Cache Coherency

A multiprocessor and a multicomputer each comprise a number of independent processors connected by a communications medium, either a bus or more advanced switching system, such as a crossbar switch.

We focus this discussion on multiprocessors, which use a common main memory as the primary means for inter–processor communication. Later, we shall see that many of the same issues appear for multicomputers, which are more loosely coupled.

Logically speaking, it would be better to do without cache memory. Such a solution would completely avoid the problems of cache coherency and stale data. Unfortunately, such a solution would place a severe burden on the communications medium, thereby limiting the number of independent processors in the system.

This lecture will focus on a common bus as a communications medium, but only because a bus is easier to draw. The same issues apply to other switching systems.

Topics for today:

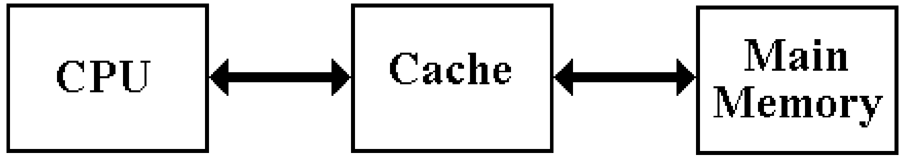
1. The cache write problem for uniprocessors and multiprocessors.
2. A simple cache coherency protocol.
3. The industry standard MESI protocol.

The Cache Write Problem

Almost all problems with cache memory arise from the fact that the processors write data to the caches. This is a necessary requirement for a stored program computer.

The problem in uniprocessors is quite simple. If the cache is updated, the main memory must be updated at some point so that the changes can be made permanent if needed.

Here is a simple depiction of a uniprocessor with a cache. As always, this could be (and often is) elaborated significantly in order to achieve better performance.



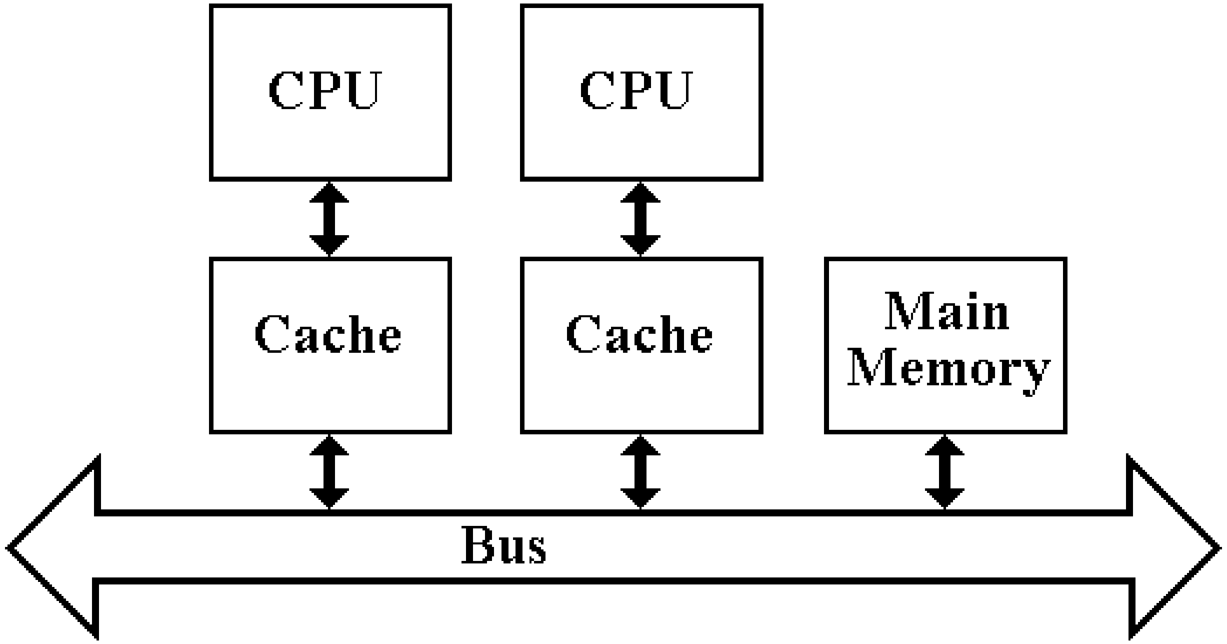
It was in this context that we first met the issue of cache write strategies. We focused on two strategies: write–through and write–back.

In the **write–through** strategy, all changes to the cache memory were immediately copied to the main memory. In this simpler strategy, memory writes could be slow.

In the **write–back** strategy, changes to the cache were not propagated back to the main memory until necessary in order to save the data. This is more complex, but faster.

The Cache Write Problem (Part 2)

The uniprocessor issue continues to apply, but here we face a bigger problem.



The **coherency problem** arises from the fact that the same block of the shared main memory may be resident in two or more of the independent caches.

There is no problem with reading shared data. As soon as one processor writes to a cache block that is found in another processor's cache, the possibility of a problem arises.

Cache Coherency: Introductory Comments

We first note that this problem is not unique to parallel processing systems. Those students who have experience with database design will note the strong resemblance to the “lost update” problem. Those with experience in operating system design might find a hint of the theoretical problem called “readers and writers”.

It is all the same problem: handling the problem of inconsistent and stale data.

The cache coherency problems and strategies for solution are well illustrated on a two processor system. We shall consider two processors P1 and P2, each with a cache.

Access to a cache by a processor involves one of two processes: read and write. Each process can have two results: a cache hit or a cache miss.

Recall that a **cache hit** occurs when the processor accesses its private cache and finds the addressed item already in the cache. Otherwise, the access is a **cache miss**.

Read hits occur when the individual processor attempts to read a data item from its private cache and finds it there. There is no problem with this access, no matter how many other private caches contain the data.

The problem of processor receiving stale data on a read hit, due to updates by other independent processors, is handled by the cache write protocols.

Cache Coherency: The Wandering Process Problem

This strange little problem was much discussed in the 1980’s (Ref. 3), and remains somewhat of an issue today. Its lesser importance now is probably due to revisions in operating systems to better assign processes to individual processors in the system.

The problem arises in a time–sharing environment and is really quite simple. Suppose a dual–processor system: CPU_1 with cache C_1 and CPU_2 with cache C_2. Suppose a process P that uses data to which it has exclusive access.

Consider the following scenario:

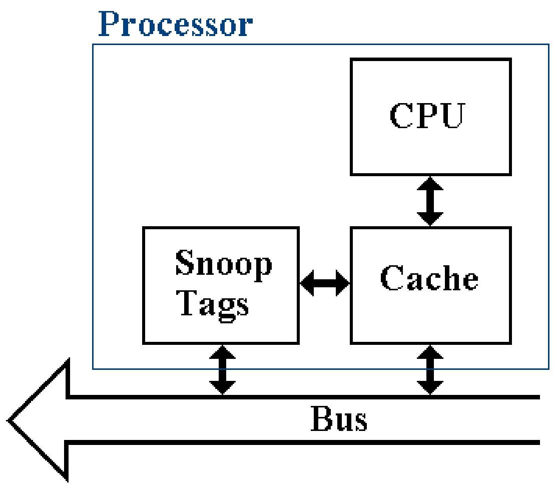
1. The process P runs on CPU_1 and accesses its data through the cache C_1.
2. The process P exceeds its time quantum and times out.
   All dirty cache lines are written back to the shared main memory.
3. After some time, the process P is assigned to CPU_2. It accesses its data through cache C_2, updating some of the data.
4. Again, the process P times out. Dirty cache lines are written back to the memory.
5. Process P is assigned to CPU_1 and attempts to access its data. The cache C_1 retains some data from the previous execution, though those data are stale.

In order to avoid the problem of cache hits on stale data, the operating system must flush every cache line associated with a process that exceeds its time quota.

Cache Coherency: Snoop Tags

Each line in a cache is identified by a cache tag (block number), which allows the determination of the primary memory address associated with each element in the cache.

Cache blocks are identified and referenced by their memory tags.



In order to maintain coherency, each individual cache must monitor the traffic in cache tags, which corresponds to the blocks being read from and written to the shared primary memory. This is done by a **snooping cache** (or **snoopy cache**, after the Peanuts comic strip), which is just another port into the cache memory from the shared bus.

The function of the snooping cache is to "**snoop the bus**", watching for references to memory blocks that have copies in the associated data cache.

Cache Coherency: A Simple Protocol

We begin our consideration of a simple cache coherency protocol. After a few comments on this, we then move to consideration of the MESI protocol.

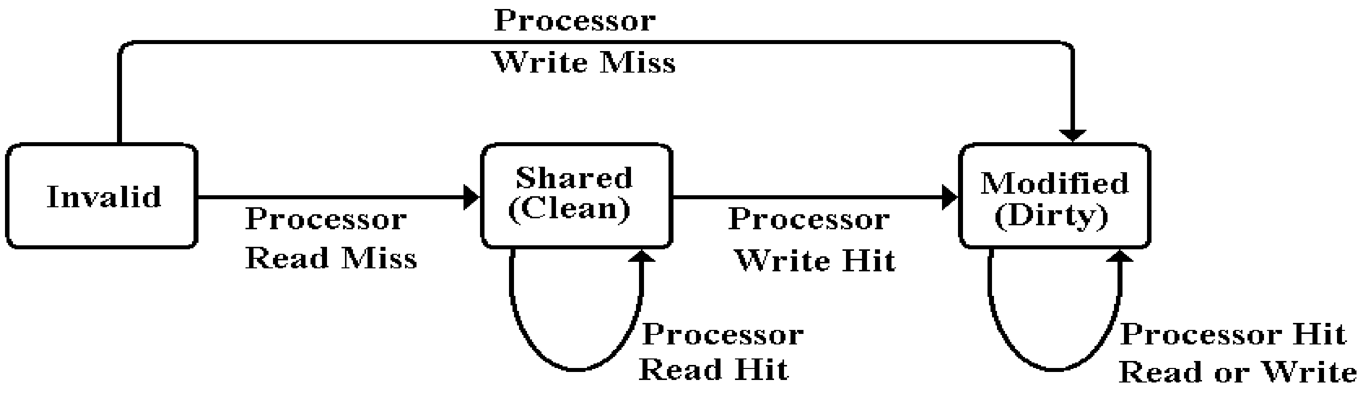
In this simple protocol, each block in the cache of an individual processor can be in one of three states:

1. Invalid: the cache block does not contain valid data.
2. Shared (Read Only): the cache block contains valid data, loaded as a result of a read request. The processor has not written to it; it is "clean" in that it is not "dirty" (been changed). This cache block may be shared with other processors; it may be present in a number of individual processor caches.
3. Modified (Read/Write): the cache block contains valid data, loaded as a result of either a read or write request. The cache block is "dirty" because its individual processor has written to it. It may **<u>not</u>** be shared with other individual processors, as those other caches will contain stale data.

Terminology: The word **"invalid**" has two uses here: 1) that a given cache block has no valid data in it; and 2) the state of a cache just prior to a cache miss.

A First Look at the Simple Protocol

Let’s consider transactions on the cache when the state is best labeled as “Invalid”. The requested block is not in the individual cache, so the only possible transitions correspond to misses, either read misses or write misses.



Note that this process cannot proceed if another processor’s cache has the block labeled as “Modified”. We shall discuss the details of this case later.

In a **read miss**, the individual processor acquires the bus and requests the block. When the block is read into the cache, it is labeled as “not dirty” and the read proceeds.

In a **write miss**, the individual processor acquires the bus, requests the block, and then writes data to its copy in the cache. This sets the dirty bit on the cache block.

Note that the processing of a write miss exactly follows the sequence that would be followed for a read miss followed by a write hit, referencing the block just read.

Cache Misses: Interaction with Other Processors

We have just established that, on either a read miss or a write miss, the individual processor must acquire the shared communication channel and request the block.

If the requested block is not held by the cache of any other individual processor, the transition takes place as described above. We shall later add a special state to account for this possibility; that is the contribution of the MESI protocol.

If the requested block is held by another cache and that copy is labeled as "Modified", then a sequence of actions must take place: 1) the modified copy is written back to the shared primary memory, 2) the requesting processor fetches the block just written back to the shared memory, and 3) both copies are labeled as "Shared".

If the requested block is held by another cache and that copy is labeled as "Shared", then the processing depends on the action. Processing a read miss only requires that the requesting processor fetch the block, mark it as "Shared", and execute the read.

On a **write miss**, the requesting processor first fetches the requested block with the protocol responding properly to the read miss. At the point, there should be no copy of the block marked "Modified". The requesting processor marks the copy in its cache as "Modified" and sends an **invalidate signal** to mark all copies in other caches as stale.

The protocol must insure that no more than one copy of a block is marked as "Modified".

Write Hits and Misses

As we have noted above, the best way to view a write miss is to consider it as a sequence of events: first, a read miss that is properly handled, and then a write hit.

This is due to the fact that the only way to handle a cache write properly is to be sure that the affected block has been read into memory.

As a result of this two–step procedure for a write miss, we may propose a uniform approach that is based on proper handling of write hits.

At the beginning of the process, it is the case that no copy of the referenced block in the cache of any other individual processor is marked as “Modified”.

If the block in the cache of the requesting processor is marked as “Shared”, a write hit to it will cause the requesting processor to send out a “Cache Invalidate” signal to all other processors. Each of these other processors snoops the bus and responds to the Invalidate signal if it references a block held by that processor. The requesting processor then marks its cache copy as “Modified”.

If the block in the cache of the requesting processor is already marked as “Modified”, nothing special happens. The write takes place and the cache copy is updated.

The MESI Protocol

This is a commonly used cache coherency protocol. Its name is derived from the fours states in its FSM representation: **M**odified, **E**xclusive, **S**hared, and **I**nvalid.

This description is taken from Section 8.3 of Tanenbaum (Reference 4).

Each line in an individual processors cache can exist in one of the four following states:

1.	Invalid	The cache line does not contain valid data.
2.	Shared	Multiple caches may hold the line; the shared memory is up to date.
3.	Exclusive	No other cache holds a copy of this line; the shared memory is up to date.
4.	Modified	The line in this cache is valid; no copies of the line exist in other caches; the shared memory is not up to date.

The main purpose of the Exclusive state is to prevent the unnecessary broadcast of a Cache Invalidate signal on a write hit. This reduces traffic on a shared bus.

Recall that a necessary precondition for a successful write hit on a line in the cache of a processor is that no other processor has that line with a label of Exclusive or Modified.

As a result of a successful write hit on a cache line, that cache line is always marked as Modified.

The MESI Protocol (Part 2)

Suppose a requesting processor processing a write hit on its cache. By definition, any copy of the line in the caches of other processors must be in the Shared State. What happens depends on the state of the cache in the requesting processor.

1.	Modified	The protocol does not specify an action for the processor.
2.	Shared	The processor writes the data, marks the cache line as Modified, and broadcasts a Cache Invalidate signal to other processors.
3.	Exclusive	The processor writes the data and marks the cache line as Modified.

If a line in the cache of an individual processor is marked as "Modified" and another processor attempts to access the data copied into that cache line, the individual processor must signal "Dirty" and write the data back to the shared primary memory.

Consider the following scenario, in which processor P1 has a write miss on a cache line.

1. P1 fetches the block of memory into its cache line, writes to it, and marks it Dirty.
2. Another processor attempts to fetch the same block from the shared main memory.
3. P1's snoop cache detects the memory request. P1 broadcasts a message "Dirty" on the shared bus, causing the other processor to abandon its memory fetch.
4. P1 writes the block back to the share memory and the other processor can access it.

Events in the MESI Protocol

This discussion is taken from Chapter 11 of the book Modern Processor Design (Ref. 5).

There are six events that are basic to the MESI protocol, three due to the local processor and three due to bus signals from remote processors.

Local Read	The individual processor reads from its cache memory.
Local Write	The individual processor writes data to its cache memory.
Local Eviction	The individual processor must write back a dirty line from its cache in order to free up a cache line for a newly requested block.
Bus Read	Another processor issues a read request to the shared primary memory for a block that is held in this processors individual cache. This processor’s snoop cache detects the request.
Bus Write	Another processor issues a write request to the shared primary memory for a block that is held in this processors individual cache.
Bus Upgrade	Another processor signals that a write to a cache line that is shared with this processor. The other processor will upgrade the status of the cache line from “Shared” to “Modified”.

The MESI FSM: Action and Next State (NS)

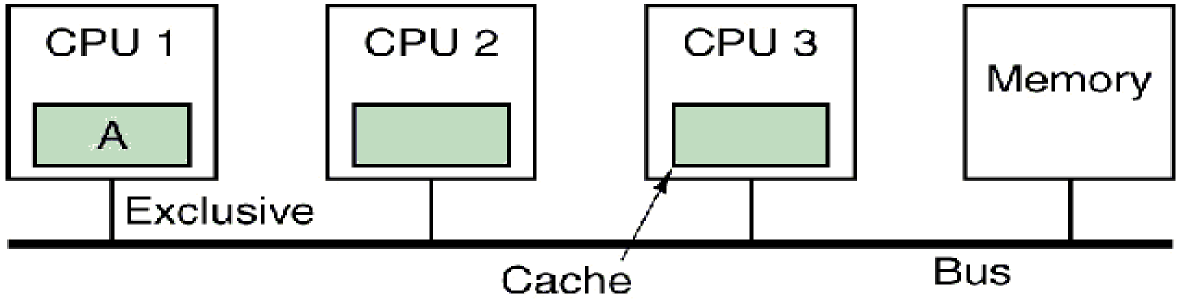
Here is a tabular representation of the Finite State Machine for the MESI protocol. Depending on its Present State (PS), an individual processor responds to events.

PS	Local Read	Local Write	Local Eviction	BR Bus Read	BW Bus Write	BU – Bus Upgrade
I Invalid	Issue BR Do other caches have this line. Yes: NS = S No: NS = E	Issue BW NS = M	NS = I	Do nothing	Do nothing	Do nothing
S Shared	Do nothing	Issue BU NS = M	NS = I	Respond Shared	NS = I	NS = I
E Exclusive	Do nothing	NS = M	NS = I	Respond Shared NS = S	NS = I	Error Should not occur.
M Modified	Do nothing	Do nothing	Write data back. NS = I.	Respond Dirty. Write data back NS = S	Respond Dirty. Write data back NS = I	Error Should not occur.

MESI Illustrated

Here is an example from the text by Andrew Tanenbaum (Ref. 4). This describes three individual processors, each with a private cache, attached to a shared primary memory.

When the multiprocessor is turned on, all cache lines are marked invalid. We begin with CPU reading block A from the shared memory.



CPU 1 is the first (and only) processor to request block A from the shared memory.

It issues a BR (Bus Read) for the block and gets its copy.

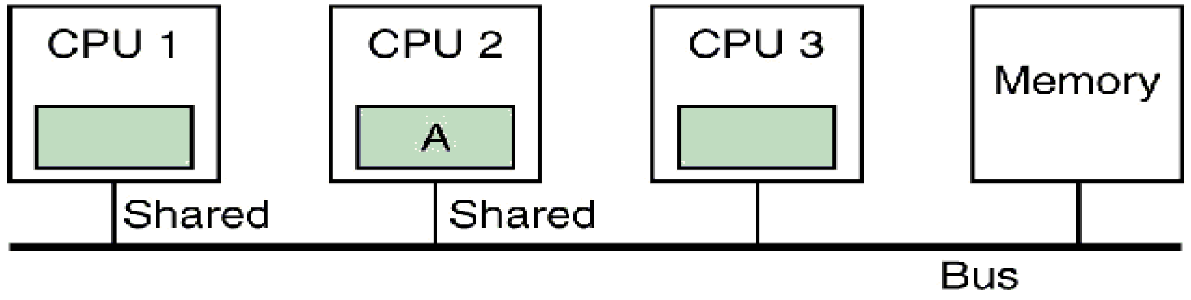
The cache line containing block A is marked Exclusive.

Subsequent reads to this block access the cached entry and not the shared memory.

Neither CPU 2 nor CPU 3 respond to the BR.

MESI Illustrated (Step 2)

We now assume that CPU 2 requests the same block. The snoop cache on CPU 1 notes the request and CPU 1 broadcasts “Shared”, announcing that it has a copy of the block.



Both copies of the block are marked as shared.

This indicates that the block is in two or more caches for reading and that the copy in the shared primary memory is up to date.

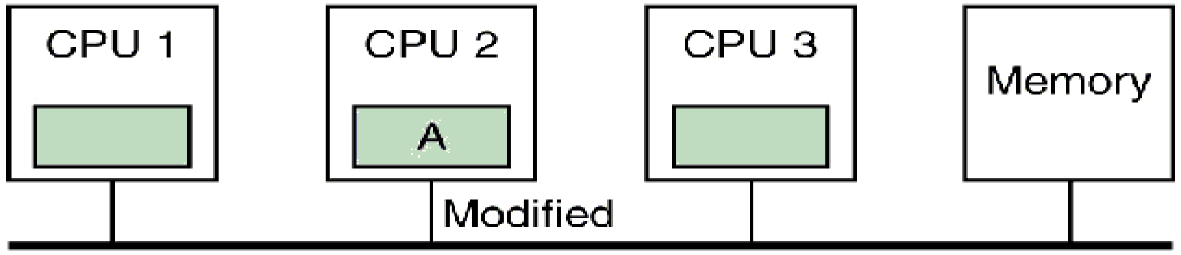
CPU 3 does not respond to the BR.

MESI Illustrated (Step 3)

At this point, either CPU 1 or CPU 2 can issue a local write, as that step is valid for either the Shared or Exclusive state. Both are in the Shared state.

Suppose that CPU 2 writes to the cache line it is holding in its cache. It issues a BU (Bus Upgrade) broadcast, marks the cache line as Modified, and writes the data to the line.

CPU 1 responds to the BU by marking the copy in its cache line as Invalid.



CPU 3 does not respond to the BU.

Informally, CPU 2 can be said to "own the cache line".

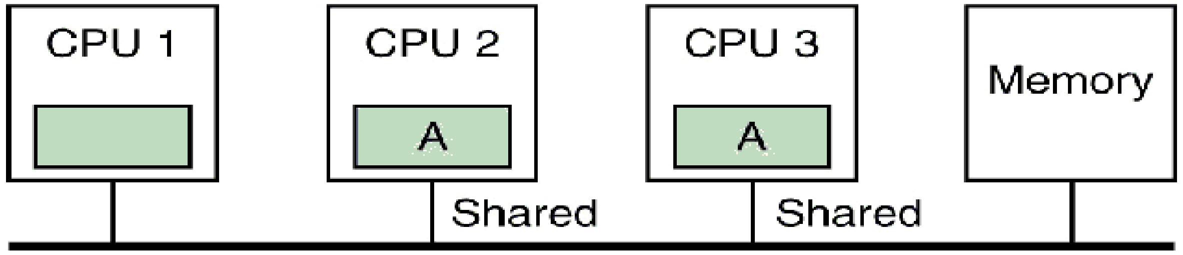
MESI Illustrated (Step 4)

Now suppose that CPU 3 attempts to read block A from primary memory.

For CPU 1, the cache line holding that block has been marked as Invalid.
CPU 1 does not respond to the BR (Bus Read) request.

CPU 2 has the cache line marked as Modified. It asserts the signal “Dirty” on the bus, writes the data in the cache line back to the shared memory, and marks the line “Shared”.

Informally, CPU 2 asks CPU 3 to wait while it writes back the contents of its modified cache line to the shared primary memory. CPU 3 waits and then gets a correct copy. The cache line in each of CPU 2 and CPU 3 is marked as Shared.



Tanenbaum’s actual example continues for a few more steps, but this sample is enough to illustrate the MESI process.

Summary

We have considered cache memories in parallel computers, both multiprocessors and multicomputers. Each of these architectures comprises a number of individual processors with private caches and possibly private memories.

We have noted that the assignment of a private cache to each of the individual processors in such architecture is necessary if we are to get acceptable performance.

We have noted that the major issue to consider in these designs is that of **cache coherency**. Logically speaking, each of the individual processors must function as if it were accessing the one and only copy of the memory block, which resides in the shared primary memory.

We have proposed a modern solution, called MESI, which is a protocol in the class called “Cache Invalidate”. This shows reasonable efficiency in the maintenance of coherency.

The only other class of protocols falls under the name “Central Database”. In this, the shared primary memory maintains a list of “which processor has which block”. This centralized management of coherency has been shown to place an unacceptably high processing load on the shared primary memory. For this reason, it is no longer used.

References

In this lecture, material from one or more of the following references has been used.

1. **Computer Organization and Design**, David A. Patterson & John L. Hennessy, Morgan Kaufmann, (3rd Edition, Revised Printing) 2007, (The course textbook) ISBN 978 – 0 – 12 – 370606 – 5.

2. **Computer Architecture: A Quantitative Approach**, John L. Hennessy and David A. Patterson, Morgan Kauffman, 1990. There is a later edition. ISBN 1 – 55860 – 069 – 8.

3. **High–Performance Computer Architecture**, Harold S. Stone, Addison–Wesley (Third Edition), 1993. ISBN 0 – 201 – 52688 – 3.

4. **Structured Computer Organization**, Andrew S. Tanenbaum, Pearson/Prentice–Hall (Fifth Edition), 2006. ISBN 0 – 13 – 148521 – 0

5. **Modern Processor Design: Fundamentals of Superscalar Processors** John Paul Shen and Mikko H. Lipasti, McGraw Hill, 2005. ISBN 0 – 07 – 057064 – 7.