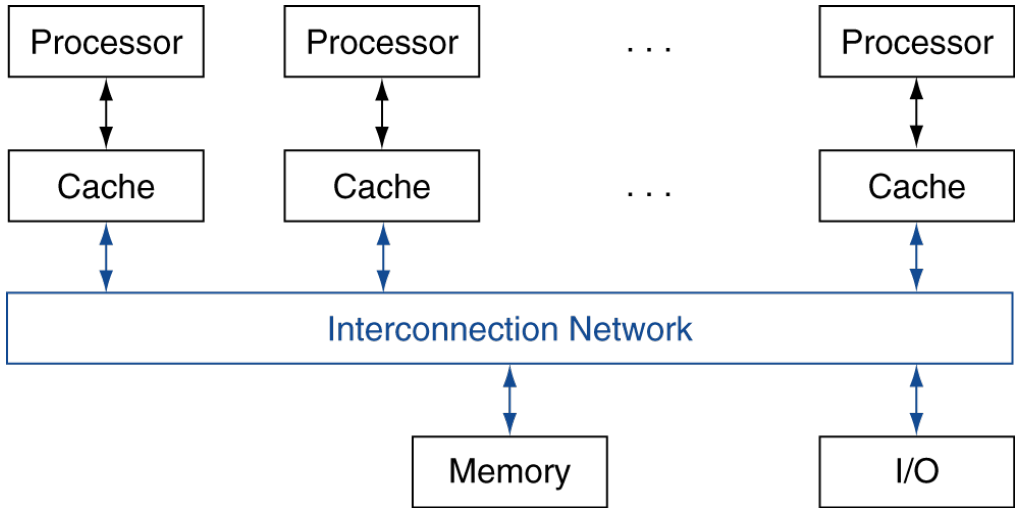
# Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



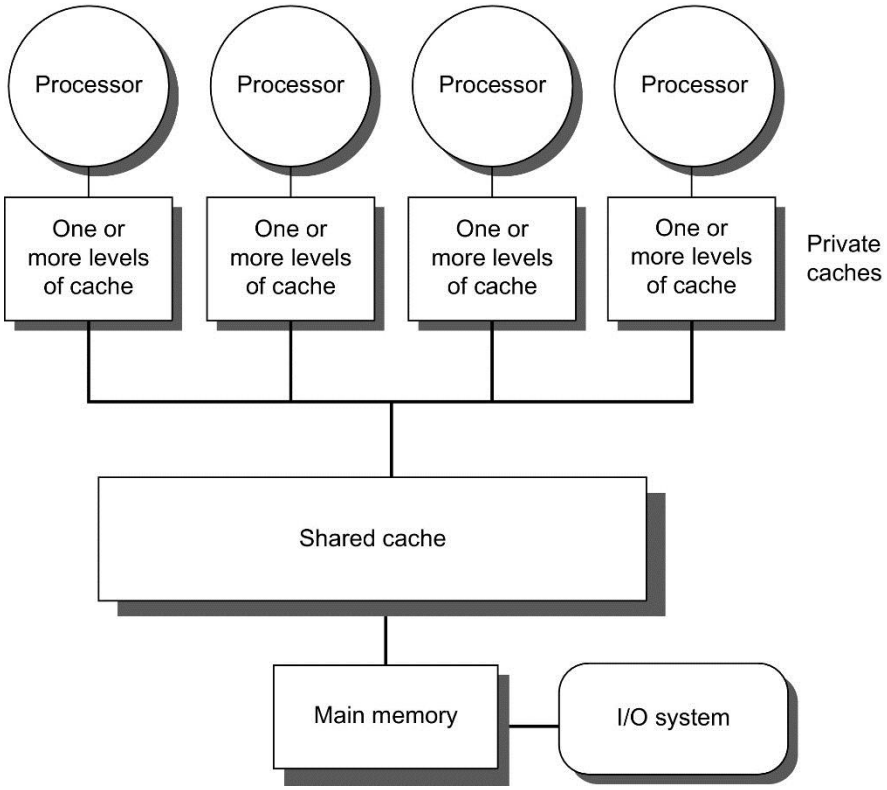



**Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip**. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache on the multicore, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip design, an interconnection network links the processors and the memory, which may be one or more banks. In a single-chip multicore, the interconnection network is simply the memory bus.

