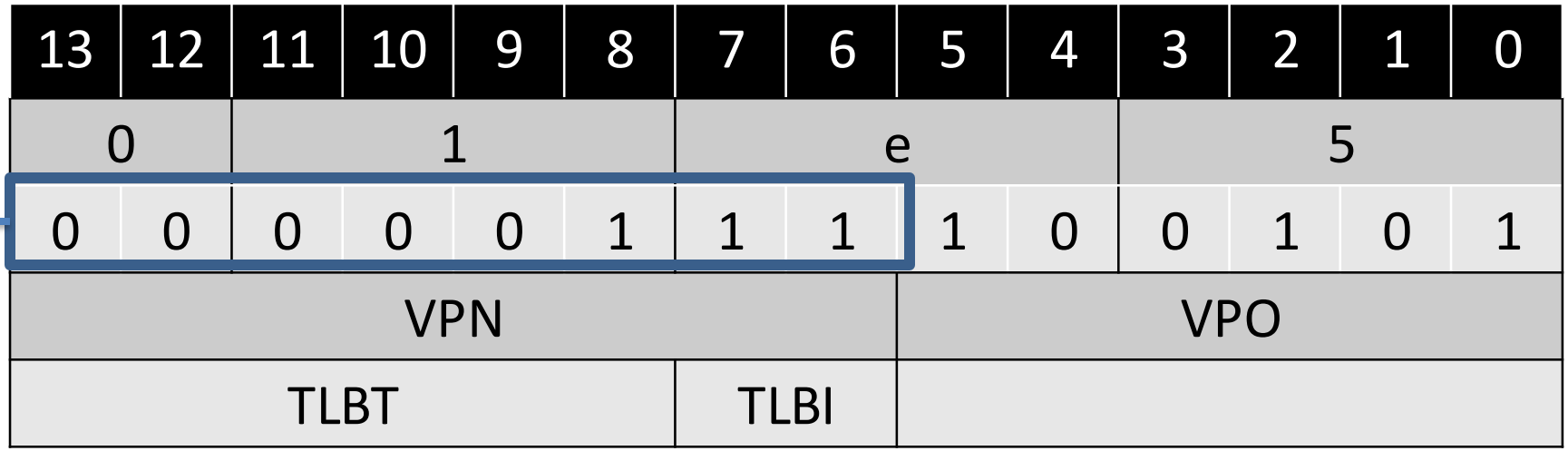
- Byte addressable
- VA = 14 bits
- PA = 12 bits
- Page is 64 bytes
- TLB: 2-way & 8 total entries
- Cache physically addr. direct mapped; 4-byte block & 16 sets

k) How would a memory access work in that case (virtually addressed cache)?

The processor would first look in the virtually addressed cache.
On a miss, the virtual address would be translated by the MMU and the physical address then used for a lookup in a physically addressed cache or into memory.

Question 3

- VA=0x01e5



VPN = 0x7

Page table: (all values are hexadecimal)

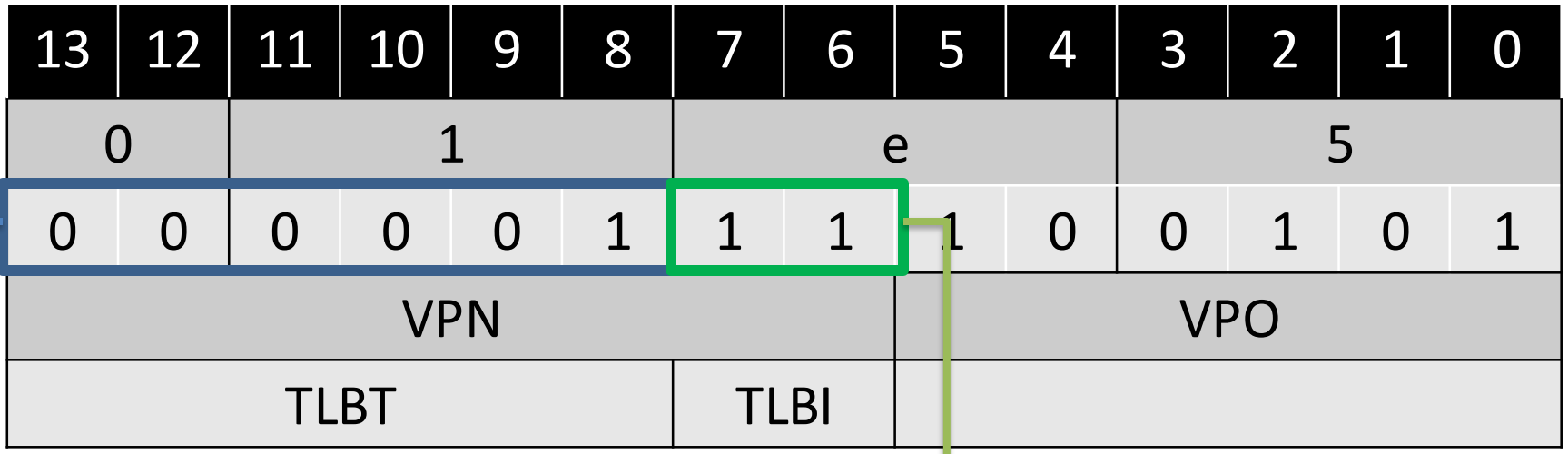
VPN	PPN	Valid
00		
01	02	1
02	03	1
03		
04		
05		
06	22	1
07	22	1

