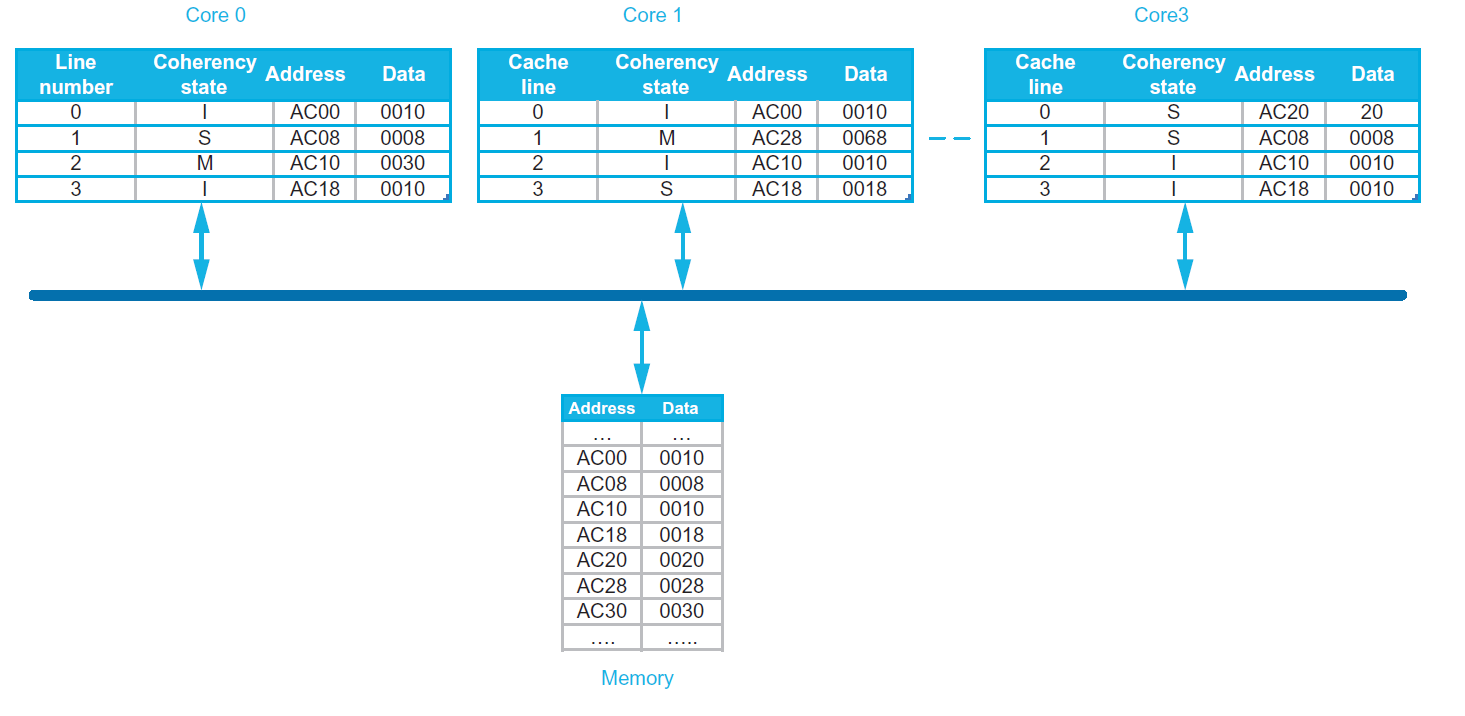
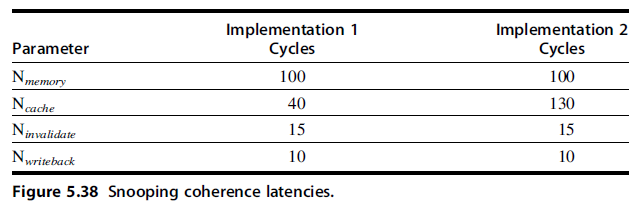
**Final Review Part 2**

**Q1. Memory Coherence Protocol**



Each core has a single, private cache with coherence maintained using the snooping MSI coherence protocol. Each cache is direct-mapped, with four lines, each holding 8 bytes (only the lower 2 bytes are shown to simplify the diagram). The above figure shows the initial state of all caches and memory. The performance of a snooping cache-coherent multiprocessor depends on many detailed implementation issues that determine how quickly a cache responds with data in an exclusive or Mstate block. In some implementations, a processor read miss to a cache block that is exclusive in another processor’s cache is faster than a miss to a block in memory. This is because caches are smaller, and thus faster, than main memory. Conversely, in some implementations, misses satisfied by memory are faster than those satisfied by caches. This is because caches are generally optimized for “front side” or CPU references, rather than “back side” or snooping accesses. Now consider the execution of a sequence of operations on a single processor core where

* read and write hits generate no stall cycles;
* read and write misses generate Nmemory and Ncache stall cycles if satisfied by memory and cache, respectively;
* write that generate an invalidate or write miss incur Ninvalidate stall cycles; and
* a write-back of a block, either due to a conflict or another processor’s request to an exclusive block, incurs an additional Nwriteback stall cycles.



For the following sequence of operations, how many stall cycles are generated by each implementation? And what is the final state of each cache and memory?

C0: R, AC28

C0: W, AC08 <-- 48

C0: W, AC30 <-- 78

|  |  |  |  |
| --- | --- | --- | --- |
| Core 0 | | | |
| line number | state | address | data |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Core 1 | | | |
| line number | state | address | data |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Core 3 | | | |
| line number | state | address | data |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

|  |  |
| --- | --- |
| Address | Data |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Q2. Coherence misses**

Assume that words z1 and z2 are in the same cache block, which is in the shared state in the caches of both P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit.

|  |  |  |  |
| --- | --- | --- | --- |
| Time | P1 | P2 | True, false or hit |
| 1 | write z1 |  |  |
| 2 |  | write z2 |  |
| 3 | read z1 |  |  |
| 4 |  | read z2 |  |

**Q3. Synchronization**

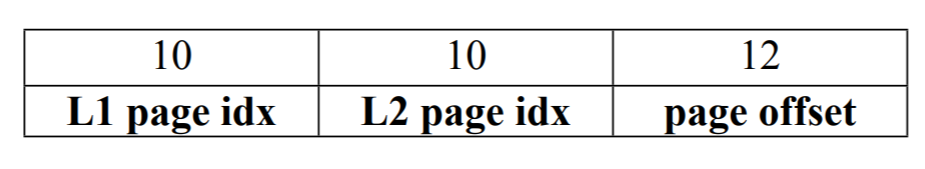
Last week, we have seen this producer-consumer model.

|  |  |
| --- | --- |
| Producer | Consumer |
| # address of tail pointer in x1  # data to be written in x2  ld x3, 0(x1)  sd x2, 0(x3)  addi x3, x3, 8  sd x3, 0(x1) | # address of tail pointer in x1  # address of head pointer in x2  ld x3, 0(x2)  spin:  ld x4, 0(x1)  beq x3, x4, spin  ld x5, 0(x3)  addi x3, x3, 8  sd x3, 0(x2)  # then process x5 |

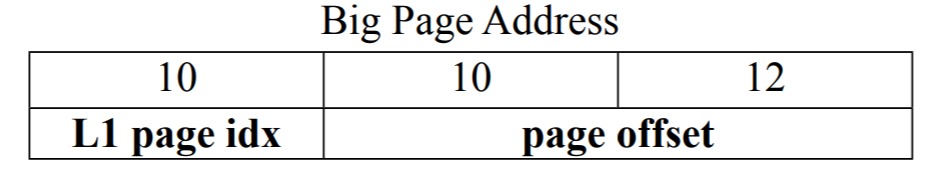
Now assume we have multiple producers sharing the same queue. We want to rewrite the producer code to make it thread safe. Assume we no longer have to worry about the order of storing to the queue vs. storing the tail pointer. This time we use **LR/SC** to write the producer code Assume as before that register x1 contains the address of the tail pointer and register x2 contains the data to be stored. Pseudocode for these primitives is shown below.

**Q4. Virtual Memory (2012 quiz2)**

For this problem, the machine you are studying uses a 2-level page table scheme. Also, the OS is smart enough to only allocate memory for the pages that it uses. Addresses in this machine are 32-bits long. The page offset is 12-bits in size. Both the level-1 page table index and level-2 page table index are 10 bits each.



1. You are asked to design a virtually indexed, physically tagged cache. The cache size is 16KB. What associativity must the cache have in order for there to be no aliasing?
2. Provide a sequence of addresses (in hexadecimal) that a user program could issue to the memory system that would give the fastest growth in total physical memory usage for this system
3. Consider a memory system in which you have a direct-mapped TLB with 64 entries. Assume that page table entries are not allocated in the cache. If a user program performs sequential (unit-stride) 32-bit accesses through 4GB of memory, how **many TLB misses will occur**? How many **individual memory accesses to the page table** are required?
4. Now considering “Big Page”.



Consider a direct-mapped TLB design which holds 32 entries for small page address translations and 32 entries for “Big Page” address translations. In terms of area and power, is this design cheaper, equal to, or more expensive than a direct-mapped TLB with 64 entries of only small page address translations?

**Q5. Multithreading**

In this question, we’ll consider the scalar pipeline from question in the context of multithreading, running the dot-product code:

loop: lw t0, 0(a2)

lw t1, 0(a3)

mul t2, t0, t1

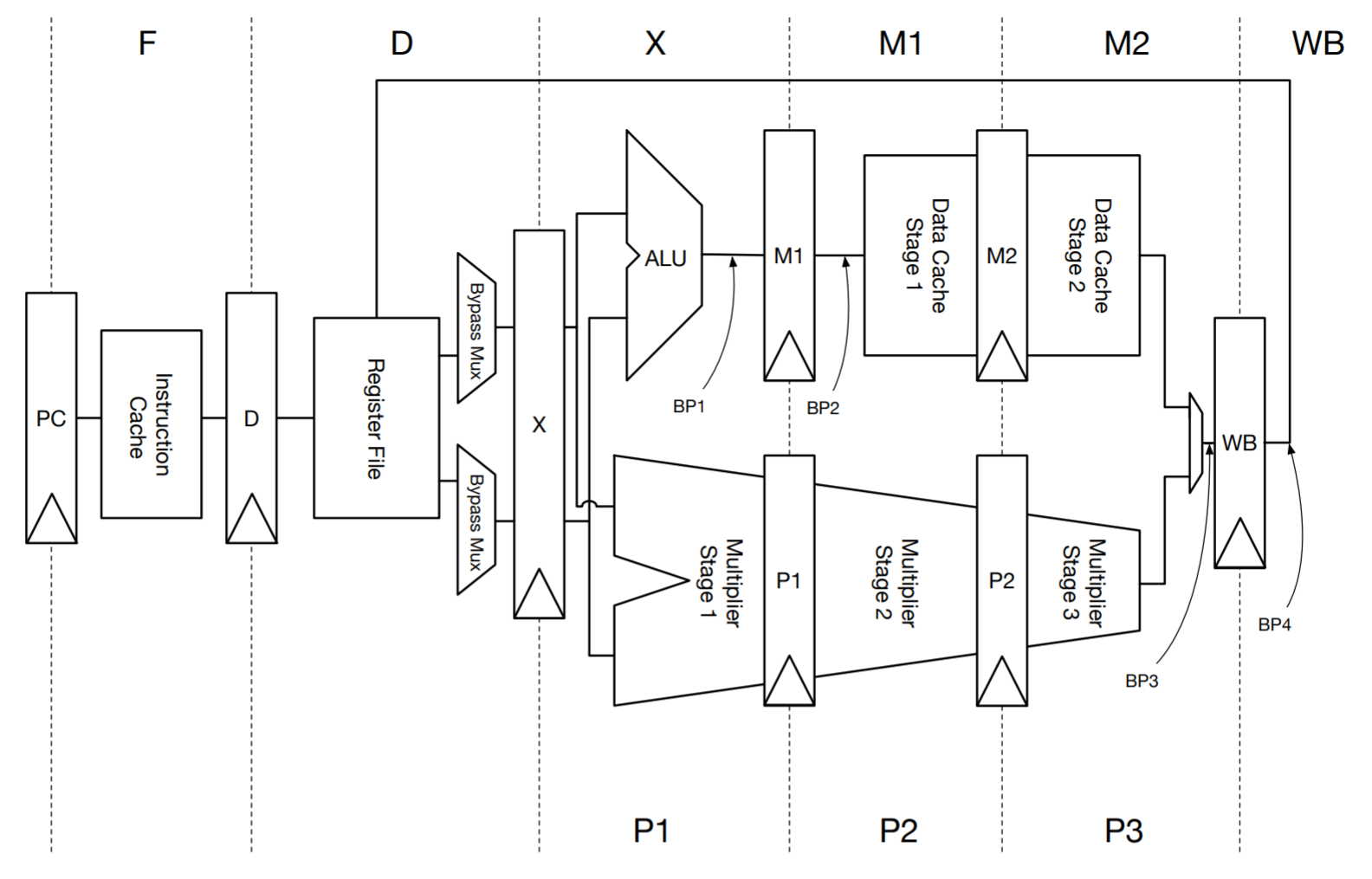
add a4, a4, t2

addi a2, a2, 4

addi a3, a3, 4

subi a1, a1, 1

bnez a1, loop



Here, all bypass paths are removed. The result of an instruction is available the cycle after register writeback. The branches are predicted as always taken.

1. If use a fixed-interleave scheduling scheme, how many threads are required to saturate this machine? Explain.
2. Using the same fixed-interleave scheduling scheme, in steady state, what fraction of the time must we insert a bubble, assuming we had only two threads available.
3. Using a dynamic scheduling scheme, in steady state, how many threads are required to saturate this machine? Explain.