

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?

1.2 What are some examples of horizontal waste?

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			
Coarse-grained multithreading			
Multiprocessing			
Simultaneous 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*)

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*)

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 |

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?

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			
Core 1: load A			
Core 0: store 7 → A			
Core 1: load A			

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:

- (a) **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).
- (b) **Eventuality:** you eventually see others' writes. A write to X can't stay hidden forever, though it may take some time to arrive.
- (c) **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 $1 \rightarrow 2$ while another sees $2 \rightarrow 1$.

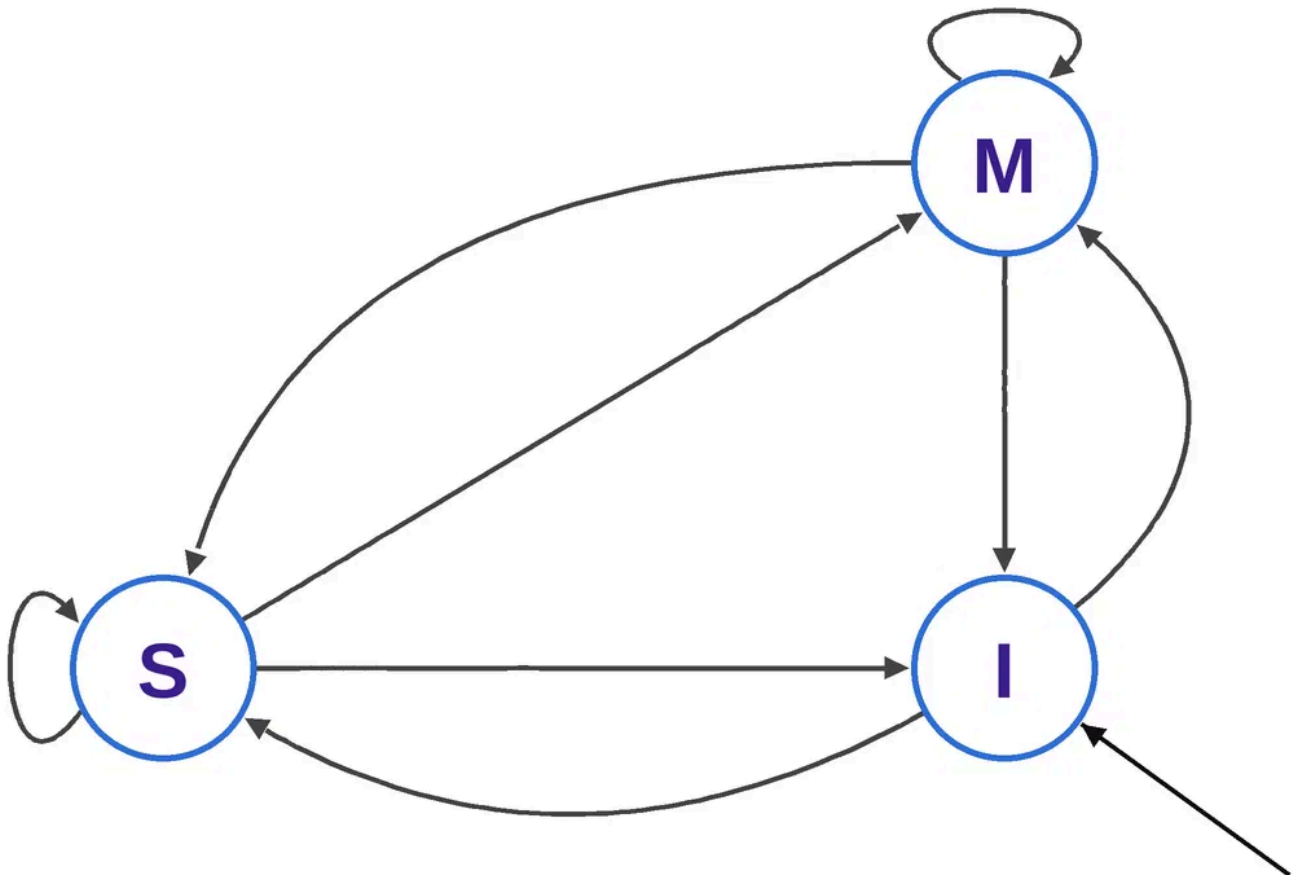
- 3.2 Back to the example: Core 0 stores 7 to A while Core 1 holds A = 6. What is the simplest way to make sure Core 1 doesn't read the stale value?

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

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	
I	M	
S	S	
S	M	
S	I	
M	M	
M	S	
M	I	

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.

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				
P2 write X				
P2 write X				
P0 read X				
P1 write X				
P0 write X				
P0 read X				