HIT

VPN	PPN	Valid
08		
09	20	1
0a		
0b		
0c		
0d	04	1
0e		
0f		

Question 3

- VA=0x01e5



TLBT= 0x1

TLBI= 0x3

HIT

TLB: (all values are hexadecimal)

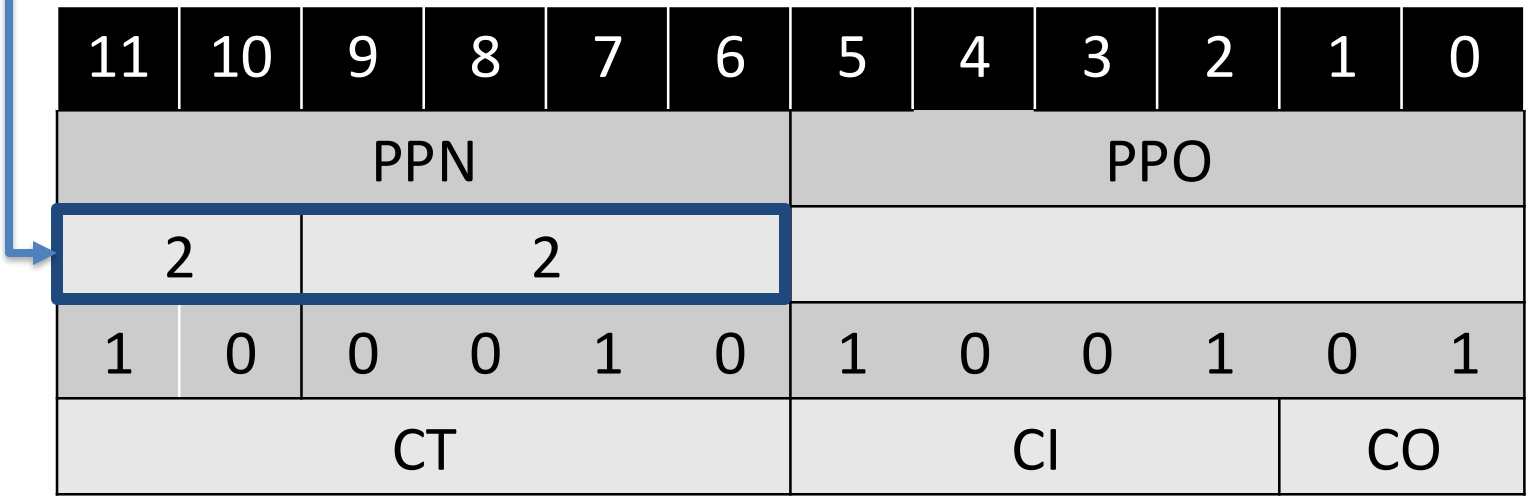
Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	01	22	1	07		0
3	01	22	1	02		0

