Threaded Programming

Lecture 8: Node Architecture

Building Blocks of HPC Systems

Four principal technologies which make up HPC systems:

- Processors
  - to calculate
- Memory
  - for temporary storage of data
- Interconnect
  - so processors can talk to each other (and the outside world)
- Storage
  - disks for storing input/output data and tapes for long term archiving of data
- We will focus on the first two of these.

Processors

- Basic functionality
  - execute instructions to perform arithmetic operations (integer and floating point)
  - load data from memory and store data to memory
  - decide which instructions to execute next

- Arithmetic is performed on values in registers (local storage in the processor)
  - moving data between memory and registers must be done explicitly by load and store instructions
  - separate integer and floating point registers
  - typical size ~100 values

- Basic characteristics:
  - Clock speed
  - Peak floating point capability

Processors (cont.)

- Clock speed determines rate at which instructions are executed
  - modern chips are around 2-3 GHz
  - integer and floating point calculations can be done in parallel
  - can also have multiple issue, e.g. simultaneous add and multiply
  - peak flop rate is just clock rate x no. of floating point operations per clock cycle

- Whole series of hardware innovations
  - pipelining
  - out-of-order execution, speculative computation
  - ...

Moore's Law

- "CPU power doubles every 18 months"
  - strictly speaking, applies to transistor density
- Held true for ~40 years
  - now maybe self-fulfilling?
- People have predicted its demise many times
  - may finally be actually happening?
- Increases in power are due to increases in parallelism as well as in clock rate
  - fine grain parallelism (pipelining)
  - medium grain parallelism (superscalar, SIMD, hardware multithreading)
  - coarse grain parallelism (multiple processors on a chip)
- First two seem to be (almost) exhausted: main trend is now towards multicore

Functional units

- Functional units are the basic building blocks of processors.
  - Number and type of units varies with processor design.
- Core units for any processor include:

Instruction unit

- Responsible for fetching, decoding and dispatching of instructions.
- Fetches instruction from instruction caches
- Decodes instruction.
- Sends the instructions to the appropriate unit.
- May also be responsible for scheduling instructions (see later).

Functional units

Integer unit

- Handles integer arithmetic
- Integer addition, multiplication and division.
- Logical ops (and, or, shift etc.)
- Also known as arithmetic and logic unit (ALU)

Floating point unit

- Handles floating point arithmetic
- Addition, multiplication, division.
- Usually the critical resource for HPC
  - Machines sold by peak flop/s

Functional units

Control unit

- Responsible for branches and jumps

Load/store unit

- Responsible for loading data from memory and storing it back.

Register file

- Local storage in the CPU
- Accessed by name (not address)
- Separate files for integers/addresses and floating point

- Also memory management, cache controller, bus interface, graphics/multimedia,.....

Pipelining

- Key implementation technique for making fast CPUs.
- Execution of instructions is broken down into stages.
- Each stage can be executed in one CPU clock cycle
  - all parts of CPU operate at a fixed frequency
- Once a stage has completed for one instruction, it can be executed for the next instruction on the subsequent clock cycle.
- Allows one instruction to be completed per clock cycle, even though the instruction itself may take many cycles to complete.
  - Intel Ivy Bridge has 14 stage pipeline
- But not all instructions are pipelined
  - e.g. FP square root, FP divide

Problems of pipelines

Any of the following can result in stopping and restarting the pipeline, and wasting cycles as a result:

- Two instructions both require the same hardware resource at the same time
- One instruction depends on the result of another instruction further down the pipeline
- The result of instruction changes which instruction to execute next (e.g. branches)

Overcoming pipeline hazards

- Out-of-order execution
  - Assembly code specifies an order of instructions....
  - But the hardware chooses to reorder instructions as they are fetched to minimise pipeline stalls.
  - Requires some complex bookkeeping to ensure correctness.
- Branch prediction
  - Hardware tries to guess which way the next branch will go
  - Uses a hardware table that tracks the outcomes of recent branches in the code.
  - Keeps the pipeline going and only stalls if prediction is wrong

Instruction level parallelism

- Pipelining is a form of instruction level parallelism (ILP)
  - multiple instructions are “in-flight” at the same time.
  - but maximum performance is 1 instruction per cycle
- Also possible to exploit ILP at a higher level
  - identify instructions that can be executed independently
  - use different functional units (no structural hazards)

Two main approaches:

- Superscalar processors
  - parallel instructions identified at run-time, in hardware.
- SIMD (or vector) instructions
  - operations on multiple data items encoded in a single instruction

Superscalar processors

- Divide instructions up into classes which use different resources
  - most obvious is integer and floating point
- If two or more instructions are in different classes, they can be issued on the same clock cycle, and proceed in parallel
  - could issue integer add, FP multiply in same cycle
- Can be combined with out of order execution

Superscalar processors

- Detection of independent instructions is done in hardware.
  - fetch several instructions at once
  - decide whether they can be issued on one clock cycle, or spread across more than one cycle
  - need to take structural and data hazards into account.

- Scheduling can be helped by compiler techniques
  - grouping instructions together in ways which favour multiple issue

SIMD instructions

- Instructions which encode operations on multiple data items (**S**ingle **I**nstruction **M**ultiple **D**ata)

Example:

A simple floating point instruction might encode

```
c = a + b
```

A SIMD floating point instruction could encode

```
c = a + b  and  f = d + e
```

in one instruction.

SIMD

- Most modern processors include support for SIMD instructions
  - e.g. SSE, AVX in x86, AltiVec in POWER
- Requires a (substantial) extension to the instruction set
  - load/stores, compares, as well as integer and FP arithmetic
- Requires multiple arithmetic units and additional vector registers to take advantage of them
- The SIMD width is the number of operations encoded in a SIMD instruction
  - typically 2, 4 or 8