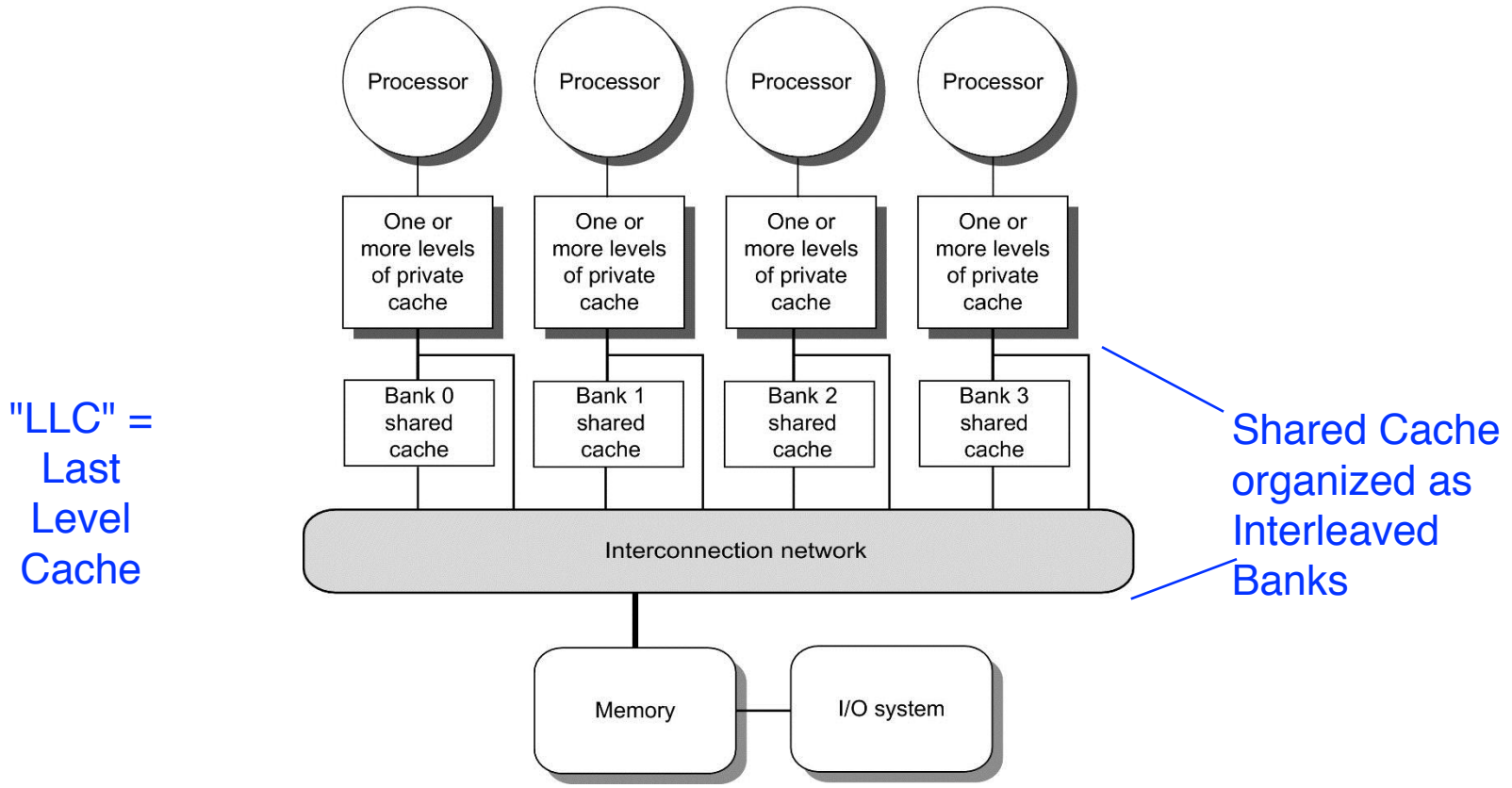



**Figure 5.8 A single-chip multicore with a distributed cache**. In current designs, the distributed shared cache is usually L3, and levels L1 and L2 are private. There are typically multiple memory channels (2–8 in today's designs). This design is NUCA, since the access time to L3 portions varies with faster access time for the directly attached core. Because it is NUCA, it is also NUMA.

# Cache Coherence



- Coherence (Συνοχή)
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors
- Consistency (Συνέπεια)
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

# Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

| Time step | Event | CPU A's cache | CPU B's cache | Memory |
|---|---|---|---|---|
| 0 | | | | 0 |
| 1 | CPU A reads X | 0 | | 0 |
| 2 | CPU B reads X | 0 | 0 | 0 |
| 3 | CPU A writes 1 to X | 1 | 0 | 1 |

# Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)
    ⇒ read returns written value
  - $P_1$ writes X; $P_2$ reads X (sufficiently later)
    ⇒ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - $P_1$ writes X, $P_2$ writes X
    ⇒ all processors see writes in the same order
    - End up with the same final value for X

# Cache Coherence Protocols

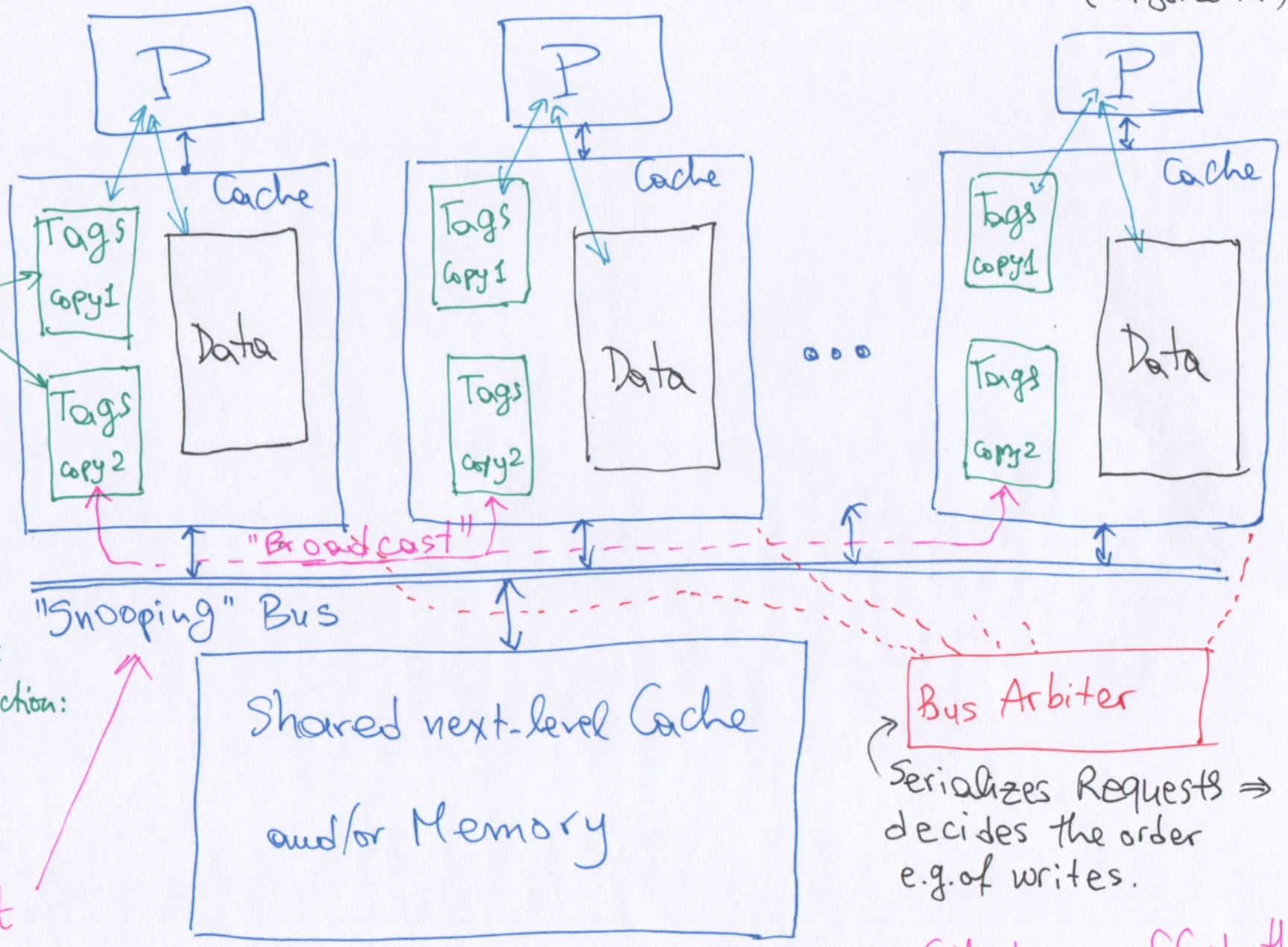
- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