Question 3

- VA=0x01e5

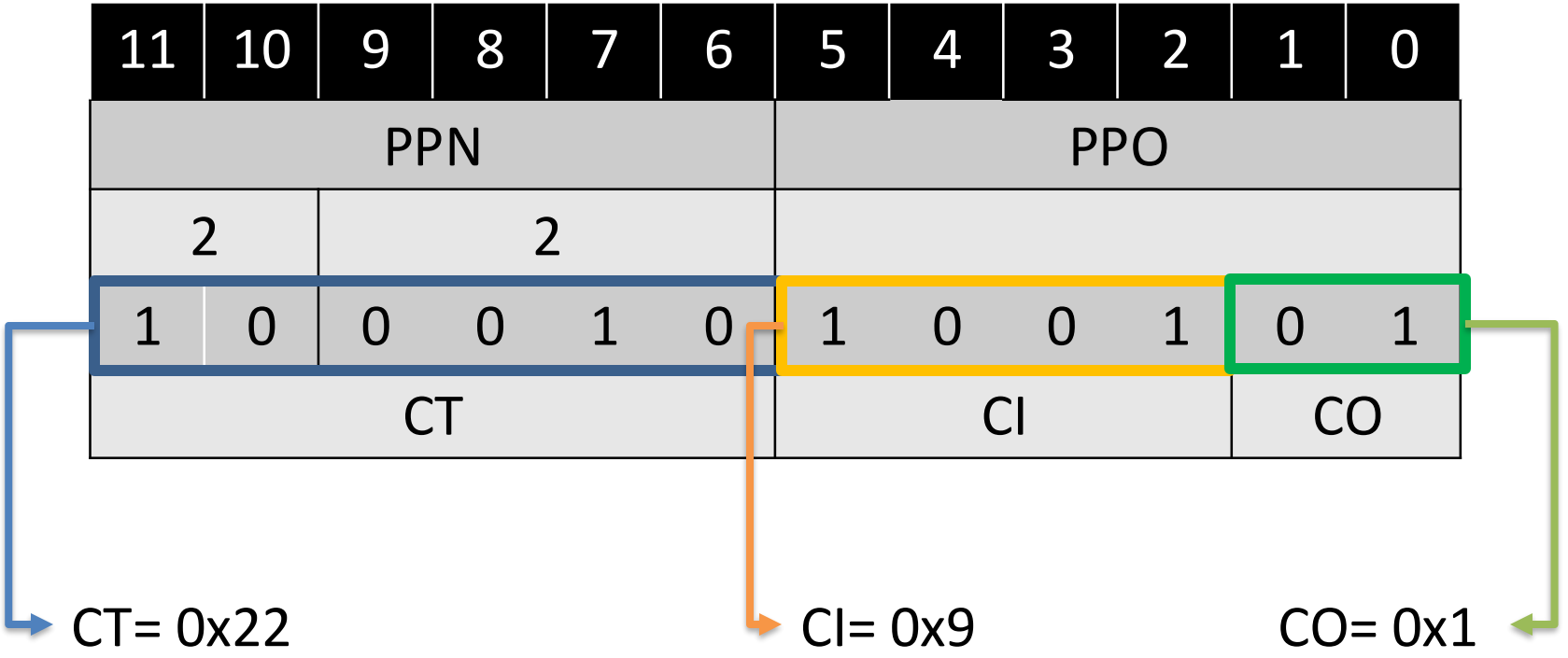
TLB: (all values are hexadecimal)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	01	22	1	07		0
3	01	22	1	02		0



Question 3

- VA=0x01e5



Question 3

- VA=0x01e5

Cache: (all values are hexadecimal)

Index	Tag	Valid	Block[0]	Block[1]	Block[2]	Block[3]
00	00	1	de	ad	fa	ce
01	31	0				
02	24	1	02	13	e1	de
03	22	1	22	23	e2	2e
04	21	0				
05	22	0				
06	18	0				
07	22	1	9a	01	00	de
08	20	0				
09	22	1	83	1a	09	ce
0a	20	1	0d	1f	f1	d0
0b	3a	0				
0c	3f	0				
0d	24	1	be	fb	57	02
0e	23	0				
0f	22	1	cf	7a	9b	a0

CI= 0x9

CT= 0x22

HIT

CO= 0x1

Have a nice rest of week