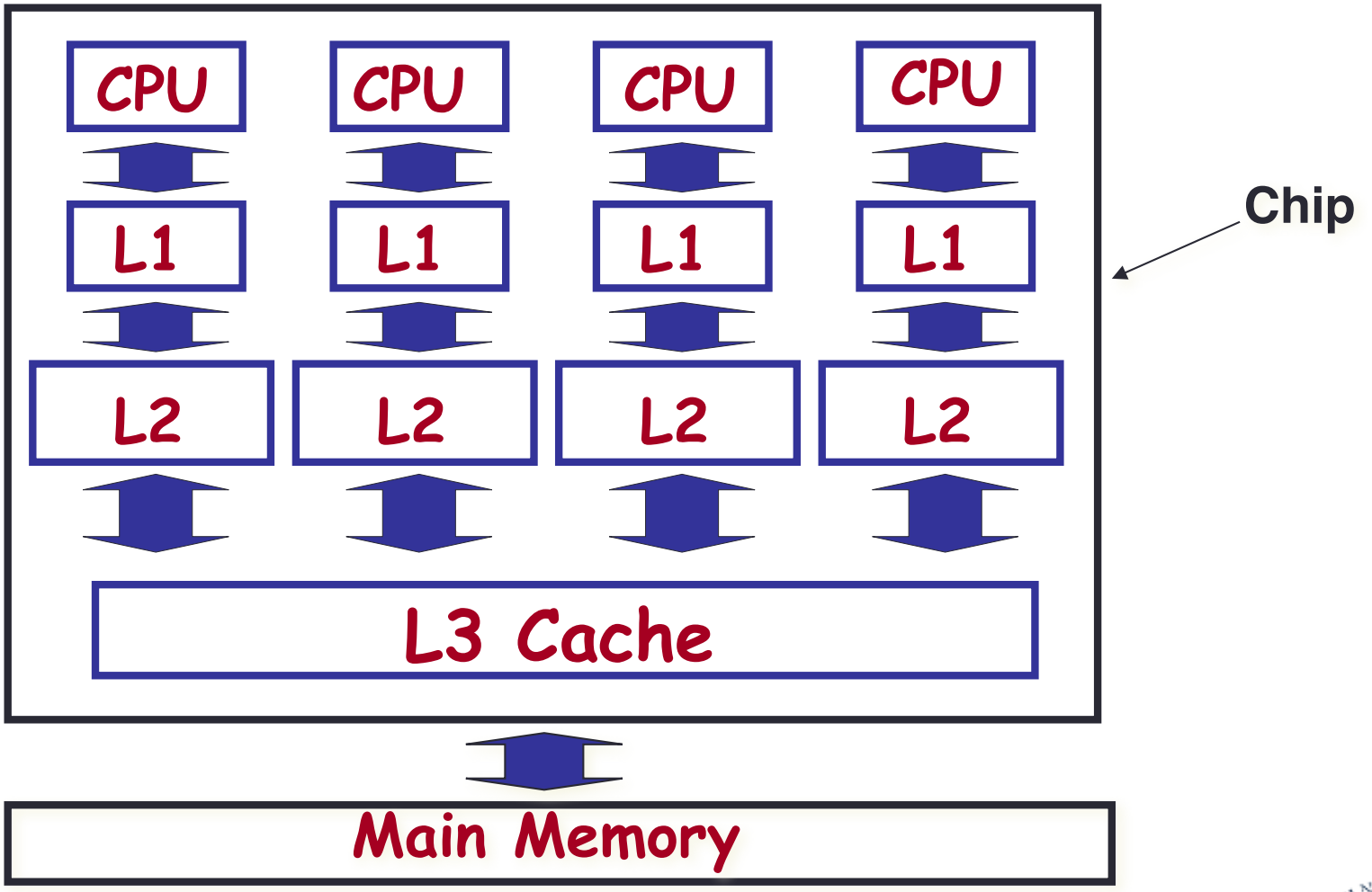
Exploiting SIMD instructions

- Restrictions on what SIMD memory operations (loads/stores) can do
  - e.g. a 2-word SIMD load of double precision values may require that the values are consecutive in memory and are 16 byte aligned.
- Compiler is responsible for identifying operations which can be combined into SIMD instructions.
- In practice, this is a tough job for the compiler.
  - alignment of data is often not determinable at compile time
  - compiler can generate multiple versions of code and pick the right one depending on the alignment encountered at run time…..
  - may require hints, or even hand-coded assembly

Multicore chips

- Now commonplace to have multiple processors on a chip.
- Main difference is that processors may share caches
- Typically, each core has its own Level 1 and Level 2 caches, but the Level 3 cache is shared between cores
- May also share other functional units
  - i.e. FPU

Typical cache hierarchy



- This means that multiple cores on the same chip can communicate with low latency and high bandwidth
  - via reads and writes which are cached in the shared cache
- However, cores contend for space in the shared cache
  - one thread may suffer capacity and/or conflict misses caused by threads/processes on another core
  - harder to have precise control over what data is in the cache
  - if only single core is running, then it may have access to the whole shared cache
- Cores also have to share off-chip bandwidth
  - for access to main memory

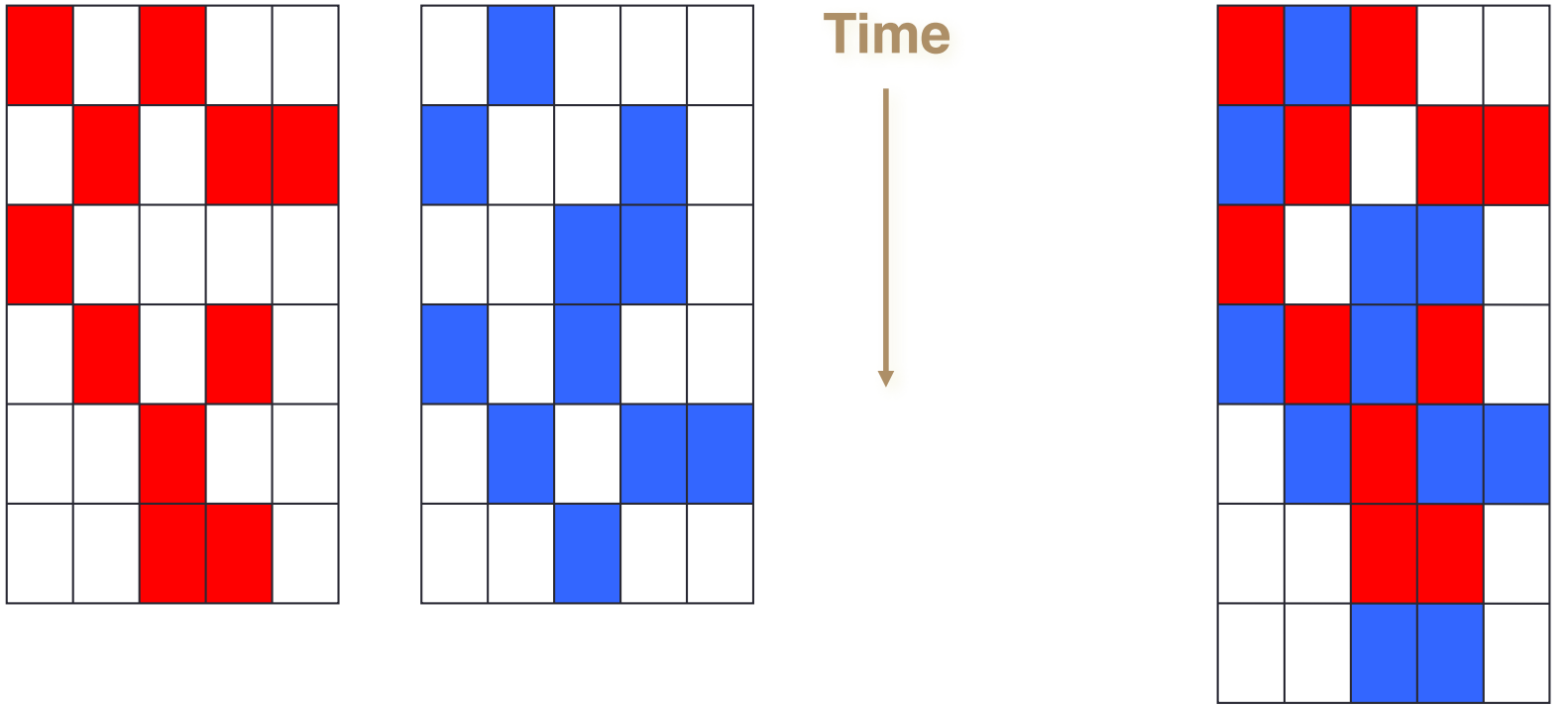
Empty instruction slots

- Most modern processors are superscalar
  - can issue several instructions in every clock cycle
  - selection and scheduling of instructions is done on-the-fly, in hardware
- A typical processor can issue 4 or 5 instructions per clock, going to different functional units
  - obviously, there must be no dependencies between instructions issue on the same cycle
- However, typical applications don't have this much instruction level parallelism (ILP)
  - 1.5 or 2 is normal
  - more than half the available instruction slots are empty

SMT

- Simultaneous multithreading (SMT) (a.k.a. Hyperthreading) tries to fill these spare slots by mixing instructions from more than one thread in the same clock cycle.
- Requires some replication of hardware
  - instruction pointer, instruction TLB, register rename logic, etc.
  - Intel Xeon only requires about 5% extra chip area to support SMT
- ...but everything else is shared between threads
  - functional units, register file, memory system (including caches)
  - sharing of caches means there is no coherency problem
- For most architectures, two or four threads is all that makes sense

SMT example



Two threads on two CPUs

Two threads on one SMT CPU

More on SMT

- How successful is SMT?
  - depends on the application, and how the threads contend for the shared resources.
- In practice, gains seem to be limited to around 1.2 to 1.3 times speedup over a single thread.
  - benefits will be limited if both threads are using the same functional units (e.g. FPUs) intensively.
- For memory intensive code, SMT can cause slow down
  - increased contention for memory bandwidth and/or cache space
  - increasing the number of threads/processes can increase overheads such as communication or load imbalance