All activities that potentially affect other caches are broadcast onto the shared bus; all caches monitor ("snoop") that shared bus.

OK for few (4, 8, 16?) sharers, but too much traffic beyond ~8.

A central Directory (may consist of interleaved banks) records which caches have copies of which blocks => only "bother" those caches that are affected

# Snooping Cache Coherence (Συνοχή κρυφής Μνήμης με κρυφοκοίταγμα/καταπολέμηση (κουτσομπολιό στη γειτονιά))




Two copies of tags, or 2-port TagsMemory: mostly read-only; only need to write infrequently: on misses or when affected by bus transaction: then, have to write into both copies.

Broadcast Medium: everybody has to listen to everything going on (that may affect others) — when does it affect others ???

Serializes Requests ⇒ decides the order e.g. of writes.

# Upon Writes: to Invalidate or to Update the Others?

- **Invalidate-based Protocols:**

  When I write (modify) something that I have, and there is a danger others may have copies of it, tell them to invalidate their copies!

  (like telling them: "If you ever need it again, come back to me and ask me for its latest value")

  - Advantage: from that point on, I know that I have the only copy, therefore I can freely "play with it", as long as nobody ask me for a copy again.

  (most usual)

- **Update-based Protocols:**

  When I write (modify) something that I have, and there is a danger others may have copies of it, broadcast the new value so as to update the values of all copies!

  - Advantage: if others will need the value again soon, they will have it in their caches
  - Disadvantage: multiple copies will keep existing, hence the need to continuously keep updating them

  (rarely (if ever) used)

(statistics showed that disadvantage is important in the general case (unless there is hint about some data by the algorithm/software)

# Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

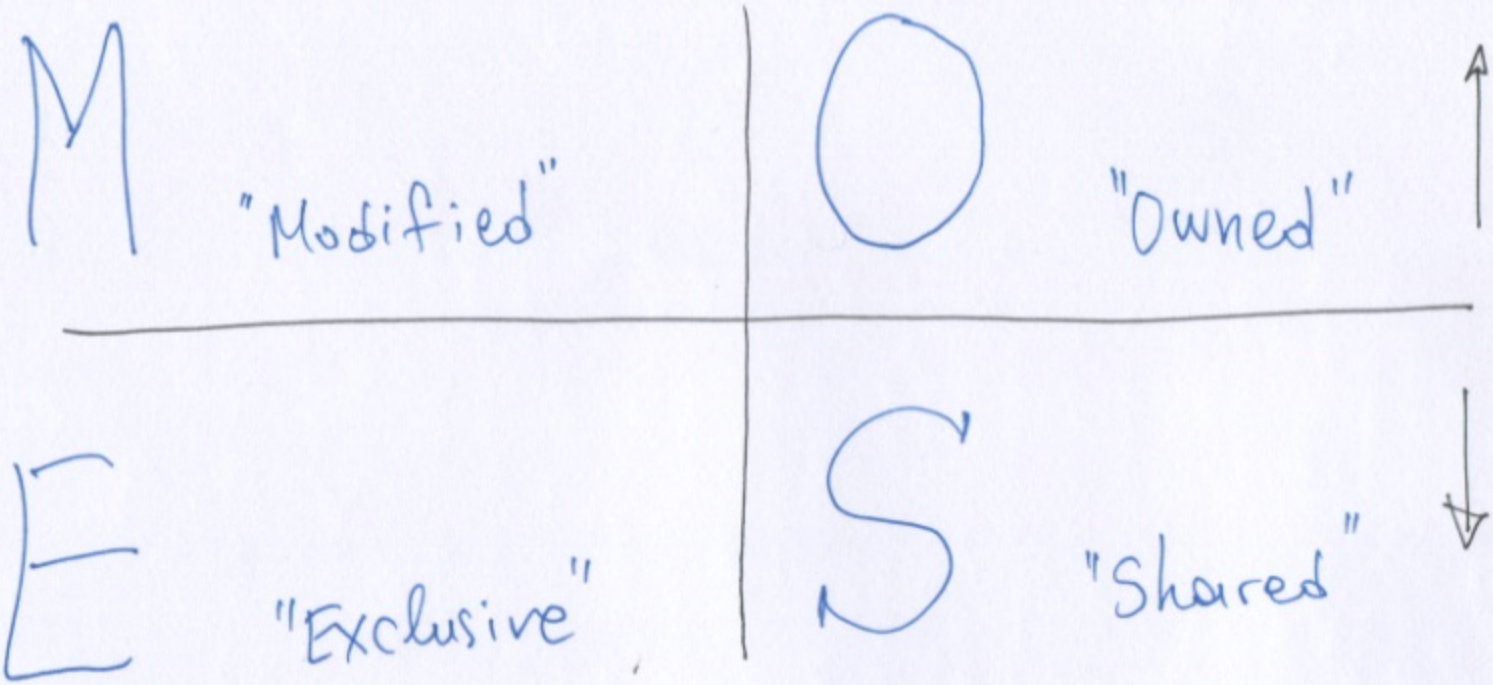
| CPU activity | Bus activity | CPU A's cache | CPU B's cache | Memory |
|---|---|---|---|---|
| | | | | 0 |
| CPU A reads X | Cache miss for X | 0 | | 0 |
| CPU B reads X | Cache miss for X | 0 | 0 | 0 |
| CPU A writes 1 to X | Invalidate for X | 1 | | 0 |
| CPU B read X | Cache miss for X | 1 | 1 | 1 |

