

1 Multithreading

Even with superscalar and out-of-order execution, we can still run into two problems:

- **We run out of independent instructions.** For example, a long chain of RAW dependences fills the ROB and stalls the CPU.
- **The CPU is idle most of the time.** Memory and I/O latency dwarf the time spent computing.

This leads to two kinds of waste in a superscalar's issue slots:

- **Vertical waste:** no instructions issue in a cycle (a completely idle cycle).
- **Horizontal waste:** some, but not all, issue slots are filled in a cycle (for example, $IPC < 4$ on a 4-wide machine).

1.1 What are some examples of vertical waste?

Cache misses, long functional-unit latencies, control hazards, branch mispredictions, and TLB misses. Each of these can leave the machine with nothing at all to issue.

1.2 What are some examples of horizontal waste?

Limited instruction-level parallelism, such as RAW dependence chains: some instructions are ready, but not enough to fill every issue slot. Structural limits (too few functional units of the type the ready instructions need) and fetch limits (a taken branch ending the fetch group partway through) also leave slots empty.

Multithreading switches between threads (programs) to reduce waste. In the classic five-stage pipeline below, a different thread fetches an instruction each cycle, and each thread has its own registers (x1 in T1 is not x1 in T2):

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9
T1: LD x1, 0(x2)	F	D	X	M	W					
T2: ADD x7, x1, x4		F	D	X	M	W				
T3: XORI x5, x4, 12			F	D	X	M	W			
T4: SD 0(x7), x5				F	D	X	M	W		
T1: LD x5, 12(x1)					F	D	X	M	W	

Hardware cost. Each hardware thread needs its own copy of the architectural state: PC, general-purpose registers, page table base register, and exception-handling state. A thread-select counter picks which thread's PC to fetch from and which register file to use. The machine above is exposed to software as 4 CPUs.

Important: for the purposes of multithreading, it is fine to treat a hardware thread as a separate program. This is *not* true later in the class (synchronization, etc.), where one program can have multiple threads.

Multithreading strategies. Back to superscalars:

- **Fine-grained multithreading:** switch between threads on every clock cycle.
- **Coarse-grained multithreading:** switch threads only on costly stalls.
- **Multiprocessing:** split one wide processor into multiple narrower cores, each running its own thread.
- **Simultaneous multithreading (SMT):** interleave instructions from multiple threads in multiple issue slots, with no restrictions.

1.3 For each strategy, how is vertical waste reduced? How is horizontal waste reduced? What are its limitations or disadvantages compared to an in-order superscalar RISC machine?

	How is vertical waste reduced?	How is horizontal waste reduced?	Limitations vs. an in-order superscalar
Fine-grained multithreading	By interleaving instructions from multiple threads.	Not reduced compared to superscalar issue: each cycle still issues from only one thread.	Low utilization if there are not enough threads, more architectural state, contention for shared resources, potentially lower single-thread performance.
Coarse-grained multithreading	By switching to another thread on long-latency stalls (e.g., cache misses).	Not reduced.	A switch / flush penalty on each switch, short stalls are not hidden, more architectural state, cache and TLB contention.
Multiprocessing	Not reduced within a core: each core still idles when its thread stalls.	Narrower cores are easier to fill.	Upper limit on each thread's peak throughput, more area (duplicated cores), needs enough threads, contention in shared caches and memory.
Simultaneous multithreading	Same as fine-grained: interleaving instructions from multiple threads.	By fetching and issuing from multiple threads in the same cycle.	The same limitations and disadvantages as out-of-order execution and fine-grained multithreading.

2 Multithreading Exam Questions

- 2.1 With coarse-grained multithreading, if each thread incurs a 40-cycle memory miss penalty every 10 instructions (all other instructions take 1 cycle), what is the minimum number of threads needed to fully hide the memory latency? (*Sp26 MT2*)

5 threads. When a thread misses, the machine switches to another thread, which runs 10 instructions (10 cycles) before it misses too. The other $N - 1$ threads must supply at least 40 cycles of work while the first thread waits:

$$(N - 1) \times 10 \geq 40 \Rightarrow N \geq 5$$

Equivalently, each additional thread hides 10 cycles, so we need 4 threads in addition to the one that missed. (This ignores the cost of switching threads.)

- 2.2 With fine-grained multithreading, if each thread incurs a 40-cycle memory miss penalty every 10 instructions (all other instructions take 1 cycle), what is the minimum number of threads needed to fully hide the memory latency? (*Sp26 MT2*)

40 threads. Fine-grained multithreading switches threads every cycle, so with N threads each thread issues once every N cycles. All threads run in lockstep, so they reach the missing load at the same point: after a thread issues it, the 39 other threads must each issue the same instruction before the first thread's next instruction can go. That next instruction issues N cycles later, which must be no sooner than the 40-cycle penalty, so $N \geq 40$.