Accelerators

- Current popular trend is to include additional special purpose processors alongside the main CPU
  - large numbers of relatively simple cores
  - high memory bandwidth
  - separate memory from CPU
- Much of current interest is focussed on GPGPUs (general purpose graphics processing units)
  - low cost due to high mass market volumes
  - tricky to program
  - significant overheads in moving data to/from GPU memory
- Also Xeon Phi
  - more like conventional CPUs but with lots of low-power cores, SMT, wide SIMD units
- May not be a long term solution
  - Industry trends suggest a tighter integration of simple cores onto a single piece of silicon

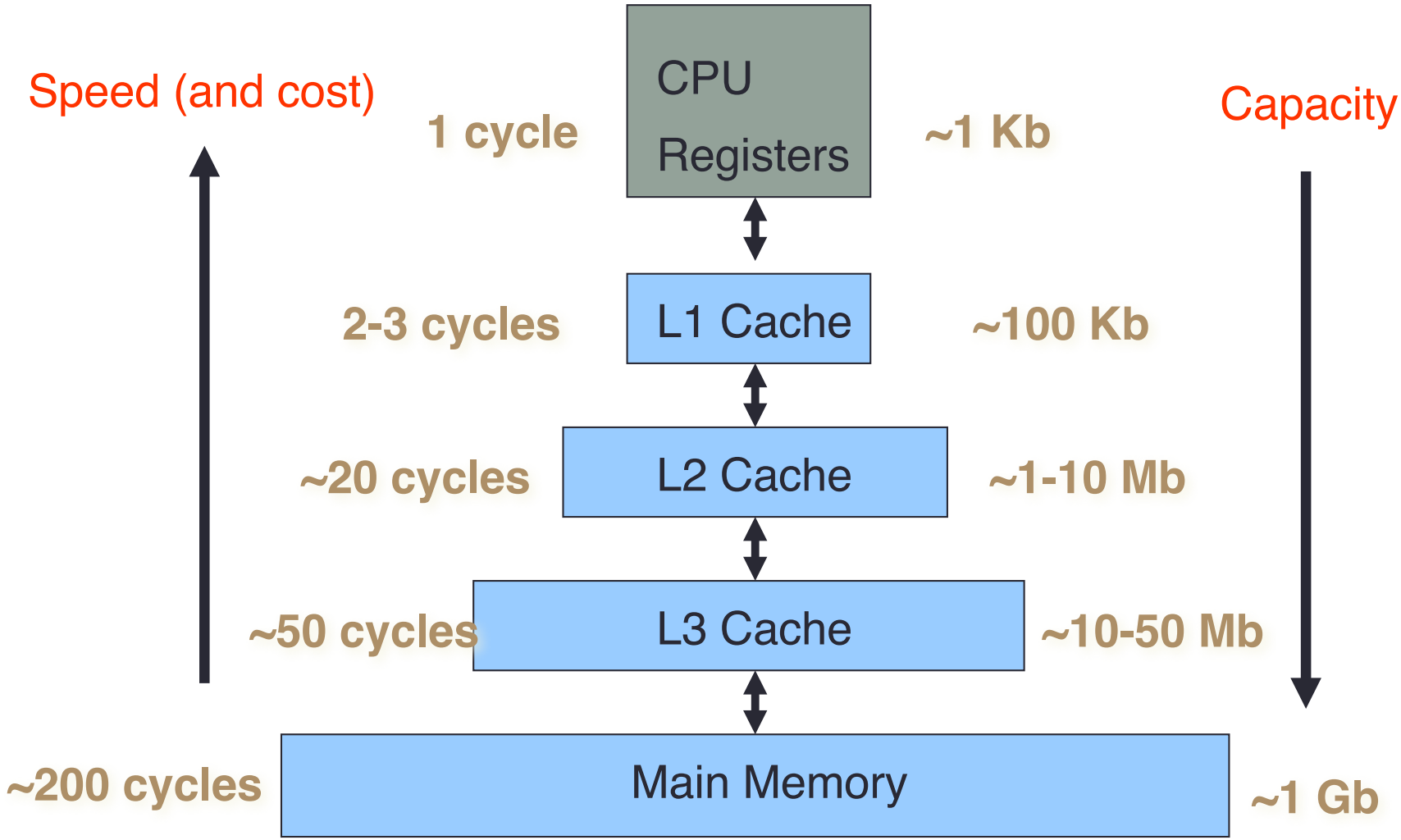
Memory

- Memory speed is often the limiting factor for HPC applications
  - keeping the CPU fed with data is the key to performance
- Memory is a substantial contributor to the cost of systems
  - typical HPC systems have a few Gbytes of memory per processor
  - technically possible to have much more than this, but it is too expensive and power-hungry
- Key metrics
  - latency: how long you have to wait for data to arrive
  - bandwidth: how fast it actually comes in
  - ballpark figures: 100's of nanoseconds and a few Gbytes/s

Cache memory

- Memory latencies are very long
  - 100s of processor cycles
  - fetching data from main memory is 2 orders of magnitude slower than doing arithmetic
- Solution: introduce cache memory
  - much faster than main memory
  - ...but much smaller than main memory
  - keeps copies of recently used data
- Modern systems use a hierarchy of two or three levels of cache
  - - typically all on-chip

Memory hierarchy



Caches

- Sophisticated book-keeping required
  - need to know where most up-to-date copy is kept
  - flush old data downwards to make space for new data
  - done automatically, transparent to the CPU
  - programs still see simple register/memory model
- Caches only help performance if the application re-uses recently accessed data
  - mostly the programmer's responsibility to order computations so as to maximise the re-use
- Many modern memory systems also do prefetching
  - make (quite simple) guesses as to which data will be used next
  - load these data into caches before the processor requests them

Principal of locality

- Almost every program exhibits some degree of locality.
  - Tend to reuse recently accessed data.

- Two types of data locality:

1. Temporal locality

   A recently accessed item is likely to be reused in the near future.

   e.g. if `x` is read now, it is likely to be read again, or written, soon.

2. Spatial locality

   Items with nearby addresses tend to be accessed close together in time.

   e.g. if `y[i]` is read now, `y[i+1]` is likely to be read soon.

How do caches help?

- Cache hold copies of data from main memory locations.
- Cache can hold recently accessed data items for fast re-access.
- Fetching an item from cache is much quicker than fetching from main memory.
  - 1 nanosecond instead of 100.
- For cost and speed reasons, cache is much smaller than main memory.

Blocks

- A cache block is the unit of data which can be transferred from main memory into the cache.
- Normally a few words long: typically 32 to 128 bytes.
- N.B. a block is sometimes also called a line.

When to cache?

- Always cache on reads
  - except in special circumstances

- If a memory location is read and there isn't a copy in the cache (read miss), then cache the data.

- What happens on writes depends on the write strategy: see later.

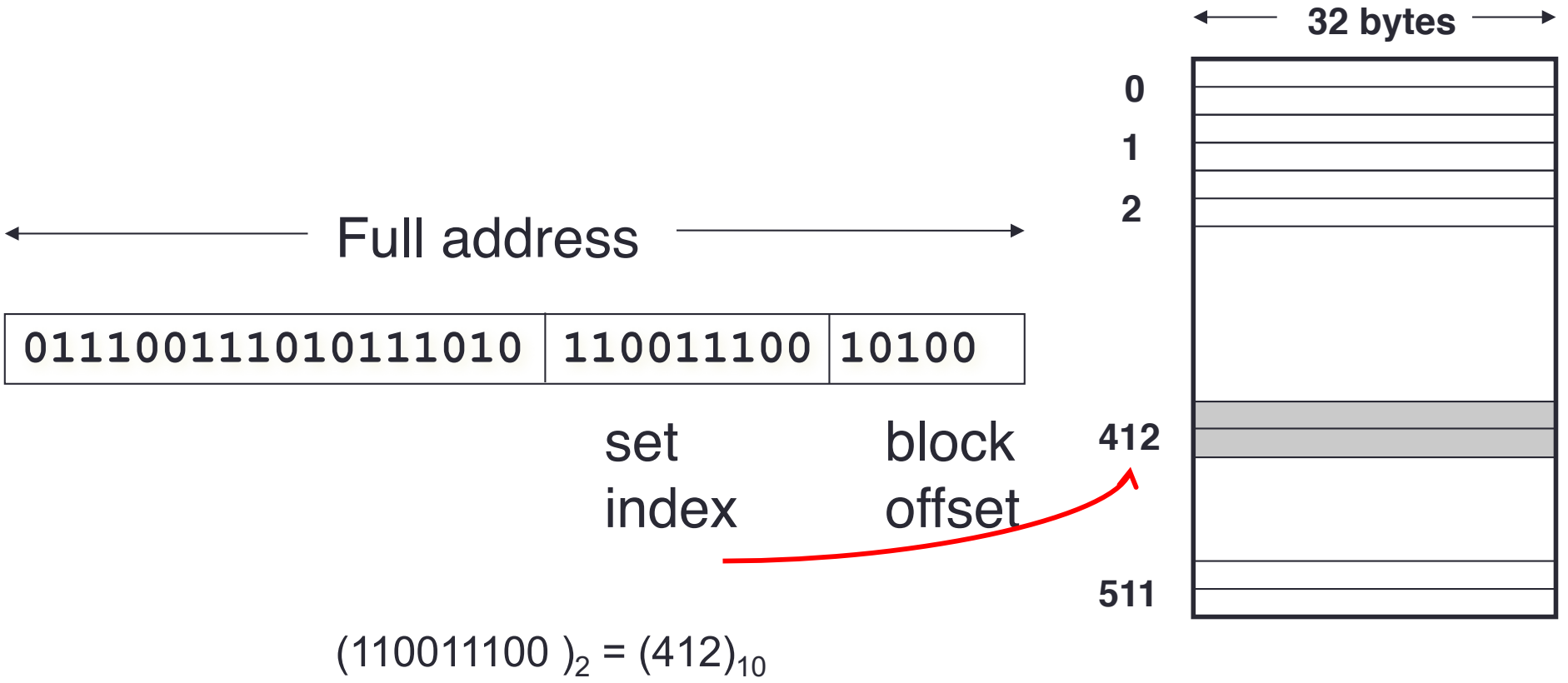
Where does the data go?

- Cache is divided into sets
- A set is a group of blocks (typically 4, 8, 16)
- e.g. 8 blocks per set is an 8-way set associative cache
- If we want to cache the contents of an address, we ignore the last *n* bits where 2^n is the block size.
- Compute set index as:

  (remaining bits) MOD (no. of sets in cache)

  - next *m* bits where 2^m is number of sets.
- Data can go into any block in the set.

32Kbytes cache, 32byte blocks, 2 blocks per set



$(110011100)_2 = (412)_{10}$

Which block to replace?

- Once a set is chosen, must choose a block in the set to store new data.
- Two common strategies:

Random

- Replace a block in the selected set at random.

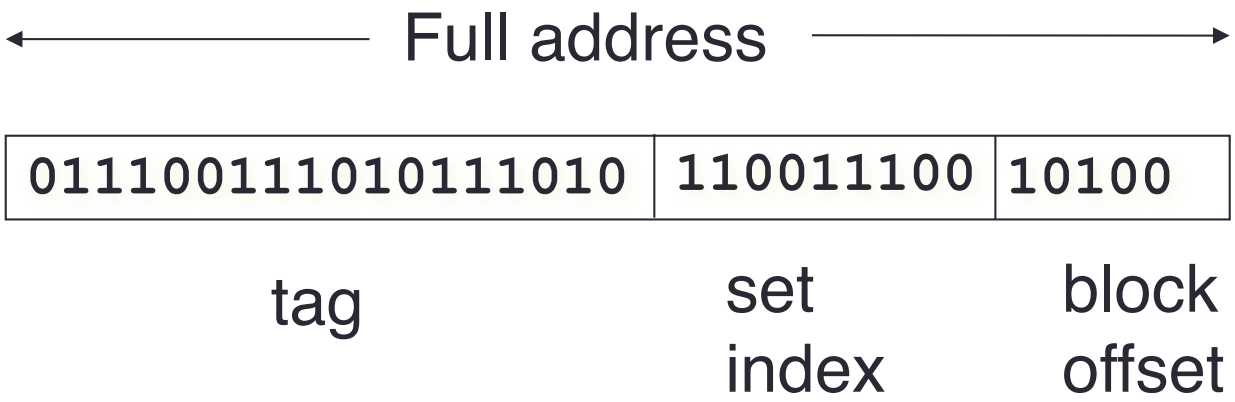
Least recently used (LRU)

- Replace the block in set which was unused for longest time.

- LRU is better, but harder to implement. Some additional sophistication is possible....

How to find a cache block

- Whenever the CPU loads an address, the cache must check whether it has the data.
- For a given address, find set where it might be cached.
- Each block has an address tag.
  - address with the block index and block offset stripped off.
- Each block has a valid bit.
  - if the bit is set, the block contains a valid address
- Need to check tags of all valid blocks in set for target address.



What happens on write?

- Writes are less common than reads.
- Two basic strategies:

Write through

- Write data to cache block and to main memory.
- Normally do not cache on miss.

Write back

- Write data to cache block only. Copy data back to main memory only when block is replaced.
- Dirty/clean bit used to indicate when this is necessary.
- Normally cache on miss.

Write through vs. write back

- With write back, not all writes go to main memory.
  - reduces memory bandwidth.
  - harder to implement than write through.
- With write through, main memory always has valid copy.
  - useful for I/O and for some implementations of multiprocessor cache coherency.
  - can avoid CPU waiting for writes to complete by use of write buffer.

What does the processor see?

- All the machinery in the cache is essentially invisible to the processor.
- Assembly code is scheduled assuming the data is in Level 1 cache.
- If it is not (i.e. a cache miss), the processor has to wait until the data is found and loaded.
- Processor can continue executing instructions
  - but there is a limit on the number of outstanding memory references.
  - after this, the processor will stop and wait

Prefetching

- One way to reduce miss rate is to load data into cache before the load is issued. This is called prefetching

- Processor must be able to support multiple outstanding cache misses.
- Additional hardware is required to keep track of the outstanding prefetches.

- Number of outstanding misses is limited (e.g. 4 or 8): extra benefit from allowing more does not justify the hardware cost.

- Hardware prefetching is typically very simple: e.g. whenever a block is loaded, fetch consecutive block.
  - very effective for instruction cache
  - less so for data caches, but can have multiple streams.
  - requires regular data access patterns.
- Compiler can place prefetch instructions ahead of loads.
  - requires extensions to the instruction set
  - cost in additional instructions.
  - no use if placed too far ahead: prefetched block may be replaced before it is used.

Multiple levels of cache

- Second level cache should be much larger than first level.
  - otherwise a level 1 miss will almost always be level 2 miss as well.
- Second level cache will therefore be slower
  - still much faster than main memory.
- Typically, everything in level 1 must be in level 2 as well (inclusion)
  - required for cache coherency in multiprocessor systems.
- Three levels of cache are now commonplace.
  - All 3 levels now on chip

Virtual memory

- Allows memory and disk to be seamless whole
  - processes can use more data that will fit into physical memory
- Allows multiple processes to share physical memory
- Can think of main memory as a cache of what is on disk
  - blocks are called pages (4 to 64 kbytes)
  - a miss is called a page fault

Virtual memory

- CPU issues virtual addresses which are translated to physical addresses.
- Pages can be placed anywhere in memory
  - like a fully associative cache
  - approximate LRU replacement strategy
  - write back, not write through
- Mapping from virtual to physical address is stored in a page table in memory
- Page table lookup is relatively expensive
- Page faults are very expensive
  - requires system call (~1ms)

Page table

- Page table stores virtual to physical address mapping.
  - can be hierarchical (table of tables of tables of pages)
- Look-up table for every virtual page is not feasible (especially for 64-bit virtual address space)
- Use inverted page table instead
  - store virtual address for each physical page
- Page table lookup requires a search
  - can be made efficient by use of hash-table
- Replacement policy
  - typically random or FIFO (but prefer unused/unmodified pages)

TLB

- Translation look-aside buffer (TLB) is name given to a hardware cache of virtual to physical mappings.
- Can store the mapping for a limited number of pages
  - TLB miss results in a page table lookup
  - generally has fewer blocks than level 1 cache (so that TLB lookup is not on critical path)
- TLB relies on locality
  - widely scattered memory accesses can result in lots of TLB misses
  - can be as important as cache misses in some applications
  - some systems offer a choice of page sizes: larger pages => fewer TLB misses

Cache coherency

- Main difficulty in building multiprocessor systems is the cache coherency problem.
- The shared memory programming model assumes that a shared variable has a unique value at a given time.
- Caching in a shared memory system means that multiple copies of a memory location may exist in the hardware.
- To avoid two processors caching different values of the same memory location, caches must be kept *coherent.*
- To achieve this, a write to a memory location must cause all other copies of this location to be removed from the caches they are in.

Coherence protocols

- Need to store information about sharing status of cache blocks
  - has this block been modified?
  - is this block stored in more than one cache?

- Two main types of protocol

1. Snooping (or broadcast) based
   - every cached copy caries sharing status
   - no central status
   - all processors can see every request
2. Directory based
   - sharing status stored centrally (in a directory)

Snoopy protocols

- Already have a valid tag on cache lines: this can be used for invalidation.
- Need an extra tag to indicate sharing status.
  - can use clean/dirty bit in write-back caches
- All processors monitor all bus transactions
  - if an invalidation message is on the bus, check to see if the block is cached, and if so invalidate it
  - if a memory read request is on the bus, check to see if the block is cached, and if so return data and cancel memory request.
- Many different possible implementations

3 state snoopy protocol: MSI

- Simplest protocol which allows multiple copies to exist
- Each cache block can exist in one of three states:
  - ***M****odified*: this is the only valid copy in any cache and its value is different from that in memory
  - ***S****hared*: this is a valid copy, but other caches may also contain it, and its value is the same as in memory
  - ***I****nvalid*: this copy is out of date and cannot be used.
- Model can be described by a state transition diagram.
  - state transitions can occur due to actions by the processor, or by the bus.
  - state transitions may trigger actions

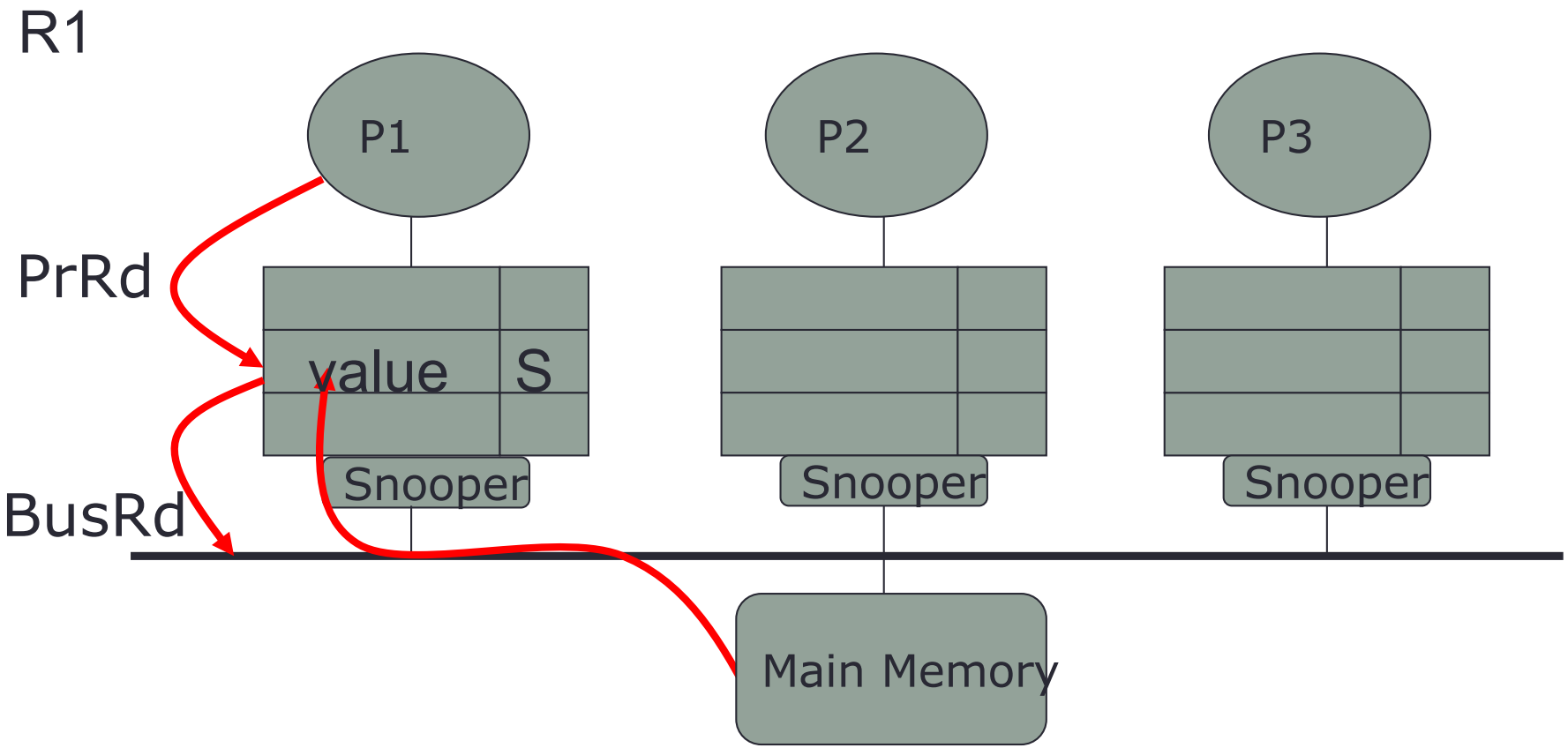
Processor actions
- read (PrRd)
- write (PrWr)

Bus actions
- read (BusRd)
- read exclusive (BusRdX)
- flush to memory (Flush)