Write-Back

# Snoopy Coherence Protocols

- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor
- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

# Cache Line States: what do I know about it? "MOESI"




Dirty: I have modified my copy, and I am responsible for writing it back to memory

Clean: It is guaranteed that the memory (and others) have the same value as I do.

Exclusive:
It is guaranteed that my copy is the ONLY copy currently existing in any cache.

(potentially) Shared:
other caches may have copies of this line (not known for sure)

I "Invalid"
nothing in this Line.

# Simplification: MSI Protocol

(No "E" info when Clean; If Dirty, must be Exclusive (no "O" state))

**M (Modified): Dirty and Exclusive**

- with invalidate-based protocol, when I write into my copy, I have to invalidate all others, hence I am guaranteed to have the ONLY copy ⇒ Exclusive

**S (Shared): (potentially) Shared and guaranteed Clean**

- when first reading from memory → S (no "E" info maintained)
- for simplicity, I do not keep track whether or not others too may have copies
- if I had the line as M, and another cache misses it, then:
  - I have to write it back to memory (no "O" state, for simplicity)
  - the other cache gets a copy automatically on the bus
- I may have had the line as S, and all other caches may have evicted their copies, so I may be the only one having a copy, but I do NOT know that for sure...
- If I have the line as S, and I want to write into it, I must broadcast an Invalidate command, since I have no "E" info!

**I (Invalid)**

# from MSI to MESI protocol:

- Add and maintain E (exclusive and clean) info:
  - When I read-miss, if no other cache responds (faster than memory) providing me the data, and I have to wait for the memory to bring me the data ⇒ then I know I am "E".
  - Advantage versus MSI: If the line is "E" and I want to write into it, I do NOT need to broadcast any invalidate: save bus traffic.

# MOESI protocol:

- Add "O" (Owned) info: Line is shared, Memory is "old", and the responsibility to write back is mine!
  - When the line is "M" (Modified: dirty and exclusive), and another cache read-misses on it, I supply the data to the other cache (faster than memory), but I do not spend the time to write-back to memory (now): save time (for now).
  - The other caches that have copies, have them in S state, hence they are allowed to evict their copies without writing back: I have the (only) "O" copy, hence the responsibility to write-back is mine!

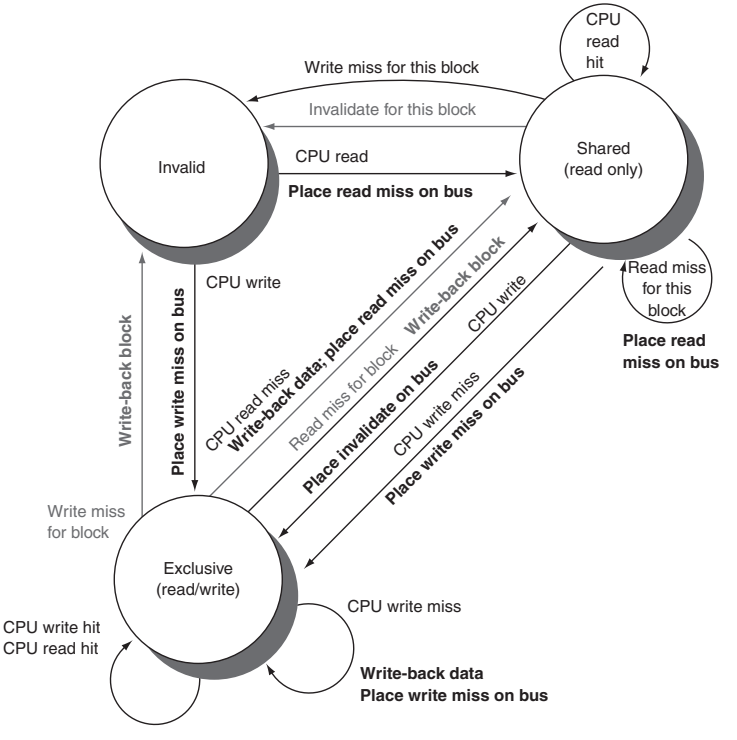



**FIGURE e5.12.11 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray.** As in Figure e5.12.10, the activities on a transition are shown in bold.