- 2.3 To support simultaneous multithreading (SMT) on an out-of-order machine supporting precise exceptions, which processor structures need to have separate per-thread instances? Select all that apply. (*Sp26 MT2*)

- | | | |
|---|--|---|
| ☐ Branch target buffer | ☐ ALU | ☐ Physical register file |
| ☐ Branch history table | ☐ Load/store unit | ☐ Reorder buffer |
| ☒ Program counter | ☒ Rename table | ☐ None of the above |

Only structures that hold each thread's *architectural* state need to be duplicated: every thread needs its own PC and its own rename table, which records where each of that thread's architectural registers lives. Everything else can be shared. The physical register file is a pool that every thread's rename table points into, ROB entries can be tagged with a thread ID, functional units don't care which thread an instruction came from, and branch predictors are only hints, so sharing them can hurt accuracy but never correctness.

- 2.4 *Fun question.* In 2024, Intel dropped Hyper-Threading (its SMT implementation) from the Lion Cove P-cores in Lunar Lake. What goes into making an SMT core? What hardware do you need, how could you improve its performance, and what are the drawbacks?

- **Required hardware:** a PC and rename table for each thread.
- **To improve performance:** more physical registers, separate branch predictors, and bigger caches, all of which increase area.

- **Drawbacks:** threads share the same microarchitecture (branch predictors, caches, etc.), which causes thrashing and has security implications, since one thread can observe another's effects on shared structures. Intel found that a hybrid architecture gives better results: just add more cores.

3 Cache Coherency

- 3.1 In multicore systems, L1 caches are often private to a single core. What could be a potential problem with this? Trace the sequence below on two cores with write-back caches (used to minimize bus traffic), where A starts as 6 in DRAM.

Step	Core 0 cache	Core 1 cache	DRAM
Initial	–	–	A = 6
Core 0: load A	A = 6	–	A = 6
Core 1: load A	A = 6	A = 6	A = 6
Core 0: store 7 → A	A = 7	A = 6	A = 6
Core 1: load A	A = 7	A = 6 (stale!)	A = 6

Each core can hold its own copy of the same address, and nothing keeps the copies in sync. Core 0's store only updates its own write-back cache, so Core 1 hits on its stale copy and reads 6 instead of the updated value of 7.

Cache coherence ensures that data stored in multiple private caches stays synchronized across a multicore system. For each memory location X, a coherent system maintains three invariants:

- Program order:** you see your own writes. If P writes X and then reads X, it gets its own value (assuming no one else wrote to X).
- Eventuality:** you eventually see others' writes. A write to X can't stay hidden forever, though it may take some time to arrive.
- Write serialization:** everyone agrees on the order of writes to X. If P1 writes X = 1 and P2 writes X = 2, no core can see 1 → 2 while another sees 2 → 1.

- 3.2 Back to the example: Core 0 stores 7 to A while Core 1 holds A = 6. What is the simplest way to make sure Core 1 doesn't read the stale value?

Core 0 tells Core 1 to invalidate A. Core 0 broadcasts an invalidation for A on the bus when it writes, and Core 1, which listens to (*snoops*) the bus for coherence messages, drops its copy. Core 1's next load of A then misses.

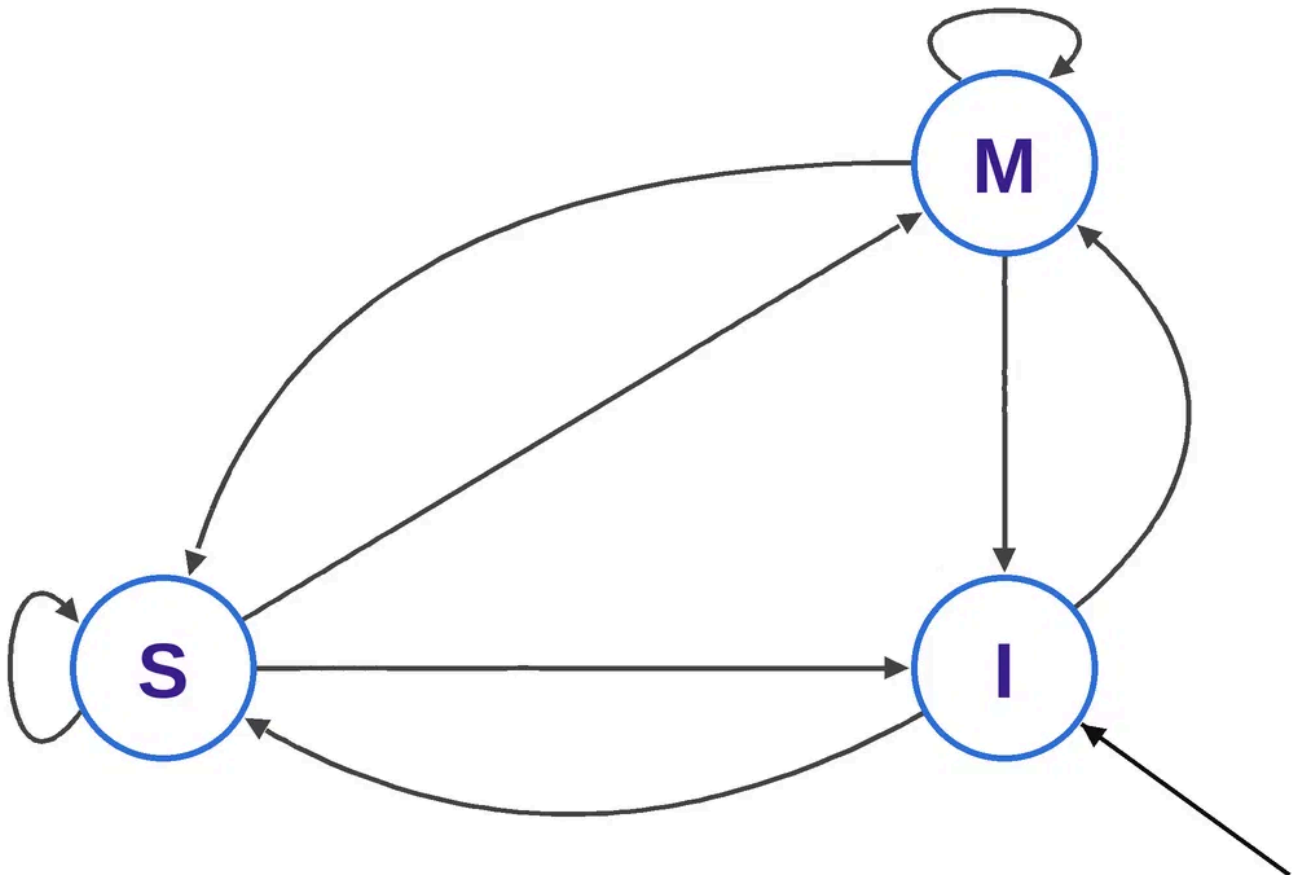
- 3.3 Core 1 now loads A and misses. Who supplies A?

Core 0, since it has the only up-to-date value (DRAM still holds 6). Core 1 asks for A on the bus, and Core 0 responds by putting A = 7 on the bus. Since the value is on the bus anyway, DRAM is updated too.

The MSI protocol. Every cache line in every cache has a state machine with three states:

- **Invalid (I):** no usable copy of this line. It was never loaded, was evicted, or was invalidated.
- **Shared (S):** a clean copy of the data. Other caches can hold this data too. Safe to read.
- **Modified (M):** this cache has the only valid copy of the line. At most one cache can hold a given line in M, and every other cache has it in I.

3.4 Fill in what causes each transition of the MSI state machine. Lines start in I on reset. The possible inputs are a *local* read, write, or eviction (performed by this cache's core) and a *remote* read or write (performed by another core / cache).



From	To	Cause
I	S	Local read: read miss, so load A from memory.
I	M	Local write: write miss brings the line into the cache and invalidates any other copies, so we are the only up-to-date holder.
S	S	Local read or remote read: A is in the cache and hasn't changed.
S	M	Local write: broadcast an invalidation to the other caches.
S	I	Remote write: another core's write invalidates our copy. Or local eviction: the line is clean, so it is simply dropped.
M	M	Local read or local write: no other core has requested the line, so we are still the only up-to-date holder.
M	S	Remote read: another core requests A, we supply it, and now everyone has the same value.
M	I	Remote write: another core writes A, so our data is no longer valid. Or local eviction: we hold the only up-to-date copy, so we write it back to memory first.

Implementations. So far, MSI has used **snooping**: each cache tracks the status of its cached lines by monitoring a broadcast medium (e.g., a bus) for transactions.

- 3.5 True or false: a snooping cache coherence protocol requires cores to communicate on a single physical bus.

False. Snooping requires a totally ordered broadcast network, but that can be built without a single shared-wire bus, for example with parallel interleaved buses (separate data and snoop broadcast) and multiple tag banks. See Sorin et al., *A Primer on Memory Consistency and Cache Coherence* (2011).

- 3.6 Three processors, P0–P2, run the MSI protocol, and every access below goes to the same cache line X. For each step, give the state of X in each cache, the bus operation, whether the step causes bus data traffic, and whether the access misses in the local cache.
- **Bus operations:** BusRd is a read miss, BusRdX is a write miss, and BusInv is a write hit to an S line (invalidate other copies).
 - **Bus data traffic:** 1 for BusRd or BusRdX, otherwise 0.
 - **Read/write misses:** 1 if the access misses in the local cache, otherwise 0.

Step	P0, P1, P2	Bus op	Bus data traffic	Read/write misses
Initial	I, I, I	–	0	0
P1 read X	I, S, I	BusRd	1	1
P2 read X	I, S, S	BusRd	1	1
P2 write X	I, I, M	BusInv	0	0
P2 write X	I, I, M	– (stays in M)	0	0
P0 read X	S, I, S	BusRd	1	1
P1 write X	I, M, I	BusRdX	1	1
P0 write X	M, I, I	BusRdX	1	1
P0 read X	M, I, I	– (stays in M)	0	0

When P0 reads X, P2 holds it in M, so P2 supplies the data and drops to S. Each write miss (BusRdX) invalidates every other copy, so only the writer is left holding X.