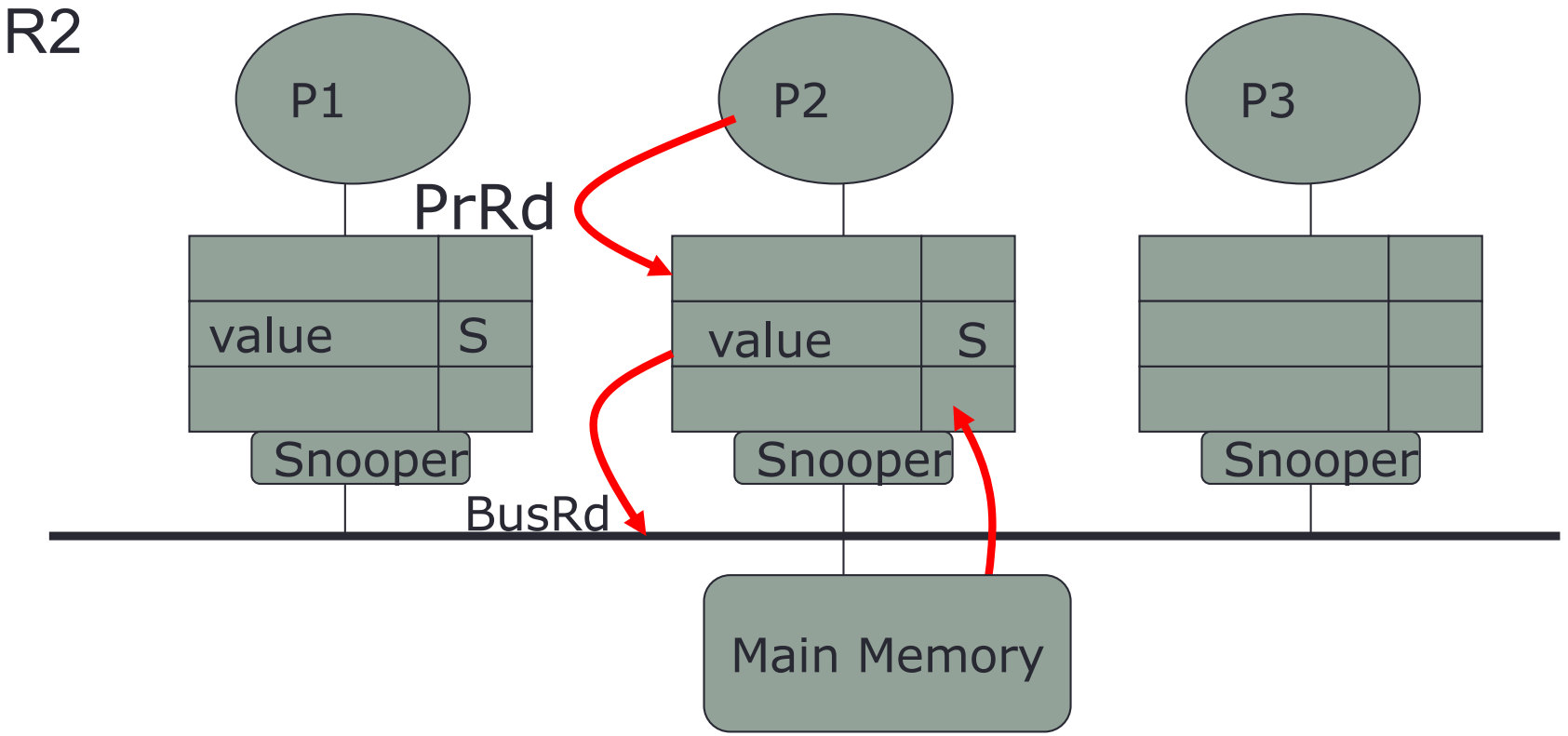
MSI Protocol walk through

- Assume we have three processors.
- Each is reading/writing the same value from memory where R1 means a read by processor 1 and W3 means a write by processor 3.
- For simplicity sake, the memory location will be referred to as “value.”
- The memory access stream we will walk through is:

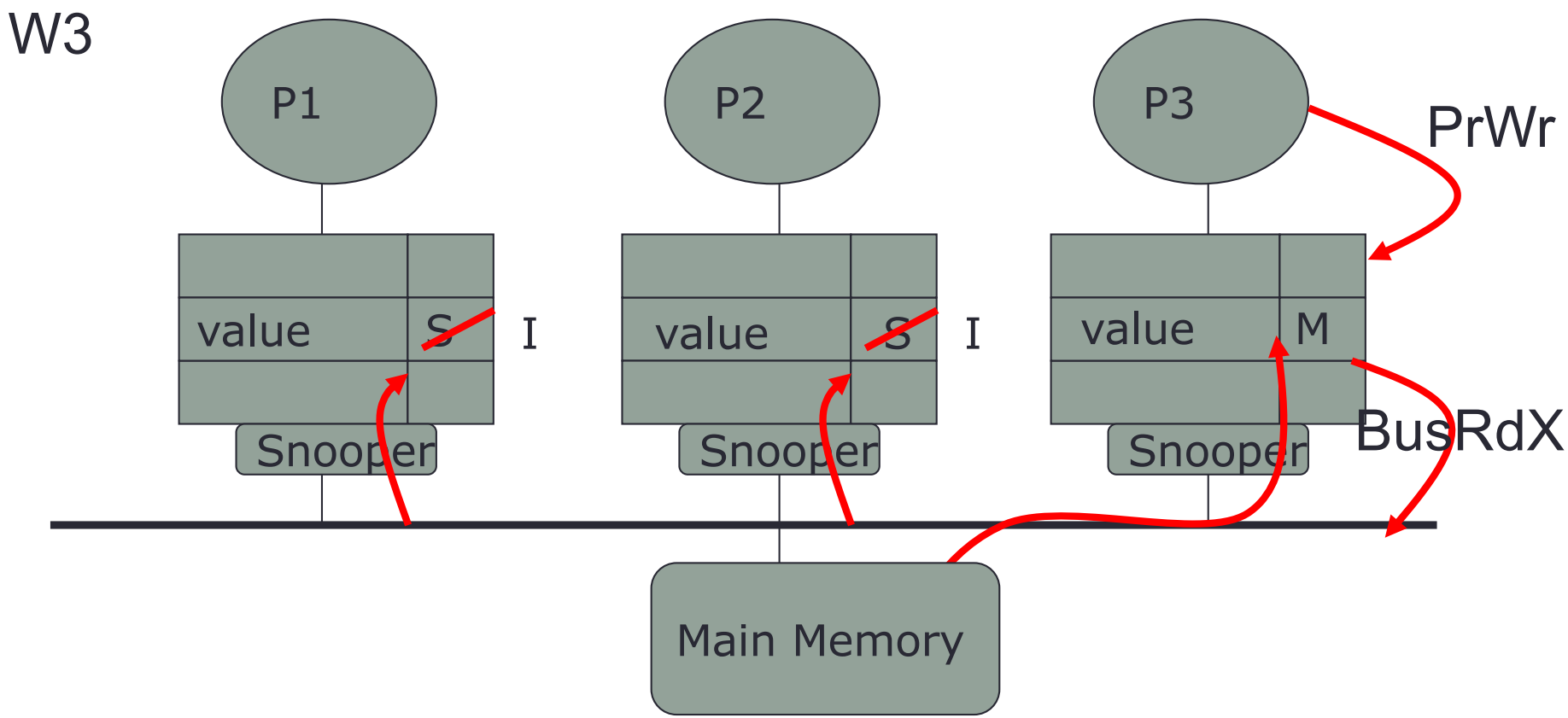
R1, R2, W3, R2, W1



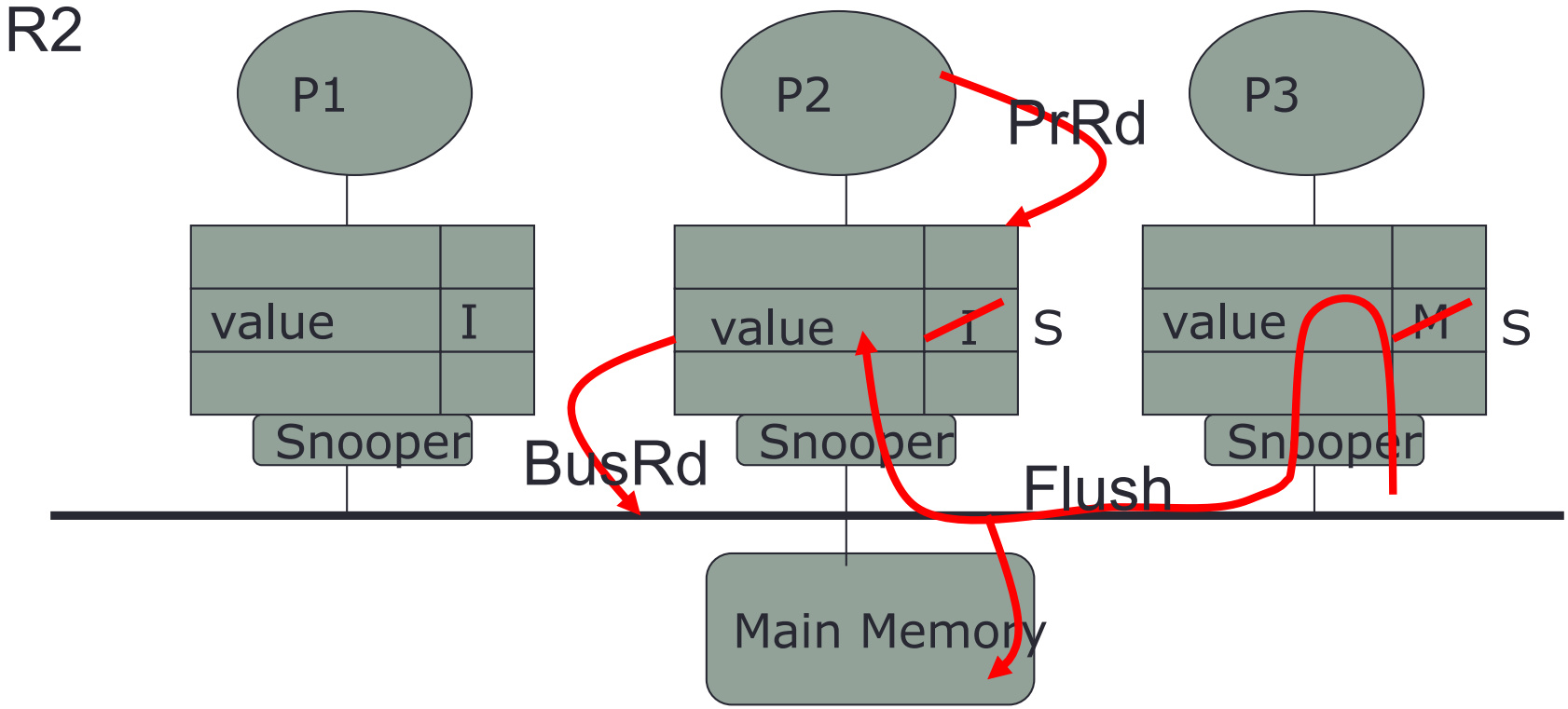
P1 wants to read the value. The cache does not have it and generates a BusRd for the data. Main memory controller provides the data. The data goes into the cache in the shared state.



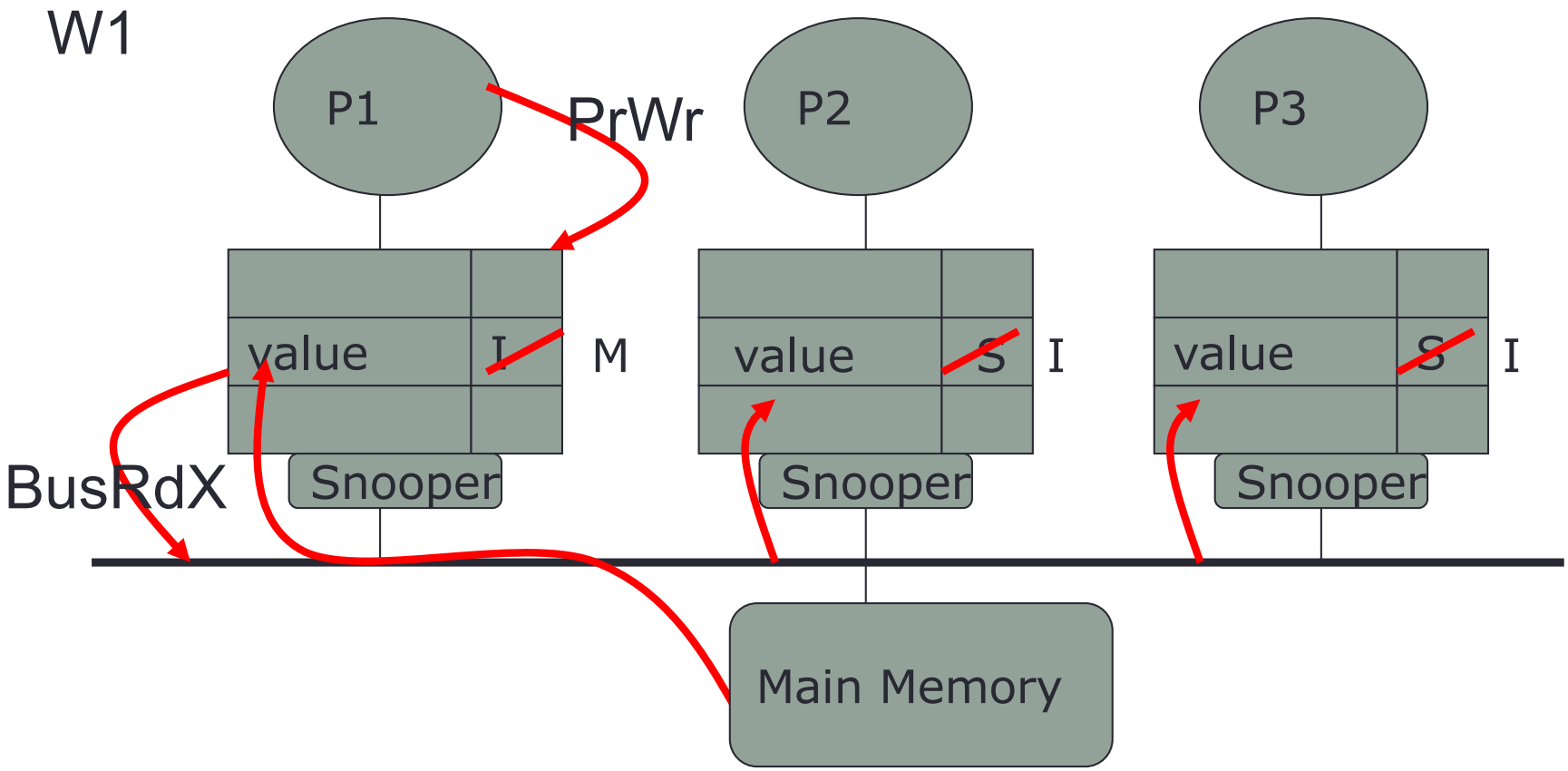
P2 wants to read the value. Its cache does not have the data, so it places a BusRd to notify other processors and ask for the data. The memory controller provides the data.



P3 wants to write the value. It places a BusRdX to get exclusive access and the most recent copy of the data. The caches of P1 and P2 see the BusRdX and invalidate their copies. Because the value is still up-to-date in memory, memory provides the data.



P2 wants to read the value. P3's cache has the most up-to-date copy and will provide it. P2's cache puts a BusRd on the bus. P3's cache snoops this and cancels the memory access because it will provide the data. P3's cache flushes the data to the bus.



P1 wants to write to its cache. The cache places a BusRdX on the bus to gain exclusive access and the most up-to-date value. Main memory is not stale so it provides the data. The snoopers for P2 and P3 see the BusRdX and invalidate their copies in cache.