# Advanced Processors

## Instruction-Level Parallelism (ILP)

# Multiple Issue

Fetch multiple (e.g. 2, 4) instructions in parallel, and then consider how many and which of them to execute in parallel

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards where none independent available, fills-in noop’s
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

# Static Multiple Issue

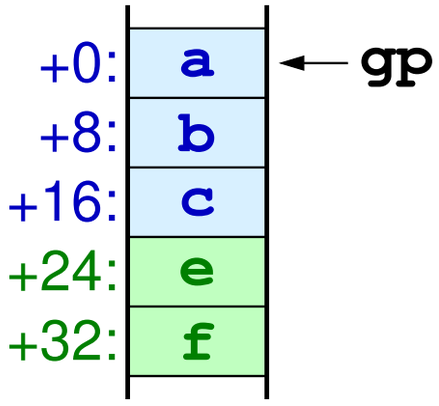
- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

# Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with in a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

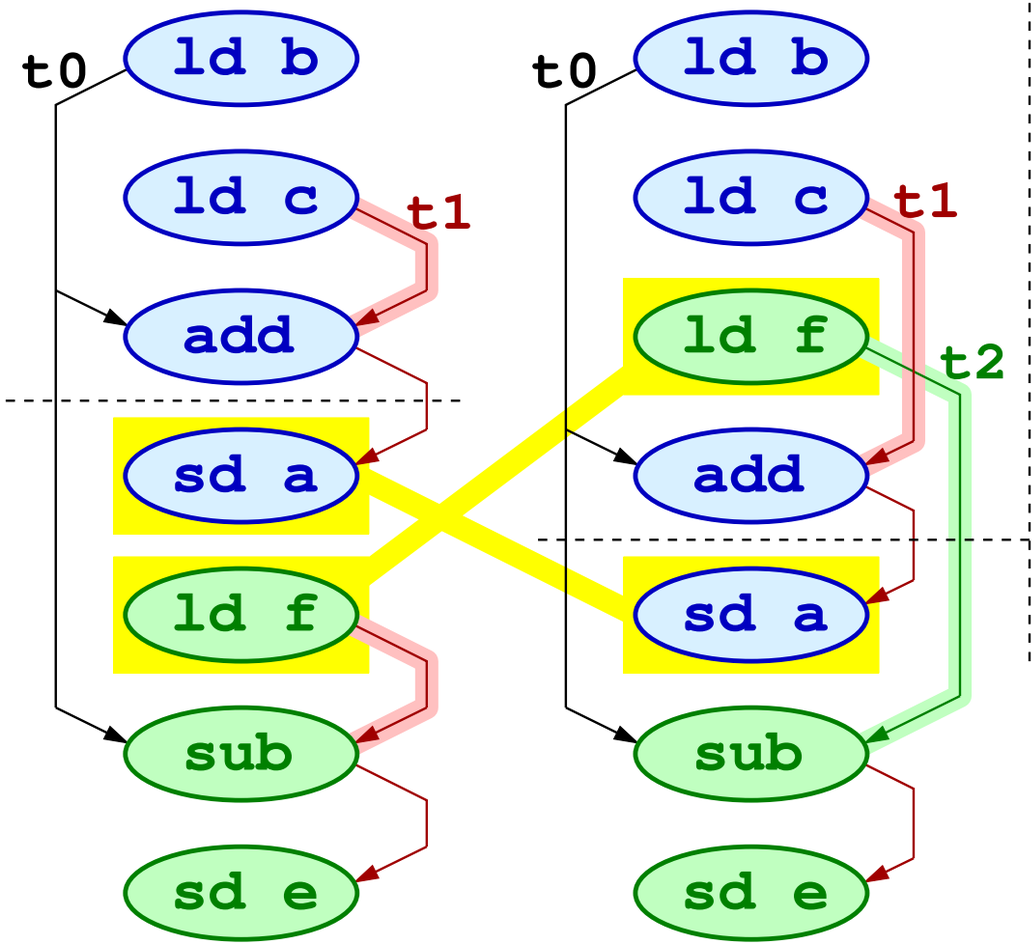
# Instruction Scheduling

```
a = b + c;
e = b - f;
```




```
ld  t0,  8(gp)
ld  t1, 16(gp)
add t1, t0, t1
sd  t1,  0(gp)
ld  t1, 32(gp)
sub t1, t0, t1
sd  t1, 24(gp)
```

*2 extra clock cycles lost*




*This is 'Static' Scheduling, at Compile Time*

the more things you have 'up in the air' (in parallel), the more temporary registers you need in order to 'name' those 'pending' values

```
ld  t0,  8(gp)
ld  t1, 16(gp)
ld  t2, 32(gp)
add t1, t0, t1
sd  t1,  0(gp)
sub t1, t0, t2
sd  t1, 24(gp)
```

*No extra clock cycle lost*

*What if the program is?:*
*RAW dependence?*

```
a[i] = b + c;
e = b - a[j];
```

- Does the compiler know for sure if i!=j (OK to reorder sd–ld) or i==j (fwd in reg.)?
- If unknown to compiler, static sch. impossible => dynamic scheduling at runtime (ooo pipe)

# Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle (checks dependencies and)
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Allows executables to run on newer processors, with same ISA but different pipeline, without needing to be recompiled

# Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - Can start sub while add is waiting for ld

Out-of-Order (ooo) Execution

In-Order Commit

(so as to flush results of mis-speculated instructions, and also allow precise exceptions)

# Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

# Does Multiple Issue Work?

**The BIG Picture**

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

# Parallelism

# Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

e.g. parallelizable

e.g. non-parallelizable

e.g. available parallelism

80 seconds out of total 100 seconds

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- Corollary: make the common case fast

# Strong vs Weak Scaling

- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

# Instruction and Data Streams



- An alternate classification

Multimedia (e.g. MMX)
Streaming SIMD Extension (SSE)

| | | Data Streams | |
|---|---|---|---|
| | | Single | Multiple |
| Instruction Streams | Single | **SISD**: Intel Pentium 4 | **SIMD**: SSE instructions of x86 |
| | Multiple | **MISD**: No examples today | **MIMD**: Intel Xeon e5345 |

Vector

often

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

# SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

# Vector Processors

Data-Level Parallelism
Identical & Independent operations on all elements of a vector (array) - one vector instr. replaces a loop

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to RISC-V
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
  - Vector instructions
    - `fld.v`, `fsd.v`: load/store vector
    - `fadd.d.v`: add vectors of double
    - `fadd.d.vs`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector Length Register assists in counting the number of remaining elements to process

