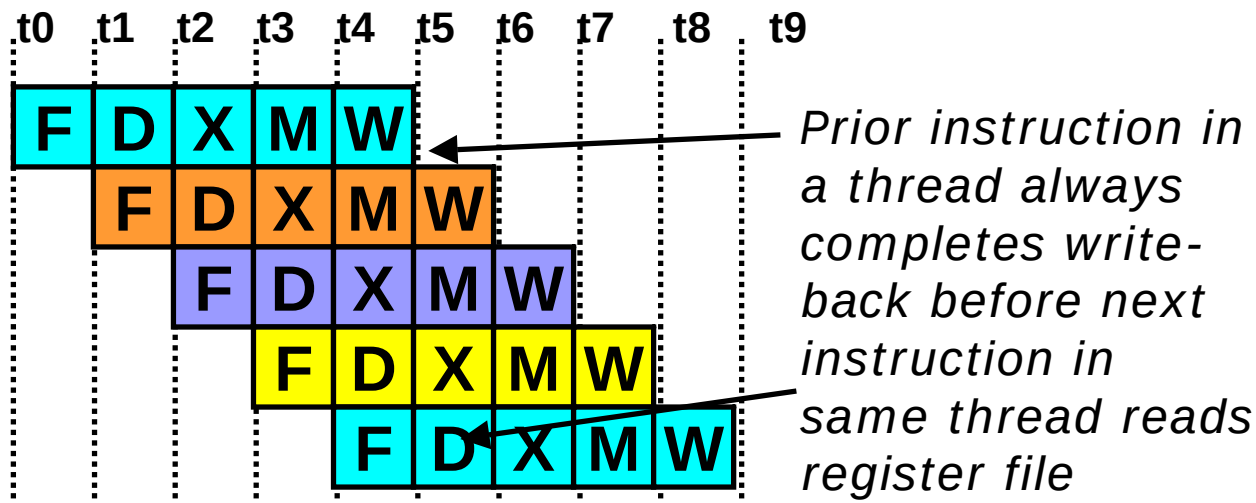
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

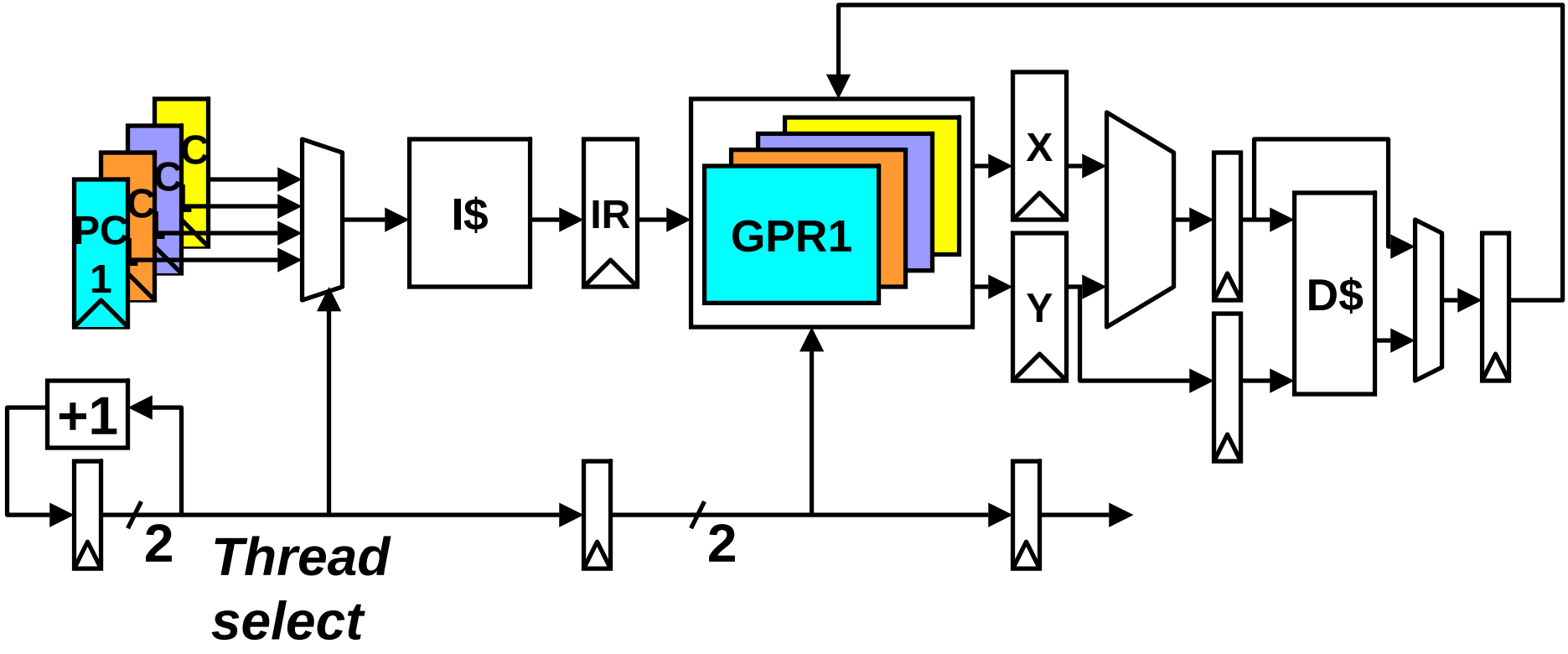
-- One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)