Other protocols

- MSI is inefficient: it generates more bus traffic than is necessary
- Can be improved by adding other states, e.g.
  - ***E****xclusive*: this copy has not been modified, but it is the only copy in any cache
  - ***O****wned*: this copy has been modified, but there may be other copies in shared state
- MESI and MOESI protocols are more commonly used protocols than MSI
- MSI is nevertheless a useful mental model for the programmer

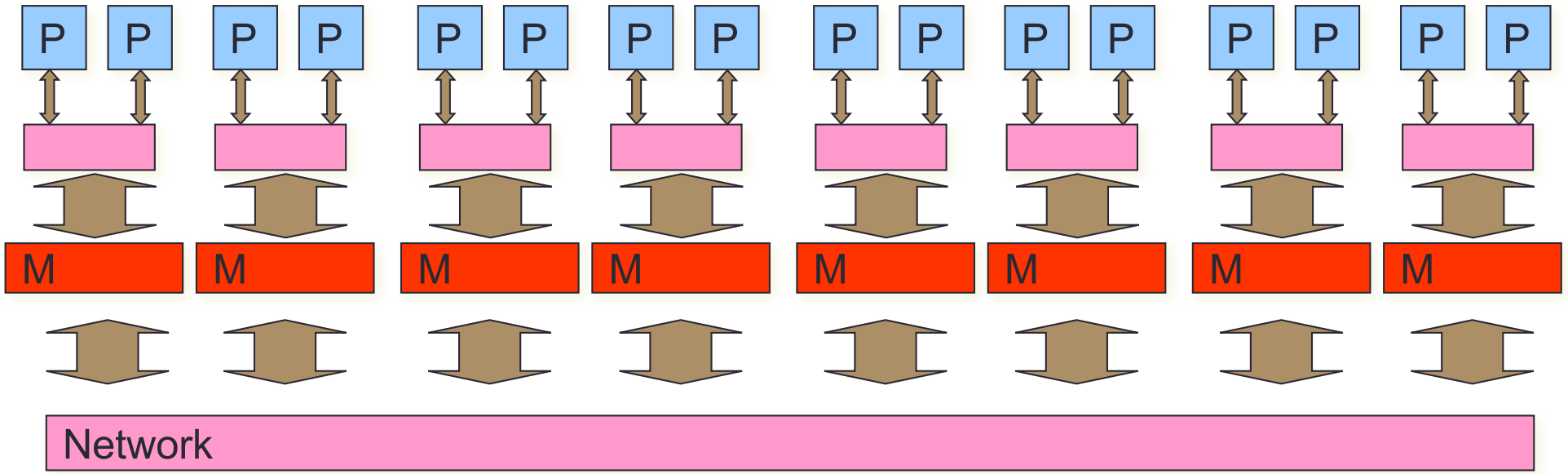
False sharing

- The units of data on which coherency operations are performed are cache blocks: the size of these units is usually 64 or 128 bytes.
- The fact that coherency units consist of multiple words of data gives rise to the phenomenon of false sharing.
- Consider what happens when two processors are both writing to different words on the same cache line.
  - no data values are actually being shared by the processors
- Each write will invalidate the copy in the other processor's cache, causing a lot of bus traffic and memory accesses.
  - same problem if one processor is writing and the other reading
- Can be a significant performance problem in threaded programs (but not for message passing)
- Quite difficult to detect

Distributed shared memory

- Shared memory machines using buses and a single main memory do not scale to large numbers of processors
  - bus and memory become a bottleneck
- Distributed shared memory machines designed to:
  - scale to larger numbers of processors
  - retain a single address space
- Modest sized multi-socket systems connected with HyperTransport or QPI are, in fact, distributed shared memory

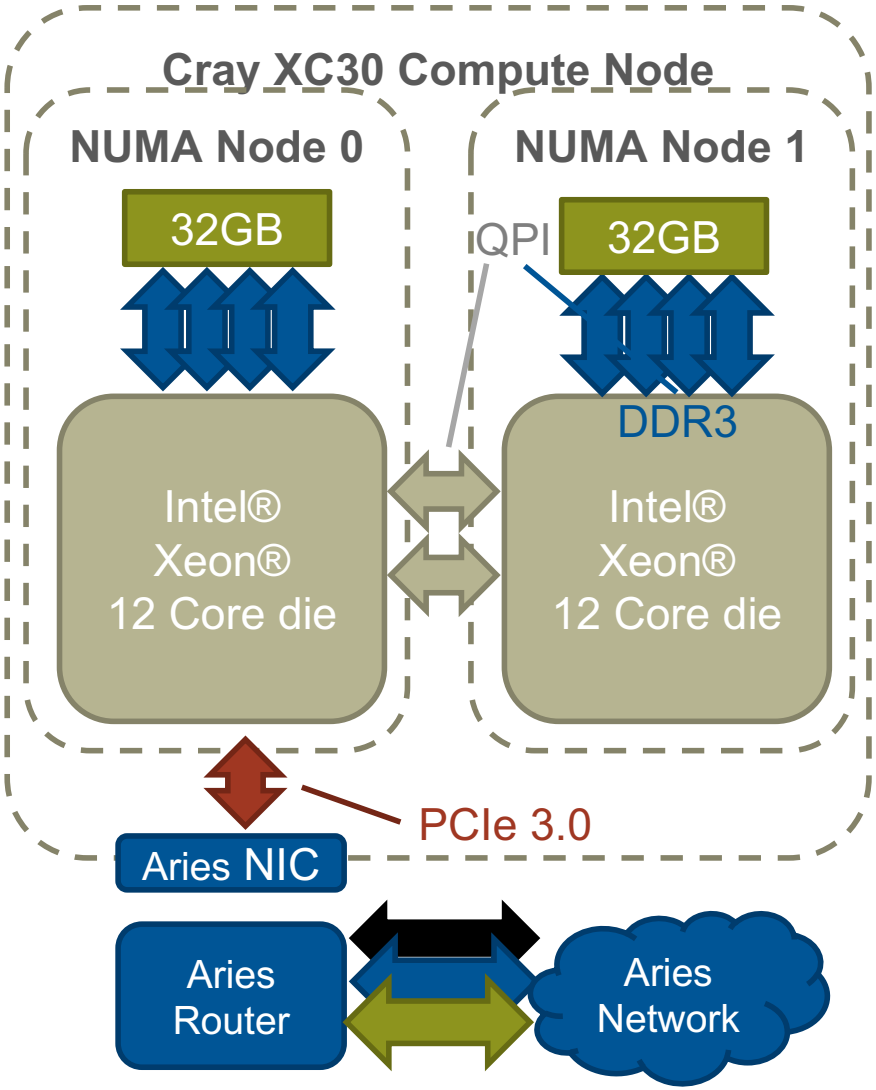
Distributed shared memory



cc-NUMA

- In most distributed shared memory systems every memory address is allocated to a fixed location (its *home node*).
- This type of system is known a a cache-coherent non-uniform memory architecture (cc-NUMA).
- Main problem is that access to remote memories take longer than to local memory
  - difficult to determine which is the best node to allocate given page on
- OS is responsible for allocating pages
- Common policies are:
  - first touch: allocate on node which makes first access to the page
  - round robin: allocate cyclically

Cray XC30 Intel® Xeon® Compute Node



The XC30 Compute node features:

- 2 x Intel® Xeon® Sockets/die
  - 12 core Ivy Bridge
  - QPI interconnect
  - Forms 2 NUMA nodes
- 8 x 1833MHz DDR3
  - 8 GB per Channel
  - 64/128 GB total
- 1 x Aries NIC
  - Connects to shared Aries router and wider network
  - PCI-e 3.0

epcc

THE UNIVERSITY OF EDINBURGH

Intel Ivy Bridge Processor

- Proper name: Xeon E5-2697v2
- No. of cores = 12
- Clock rate = 2.7 GHz
- 1 FP adder, 1 FP multiply (no FMA)
- 256-bit wide AVX vector instructions
  - 4 double precision floating point ops
- Peak 8 flops per clock = 21.6 Gflop/s per core = 259 Gflop/s per socket = 518 Gflop/s per node
- Up to two hardware threads (hyperthreads) per core

Memory hierarchy

- Each core has
  - Level 1 cache: 32 KB, 8-way set associative, 64 byte lines
  - Level 2 cache: 256 KB, 8-way set associative, 64 byte lines
- All 12 cores share one Level 3 cache
  - 30 MB, 16-way set associative, 64 byte lines
  - 2.5MB per core
- 64 GB main memory per node (128 GB on high memory nodes) in 2 NUMA regions (1 per socket)
- Bandwidths per core L1/L2/L3/Mem = 100/40/20/4 GB/s
- Memory latency = ~80ns (~210 clock cycles)