| Problem Set Three<br>Due on Tue 4/16/96 | **Advanced Computer Architecture**<br>CLA CS-550 / Spring 1996 | Prof. Azer Bestavros<br>Email: best@cs.bu.edu |
|---|---|---|

Please do not underestimate the length of this problem set. Start as early as you can.
*Your grade will depend on the clarity and soundness of your assumptions*

# 1 General Memory Hierarchy Design Problems

Solve problems 5.4, 5.8, and 5.13 from Hennesy and Patterson, "Computer Architecture: A Quantitative Approach" 2nd Edition.

# 2 Vectorized DLX Architecture

Check the vectorized implementation of the DLX architecture described in the report available on the web from `http://www.cs.bu.edu/faculty/best/crs/cs550/fridman/vdlx/dlxsim.html`

Solve the three exercises included at the end of that report. The vectorized DLX simulator code is available under directory `/home/course/cs550/vdlx/`. Included in that directory as well are the files for the code used for exercises 1, 2, and 3.

## 3 Snoopy Cache Design

In a multiprocessor system where each processor has a private cache memory, several copies of a data object may exist at any time – one in the main memory and, potentially, others in one or more of the private caches. For such systems, one has to incorporate mechanisms to guarantee that whenever a processor accesses a data object, it receives the most up-to-date copy of the object. Typically, the system is configured so that the memory bus is shared among all CPU/CACHE pairs.

Each cache snoops on the bus in order to detect transactions of interest to it – and accordingly updates the state of its own entries. Each cache also takes the responsibility of informing other caches whenever it needs to perform an access which could lead to incoherency among the caches.

Typically, the cache-consistency-maintaining hardware consists of two finite state machines – a cache conctroller (CC) and a Snoop Controller (SC). The CC is primarily responsible for its own processor's use of the cache, interposing itself between the processor and the memory. The SC snoops at the memory bus identifying any operations which could require the state of a line in its cache to be updated; it essentially responds to requests of other processors.

In this problem, you are required to design a new cache coherence protocol for a multi-processor system. The protocol is to be based on the concept of "ownership" of a block (or line). A processor must own a block of memory before it is allowed to update it. Ownership also obligates the owner to provide the data to any other cache requesting it and to update main memory when the line in cache is to be replaced (copy-back). Ownership is acquired through special read and write operations. A block can be owned exclusively or non-exclusively as is explained below.

The possible states for cache entries and the nature of memory "bus trtansactions" have been defined for you. You are to use this skeleton and are required to come up with a complete definition of the protocol.

Any entry (block) in the cache can be in one of the four states defined in the table below:

| State | Meaning |
|---|---|
| INVALID | Cache line does not contain any valid data |
| UNOWNED | Data in the cache line is valid, and copies may exist in other caches. However, this cache does not own it and will have to acquire ownership to it if it needs to update it. |
| OWNED | This cache owns the data in the cache line but it cannot modify it without informing other caches. Further, as the owner, the cache is responsible for responding to read requests from other caches and for eventually copying the data back to main memory. |
| EXCLUSIVE | This cache owns and has an exclusive copy of the data in the cache line (no other cache has a copy). Thus, the line can be updated locally, without worrying about any other cache. The updated value can eventually be written to main memory. |

There are four types of bus transactions:

| Transaction/Meaning |
|---|
| Read-Shared (RS)<br>A conventional read. It gives a cache an ‘‘unowned’’ copy of the line. The copy may come from the main memory or from another cache, which owns that line. |
| Read-For-Ownership (RFO)\|<br>Like a normal read, except that teh requesting cache becomes the exclusive owner after the read is completed -- by causing other caches to invalidate matching entries. |
| Write-For-Invalidation (RFI)<br>A quick version of a regular write. It does not update the entire block in main memory, but invalidates other cached copies. Main memory is updated later on copy-back. |
| Write-Without-Invalidation (WWI)<br>Main memory is updated. Any cached entries in any cache are kept valid. Used by a processor simply to flush its owned blocks to main memory -- without the necessary invalidations which would have taken place otherwise. |

The idea of this protocol is to try to determine (guess), at compile-time, whether each memory reference is to a shared or unshared block and then use this information to reduce the number of invalidation transactions on the bus.

A block is considered “shared” if it is “read-only” – that is, if each cache can keep a local copy without fear of another one modifying it. An “unshared” block is one which may be modified – each cache can still keep a copy, but when any cache updates its copy, it is responsible for indicating this fact to the others. Since the type of the read transaction will be determined by the compiler, we will assume that the instruction set includes two types of “load” instructions for the shared/unshared cases. The RS operation is for reads to shared data. A cache does not acquire ownership of the block when it uses this instruction to read it. The RFO, on the other hand, results in the cache becoming the owner. The processor indicates to the cache (through the use of special signals) which one of these two reads is to be used each time.

Of course, compile time guesses may prove to be wrong. Further, the shared/nonshared attribute of a cache line may differ for different phases of the computation. Therefore, we have to provide for a change in attributes of the cache block. The two kinds of “writes’ serve this purpose. Unlike the case of reads, however, the type of a write is not determined by the processor. It is determined by the cache controller (CC). If the processor attempts to write to a block that it does not own, the CC issues a WFI transaction on the bus so that other caches invalidate their copies. The cache that requested the write is then made the owner of the cache line. If the processor owns the block, the data is updated in the cache only, and a WWI has to be used when the owned block is to be copied back to main memory (to be replaced with a new line).

a. For the protocol outlined above, construct a state transition table for a line in the cache of a CPU. A state transition table is simply a state transition graph in tabular form. Since the transitions for a line in cache will be determined by the two finite state machines (CC and SC), please identify which one (CC or SC) is responsible for each transition in your table. Notice that you need to differentiate between transactions that are requested (generated) locally by the local CPU (the one connected to the cache) or remotely over the bus. In your transition table you should have the following columns:

| Current State | Local/Remote Xaction | CC Actions | SC Actions | Next State |
|---|---|---|---|---|
| | | | | |

b. The protocol described above relies to a large extent on MIMD compiler technology. How easy (or difficult) do you think is it for a compiler to guess the "shared/unshared" attributes of a memory access correctly given a MIMD program? Give examples to substantiate your claims.

c. The process of guessing efficiently the "shared/unshared" attributes of a memory access at compile time becomes almost impossible if those attributes are data-dependent. Suggest (without going into any design details) a hardware mechanism that would "learn" at run-time the attributes of cache lines. (Hint: Ideas similar to those used to predict whether a branch should be taken or not might prove to be helpful.)