Simple Multithreaded Pipeline



Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage

Multithreading Costs

- Each thread requires its own user state
  - PC
  - GPRs
- Also, needs its own system state
  - virtual memory page table base register
  - exception handling registers

Thread Scheduling Policies

- Fixed interleave *(CDC 6600 PPUs, 1965)*
  - each of N threads executes one instruction every N cycles
  - if thread not ready to go in its slot, insert pipeline bubble
- Software-controlled interleave *(TI ASC PPUs, 1971)*
  - OS allocates S pipeline slots amongst N threads
  - hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- Hardware-controlled thread scheduling *(HEP, 1982)*
  - hardware keeps track of which threads are ready to go
  - picks next thread to execute based on hardware priority scheme

Pentium-4 Hyperthreading (2002)

- First commercial SMT design (2-way SMT)
  - Hyperthreading == SMT
- Logical processors share nearly all resources of the physical processor
  - Caches, execution units, branch predictors
- Die area overhead of hyperthreading ~ 5%
- When one logical processor is stalled, the other can make progress
  - No logical processor can use all entries in queues when two threads are active
- Processor running only one active software thread runs at approximately same speed with or without hyperthreading

O-o-O Simultaneous Multithreading
[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

- Add multiple contexts and fetch engines and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

Why Parallel Processing?

parallel processors: also called *multiprocessors* (*MPs*)

- multiple processors working together, why?
- performance: break physical limits of uniprocessing
  - ILP (branch prediction, RAW dependences, etc.)
  - speed of light
- cost and cost effectiveness
  - build big systems from *commodity parts* (ordinary uniprocessors)
  - the commodity part of the future (no more uniprocessors)
- other
  - smooth upgrade path (keep adding processors)
  - fault tolerance (one processor fails, still have P-1 working)

Parallel Processing Is Hard

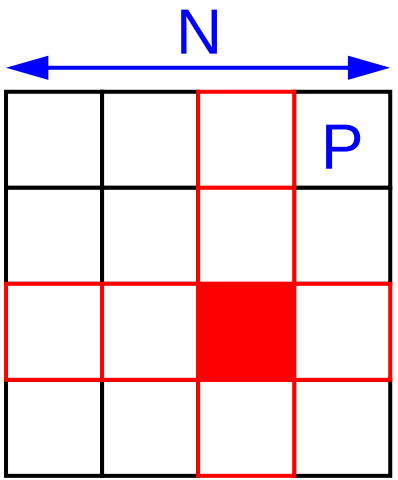
in a word: *software*

- difficult to parallelize applications
  - compiler parallelization hard (have already seen this with vectors)
  - by-hand parallelization maybe harder (very error prone)
- difficult to make parallel applications run fast
  - communication very expensive (must be aware of it)

Application Domain 1: Parallel Programs

- true parallelism in one job
  - regular loop structures
  - data usually tightly shared
  - automatic parallelization
  - called "*data-level parallelism*"
  - can often exploit vectors as well (have seen)
- workloads
  - scientific simulation codes (e.g., FFT, weather, fluid dynamics, etc.)
  - was the dominant market segment of 10–15 years ago

Parallel Program Example: Matrix Multiply



- parameters
  - N = size of matrix (N*N)
  - P = number of processors
- growth functions
  - computation grows as f(N^3)
  - computation per processor grows as f(N^3/P)
  - data size grows as f(N^2)
  - data size per processor grows as f(N^2/P)
  - *communication* grows as f($N^2/P^{1/2}$)
  - computation/communication = f($N/P^{1/2}$)

Application Domain 2: Parallel Tasks

- parallel independent-but-similar tasks
  - irregular control structures
  - loosely shared data locked at different granularities
  - programmer defines & fine-tunes parallelism
  - cannot exploit vectors
  - called "*thread-level parallelism*" or "*thruput-oriented parallelism*"
- workload
  - transaction processing, OS, databases, web-servers
  - dominant MP market segment TODAY (by far)

Taxonomy of Processors

Flynn Taxonomy [1966]

- not all encompassing but simple
- dimensions
  - instruction streams: single (SI) or multiple (MI)
  - data streams: single (SD) or multiple (MD)
- cross-product
  - SISD: uniprocessor (been there)
  - SIMD: vectors (done that)
  - MISD: no practical examples (won’t do that)
  - *MIMD*: multiprocessors + multicomputers (doing it now)

SIMD vs. MIMD

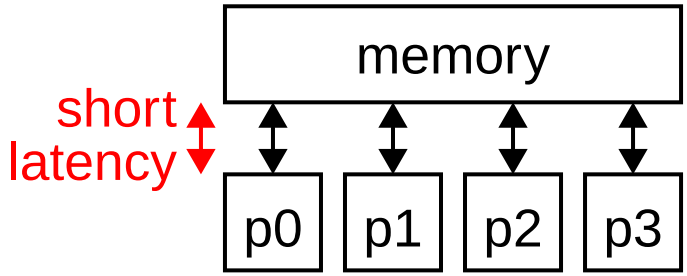
why are MPs (much) more common than vector processors?

- programming model flexibility
  - can simulate vectors with an MP, but not the other way around
  - dominant market segment cannot exploit vectors
- cost effectiveness
  - *commodity part*: high volume (translation: cheap) component
  - MPs made up of commodity parts (i.e., uniprocessors)
  - can match size of MP to your budget
  - can't do this for a vector processor
- footnote: vectors are making a comeback
  - for graphics/Multimedia applications

Taxonomy of Parallel (MIMD) Processors

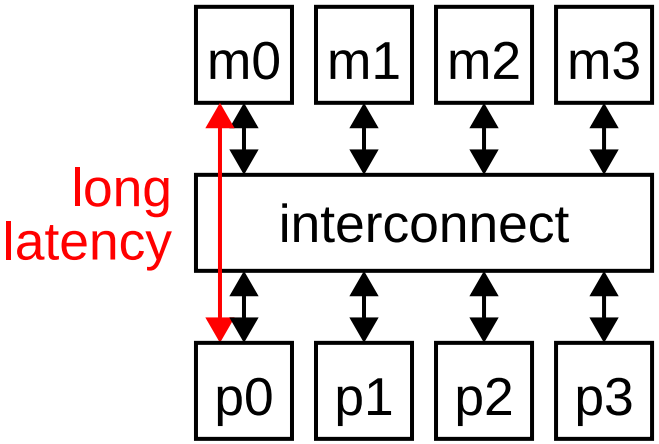
- again, two dimensions
  - center on organization of main memory (shared vs. distributed)
- dimension I: appearance of memory to *hardware*
  - Q: is access to all memory uniform in latency?
  - *shared (UMA)*: yes ⇒ where you put data doesn't matter
  - *distributed (NUMA)*: no ⇒ where you put data really matters
- dimension II: appearance of memory to *software*
  - Q: can processors communicate via memory directly?
  - *shared (shared memory)*: yes ⇒ communicate via loads/stores
  - *distributed (message passing)*: no ⇒ communicate via messages
- dimensions are orthogonal
  - e.g., DSM: (physically) distributed (logically) shared memory

UMA vs. NUMA: The Setup

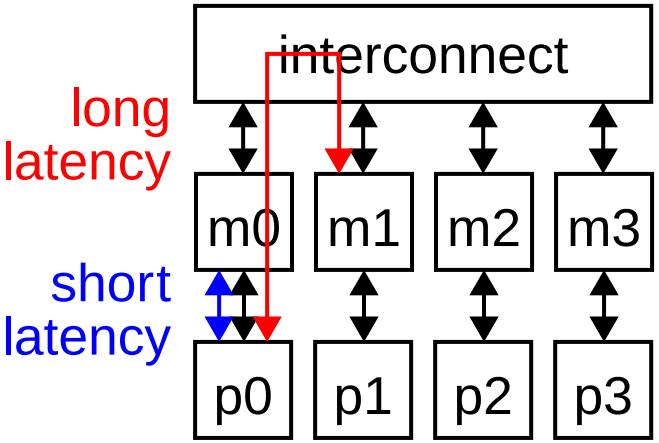


- *PRAM (parallel RAM)*: ideal theoretical model
  - perfect (single-cycle) memory latency
  - perfect (infinite) memory bandwidth
  - – not achievable
- in the real world...
  - – latencies are long *and* grow with system size
  - – bandwidth is limited
  - to get bandwidth ⇒ split memory into banks, add interconnect
  - – interconnect adds even more latency

UMA vs. NUMA



- *UMA: uniform memory access*
  - from p0 same latency to m0 as to m3
  - \+ data placement unimportant (software is easier)
  - – latency long, gets worse as system grows
  - – interconnect contention restricts bandwidth
  - typically used in small multiprocessors only

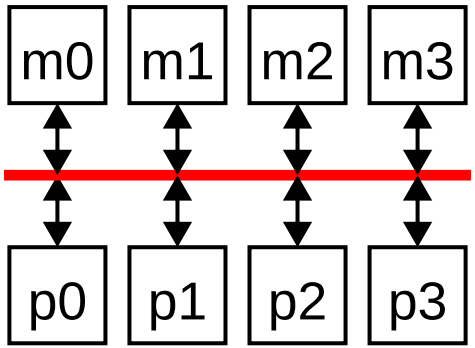


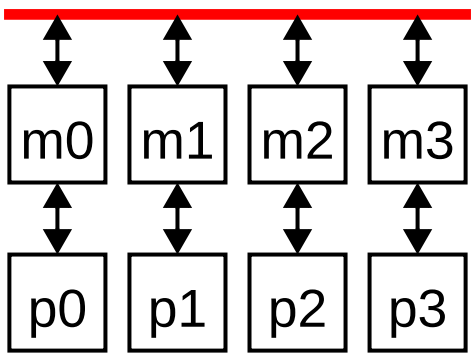
- *NUMA: non-uniform memory accesss*
  - from p0 faster to m0 (local) than m3 (non-local)
  - \+ low latency to local memory helps performance
  - – data placement important (software is harder)
  - \+ less contention (non-local only) ⇒ more scalable
  - typically used in larger multiprocessors

Interlude: What Is “Interconnect”?

- connects processors/memories to each other
  - *direct*: endpoints connected directly
  - *indirect*: endpoints connected via switches/routers
- interconnect issues
  - *latency*: average latency most important (locality optimizations?)
  - *bandwidth*: per processor
  - *cost*: # wires, # switches, # ports per switch
  - *scalability*: how latency, bandwidth, cost grow with # processors (P)
- mainly concerned with *interconnect topology*
- can have separate interconnects for addresses and data

Interconnect 1: Bus





- direct interconnect

+ cost
  - f(1) wires

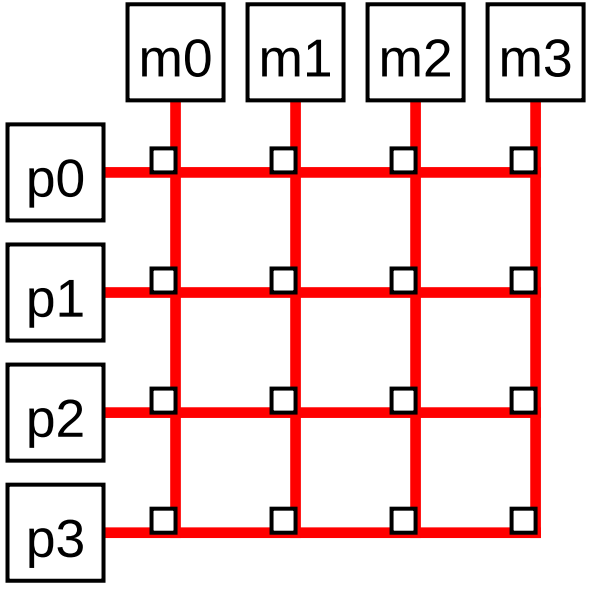
+ latency: f(1)
  - no neighbor/locality optimization

– bandwidth: *not scalable at all*, f(1/P)
  - only used in small systems (P <= 4)

+ other: capable of *ordered broadcast*
  - incapable of anything else

what about hierarchical busses?

Interconnect 2: Crossbar Switch



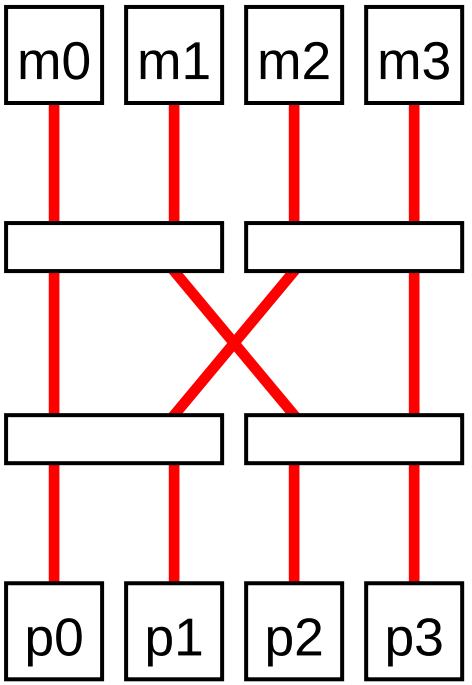
- indirect interconnect

+ latency: f(1)
  - no locality/neighbor optimizations

+ bandwidth: f(1)

– cost
  - f(2P) wires
  - f(P^2) switches
  - 4 wires per switch

Interconnect 3: Multistage Network



- indirect interconnect
  - routing done by address bit decoding
  - *k:* switch arity (# inputs and outputs per switch)
  - *d*: number of network stages = $\log_k P$

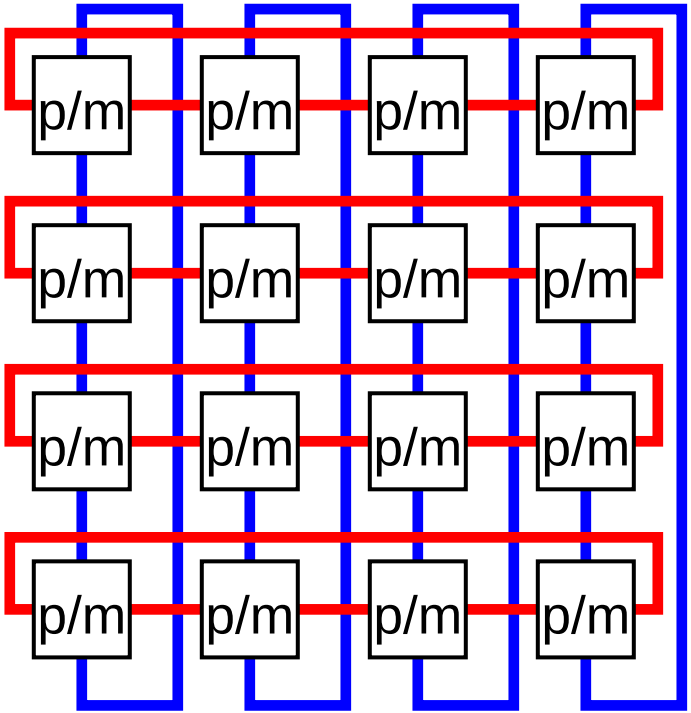
+ cost
  - f(d*P/k) switches
  - f(P*d) wires
  - f(k) wires per switch

+ latency: f(d)

+ bandwidth: f(1)

- commonly used in large UMA systems
  - a.k.a. butterfly, banyan, omega

Interconnect 4: 2D Torus



- direct interconnect
  - no dedicated switches

+ latency: $f(P^{1/2})$
  - locality/neighbor optimization

+ bandwidth: f(1)

+ cost
  - f(2P) wires
  - 4 wires per switch

- good scalability ⇒ widely used
  - variants: 3D, mesh (no “wraparound”)

Interconnect Routing

- *store-and-forward* routing
  - switch buffers entire message before passing it on
  - latency = [(message length / bandwidth) + fixed overhead] * # hops
- *wormhole* routing
  - pipeline message through interconnect
  - switch passes message on before completely arrives
  - latency = (message length / bandwidth) + (fixed overhead * # hops)

  + no buffering needed at switch

  + latency (relatively) independent of number of intermediate hops

- separate issue: dimension-order routing
  - route along dimensions in fixed order to avoid deadlocks

Shared Memory vs. Message Passing

MIMD dimension II: appearance of address space to software

- *message passing* (multicomputers)
  - each processor has its own address space (and unique processor #)
  - processors send (receive) messages to (from) each other
  - communication pattern explicit and precise (only way)
  - used for scientific codes (explicit communication patterns)
  - message passing systems: PVM, MPI

  \+ simple hardware

  – difficult programming model (in general)

Shared Memory vs. Message Passing

- *shared memory* (multiprocessors)
  - one shared address space
  - processors use conventional loads/stores to access shared data
  - communication can be complex/dynamic

  \+ simpler programming model (compatible with uniprocessors)

  – but with its own nasties (e.g., synchronization)

  – more complex hardware... (we'll see soon)

  \+ but more room for hardware optimization

Two Issues for Shared Memory Systems

- actually three issues
  - cache coherence
  - synchronization
  - memory consistency model
- not completely unrelated to each other
- not issues for message passing machines

Cache (In)Coherence

- most common cause: sharing of writeable data
  - example

processor 0	processor 1	correct value of A in..
		memory
read A		memory, p0 cache
	read A	memory, p0 cache, p1 cache
write A		p0 cache, memory (if wthru)
	read A	p1 gets stale value on hit

- other causes
  - process migration (even if jobs are independent)
  - I/O (can be fixed by OS cache flushes)

Solutions to Coherence Problem

- no caches
  - yeah, right
- make shared-data non-cacheable
  - + simplest software solution
  - – low performance if a lot of data is shared
- software flush at strategic times: e.g., after critical sections
  - + relatively simple
  - – low performance if synchronization is frequent
- hardware *cache coherence*
  - make memory and caches coherent (consistent) with each other
  - in other words: let memory and other processors see writes

Coherence Protocols

- absolute coherence
  - all copies of each block have same data at all times
  - not necessary
- what is required is *appearance of absolute coherence*
  - temporary incoherence is OK (e.g., write-back cache)
  - as long as all loads get correct values
- *coherence protocol*: FSM that runs at every cache
- two kinds of protocols
  - *invalidate protocol*: invalidate copies in other caches
  - *update protocol*: update copies in other caches
  - memory is always updated

Bus-Based Protocols (Snooping)

- bus-based protocol (snooping)
  - ALL caches see and react to ALL bus events
  - protocol relies on global visibility of events (ordered broadcast)
- 3 processor events (i.e., events from own processor)
  - read (R)
  - write (W)
  - writeback (WB)
- 2 bus events (i.e., events from other processors)
  - bus read (BR)
  - bus write (BW)

Chip Multiprocessors

trend today: multiprocessors on a single chip (CMPs)

- can’t spend all of the transistors on just one processor
  - with limited ILP, single processor would not exploit it
- e.g., IBM POWER4
  - 1 chip contains: 2 1Ghz processors, L2, L3 tags, interconnect
  - can connect 4 chips on 1 MCM to create 8 processor system
  - targets threaded server workloads)