Sequential memory addresses, or strided (e.g. for 2D/3D arrays), or scatter-gather (via array of pointers

# Vector vs. Scalar

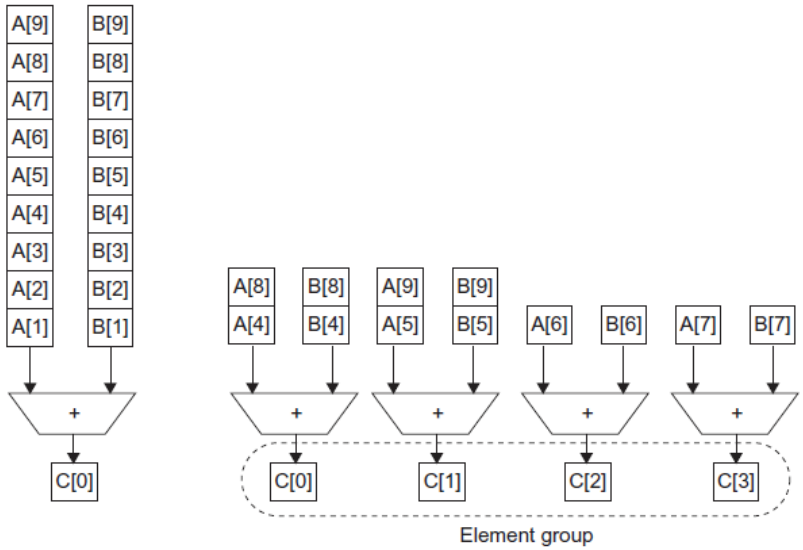
- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

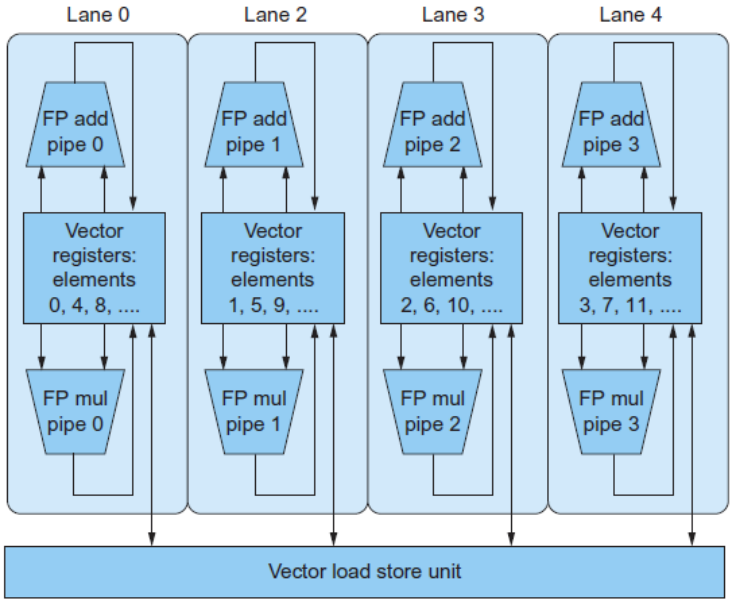
# Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width

length

- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:



# Multithreading

One "thread of control" = one (traditional) sequential program.
Multiple threads = parallel program.



- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

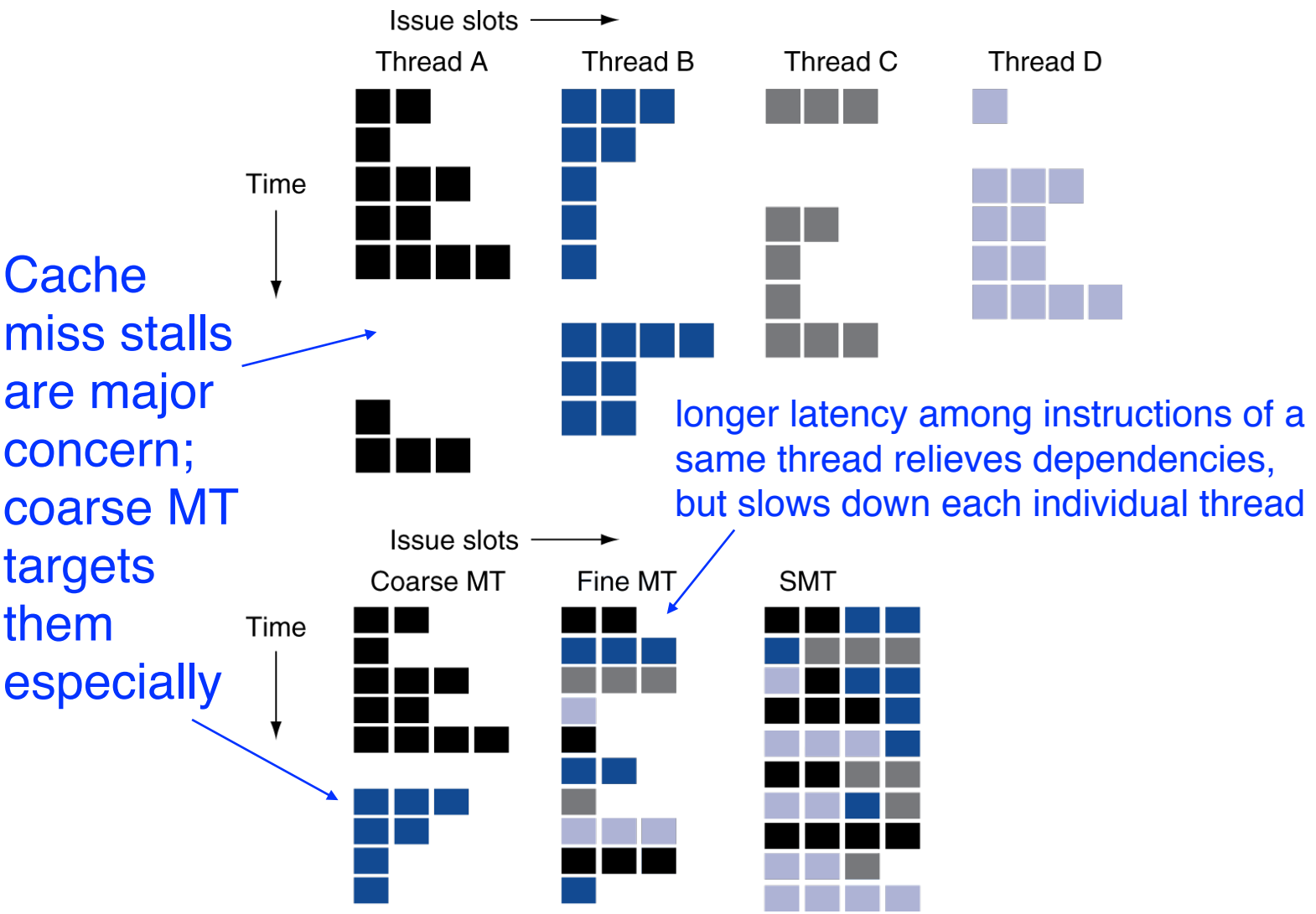
mimic multiple cores, thus:

but Share the Functional Units and the Caches

# Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

# Multithreading Example



# Future of Multithreading

- Will it survive? In what form?
- Power considerations ⇒ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Two different threads may have two different working sets of data/instructions; is it better to place them in a single cache, or in two different caches as two separate cores would do?

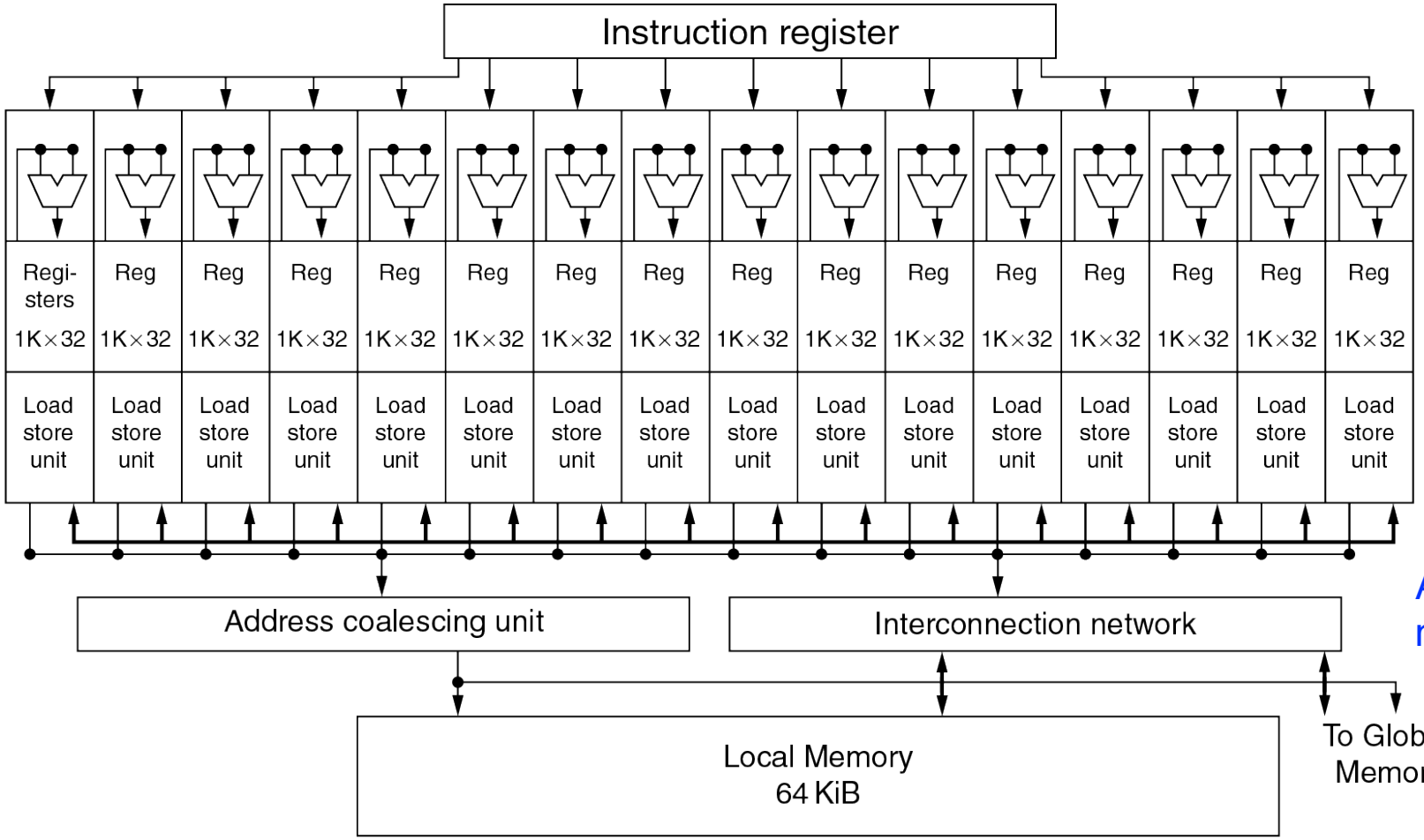
# GPU Architectures

Graphics Processing Units

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

# Example: NVIDIA Fermi

- Multiple SIMD processors, each as shown:




16 lanes

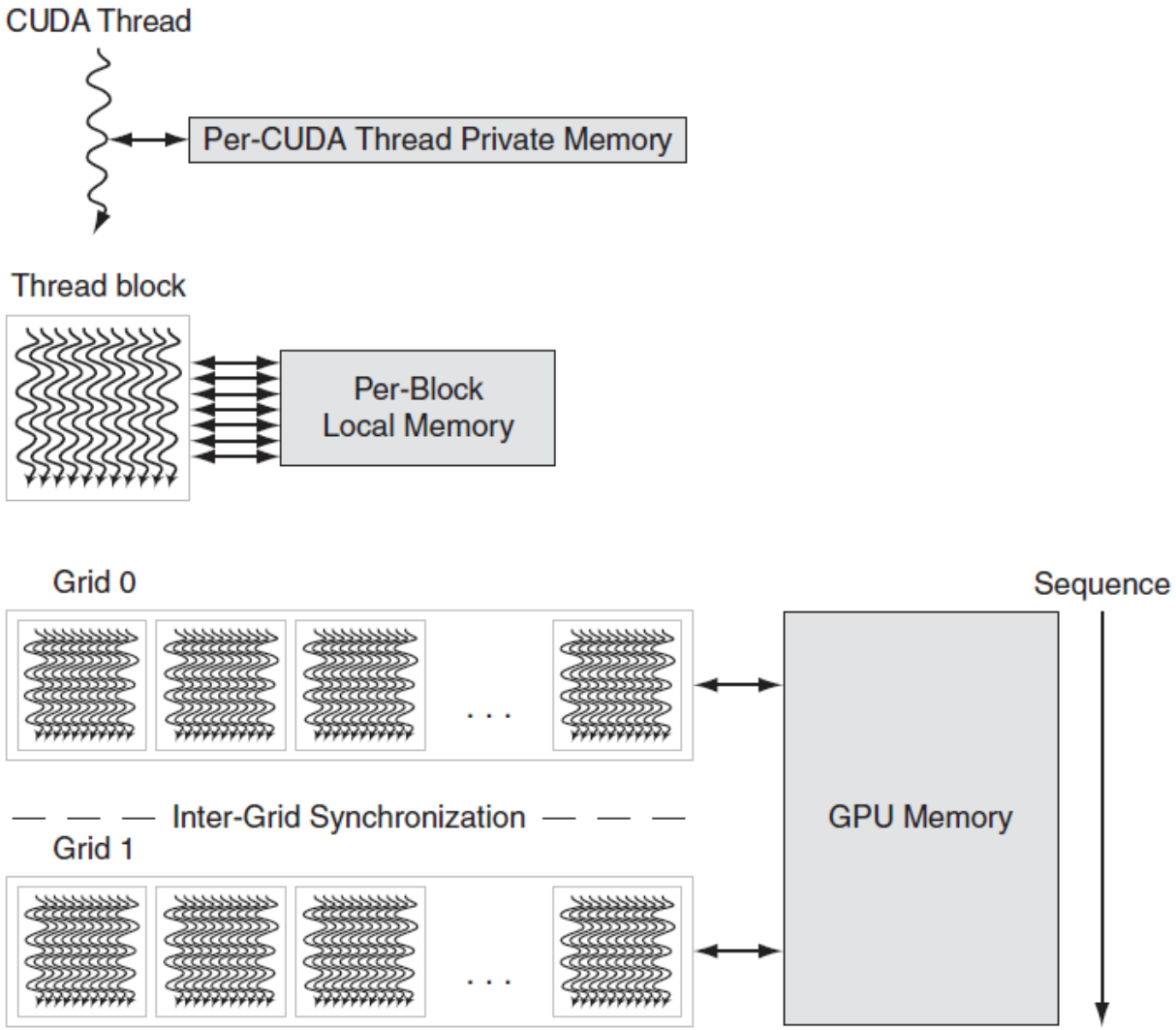
32-element SIMD instructions

Also, massively multithreaded

# Example: NVIDIA Fermi

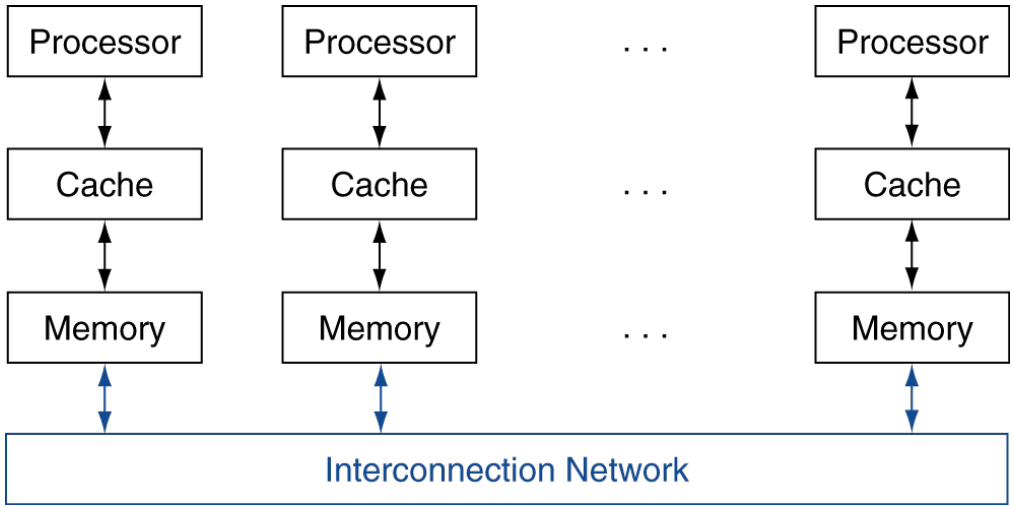
- SIMD Processor: 16 SIMD lanes
- SIMD instruction
  - Operates on 32 element wide threads
  - Dynamically scheduled on 16-wide processor over 2 cycles
- 32K x 32-bit registers spread across lanes
  - 64 registers per thread context

# GPU Memory Structures



# Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



# Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP