LESSON 28:
MULTIPROCESSORS

Objectives:

1.understanding Multiprocessor Cache Coherence.

2 .understanding Multi Level Caches

3.understanding Challenges of Parrallel Processing

Multiprocessor Cache Coherence

Introduction of private caches attached to the processors helps greatly in reducing average latencies. Caches in multiprocessors are even more useful then in uniprocessors, since they also increase effective memory and communication bandwidth. However, this solution imposes another serious problem. Multiple copies of the same data block can exist in different caches, and if processors are allowed to update freely their own copies, inconsistent view of the memory is imminent, leading to program malfunction. This is what we called as the cache coherence problem.

Hardware Solutions

The common solution to solve the cache coherence problem is to add hardware support that increases the performance of multiprocessor systems without increasing the complexity of the software itself.

Benefits

- Hardware approaches make the consistency maintenance fully transparent for all levels of software, which simplifies the programming model of the multiprocessor system.
- The hardware solution makes the system totally transparent to software. Hardware protocols free the programmer and compiler from any responsibility about coherence maintenance and impose no restrictions to any later of software.
- Technology advances made their soft quite acceptable, compared to the system costs.
- Hardware solution dynamically recognizes the inconsistency conditions for shared data entirely at run-time. Thus, better performance can be achieved, especially for higher levels of data sharing, since the coherence overhead is generated only when actual sharing of data takes place.

Disadvantages

- Increased of hardware complexity

Software Solutions

Although software solutions are not common, this method is increasingly being frequently used. Solutions using combination of hardware and software have become more promising. Software-based solutions generally rely on the actions of the programmer, compiler or operating system in dealing with coherence problem. The simplest way to solve a problem in software is to declare non-cacheable pages of shared data. Special cache managing instructions are often used for cache bypass, flush, and indiscriminate or selective invalidation, in order to maintain coherence.

Benefits

- Software schemes are generally less expensive than their hardware counterparts.
- It is claimed that they are more convenient for large, scalable multiprocessors.

Disadvantages

Decisions about coherence related actions are often made statically during the compiler analysis, creating inevitable inefficiencies since the compiler analysis is unable to predict the flow of program execution accurately and conservative assumptions have to be made.

Hardware Solution

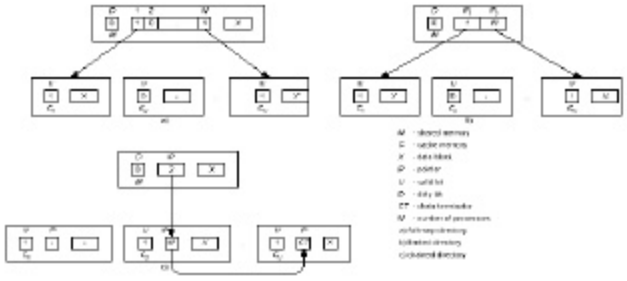

Figure 4-1 Three directory organizations

Directory Protocols

The directory protocol schemes has a global system-wide status information relevant for coherence maintenance which is stored in some kind of directory, The responsibility for coherence preserving is delegated to a centralized controller which is usually a part of the main memory controller. When the centralized controller receives a request from the local cache controllers, it checks the directory and issues the necessary commands for data transfer between memory and caches or between caches themselves. The centralized controller is responsible for keeping status information up-to-date to ensure that local action which can affect the global state of the block must be reported to the central controller. Private caches also store some local state information about the cache blocks. In contrast with snoopy protocols, the directory protocols are primarily suitable for multiprocessors with general interconnection networks.

Full-map directory schemes

One of the main characteristics of this scheme is that the directory is stored in main memory

and contains entries for each memory block. Exact locations of every cached copy of memory block and its status exist as an entry in the directory. Coherence of data in private caches is maintained by sending messages to known locations, thus avoiding usually expensive broadcasts. The classical full-map directory scheme employs a directory which contains entries for each memory block in the form of a bit vector. An entry consists of N+1 bits: one presence bit per each of N processor-cache pairs and one bit to denote whether block has been modified in one of the caches. One valid and one modified bit are stored in local cache directories.

Advantages:

1. Locating necessary cached copies is easy as only caches with valid copies are involved in coherence actions for a particular block.

Disadvantages:

1. The centralized controller is inflexible for system expansion by adding new processors. Thus, the system is not scalable.
2. Requests are directed to the central directory, thus it can become a performance bottleneck
3. The directory imposes significant memory overhead in multiprocessor systems with a large number of processors

Limited directory schemes

As seen from the previous section, memory overhead is very significant for the full-map directory scheme. Thus, to cope with this inefficiency, a partial map or limited directory scheme is used. The bit vector in the main memory directory has been now reduced to a small number of identifiers pointing to cached copies. Entries in limited directories contain a fixed number of pointers. Special actions have to be taken when the number of cached copies exceeds the number of pointers. In a broadcast capability system, an invalidate signal can be broadcasted to invalidate all copies. If such system does not exist, one copy has to be invalidated, to free the pointer for a new cached copy. The scalability of limited directory protocols is quite good. As the number of processors increase, the size of the directory would either increase a little or remain constant. However, the performance in heavily dependant on the sharing characterisitics of the parallel application. There is a limit on the number of processors which can share the data block at any one time.

Chained directory schemes

The limited directory scheme is good in terms of scalability; however it has one major disadvantage. The approach limits the number of cache copies in the multiprocessor system. The chained directory scheme contains entries which are organized in the form of linked lists, where all caches sharing the same block are chained through pointers into one list. This list spread across the individual caches. The main memory is only used to point to the head of the list and to keep the block status. The requests are sent to the main memory and then subsequent request are forwarded through the list, using the pointers. The last member of the list is terminated using the chain terminator. When a read miss occurs, the requester is put onto the head of the list. On a write miss, the invalidation signal is sent through the list to from head to the tail of the list. Actions are normally acknowledged with reply messages from the centralized arbiter. As a result of this complexity, the chained directory scheme is able to perform as good as the full-map directory schemes while maintaining its scalability ability to support more processors.

Snoopy Protocols

The Snoopy Protocol method to solve the coherence problem does not employ any centralized controller and global state information. Coherence maintenance is based solely on the actions of local cache controllers and distributed local state information. Thus, all the actions for the shared block must be announced to all other caches, via broadcast capability. Snooping protocols became popular with multiprocessors using microprocessors and caches attached to a single shared memory because these protocols can use a preexisting physical connection which is the bus to memory interconnection. This connection can be used to interrogate the status of the caches. This type of protocols are usually deployed in multiprocessor systems which use shared bus as global interconnect, since the shared bus provides very inexpensive and fast broadcasts, leading to cost-effectiveness and flexibility. The disadvantage to using this protocol is that coherence actions on the shared bus increase the bus traffic and make the bus saturation more acute. Thus, only multiprocessor systems with s mall to medium number of processors can be supported by snoopy protocols.

Write-invalidate snoopy protocols

The write-invalidate protocol is a protocol style which is used is both snooping and directory schemes. This protocol ensures that a processor has exclusive access to a data item before it writes that item to memory. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs. It invalidates all other cached copies of the items in other caches in the other processors. Write-back caches are used in this protocol as they can use the snooping protocol scheme for both cache misses and writes. Each processor snoops every address places on the bus. If a processor finds that it had a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. Despite the slight increase in complexity, write-back caches are preferable in multiprocessors as they generate lower requirements for memory bandwidth.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Figure 4-2 An example of an invalidation protocol for a single cache block (X) with write-back caches

The example above shows how an invalidation protocol works on a snooping bus. In the

beginning, the content at memory location X contains the value 0. When CPU A reads X, a cache miss is issued by A. The contents of CPU A will now contain 0. When CPU B reads X, another cache miss is issued by B. Now, both CPUs contain the value 0. However, when CPU A writes a value of 1 to X, an invalidation signal is broadcast to the rest of the CPUs on the memory bus. Doing so would invalidate all contents at that particular location in each private cache. When, CPU B reads X again, CPU B has to issue a cache miss to obtain a new value of X from shared-memory.

To perform invalidation, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. The corresponding data in the cache are then invalidated. As there is only one main memory bus, this enforces serialization of writes. One processor must obtain bus access before the other to complete to write to the same location. When a processor obtain bus access, the other processor's copy will be invalidated, causing writes to be strictly serialized.

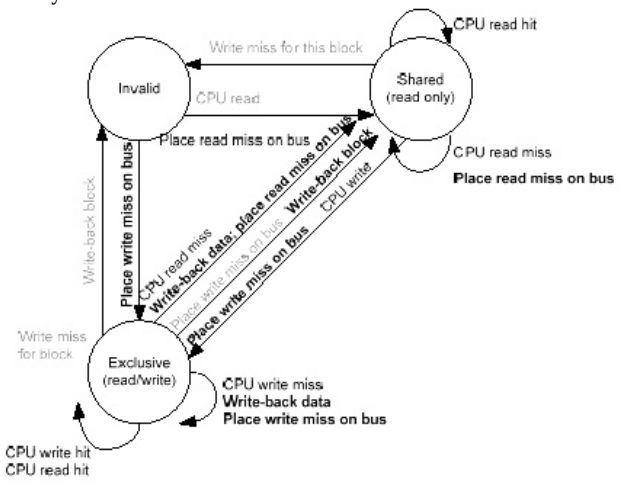

Figure 4-3 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray

The above illustrates the write-invalidate snoopy protocol. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the CPU does not match the address in the cache block. An attempt to write a block in the shared state always generates a write miss, even though the data exist in the cache. This is to invalidate all data in other caches, so that no cache has inconsistent data. The processor that generates the write miss obtains exclusive access to the data block.

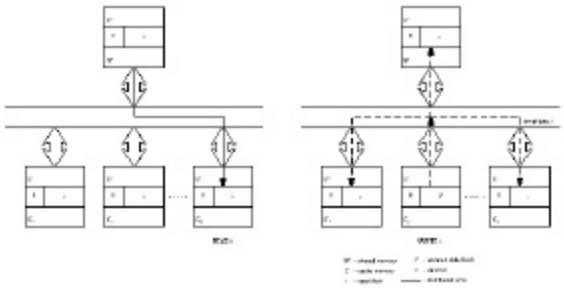

Figure 4-4 Write strategy in Write-invalidate snoopy protocols

Write-update protocols

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of scheme provides a distributed write approach which allows existence of multiple copies of cache blocks with write permission. The word to be written to a shared block is broadcast to all caches, and caches containing that block can update it. A special bus line for dynamic detection of the sharing status for a cache block is employed. When a block in cache is not shared, that particular block is marked as private and writeupdate protocol is not used. In Short, write-through is used for shared data while write-back is used for private data. The write-update protocol keeps the bandwidth under control by tracking down whether or not a word in the cache is shared by other caches in the system. If it is not, then there is no need to broadcast or update any other caches.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

Figure 4-5 An example of a write update working on a snooping bus with write-back caches

The example above shows how the write update scheme works. The contents at memory location X is 0. When CPU A reads X, a cache miss is issued by A. When B reads X, another cache miss is issued. Now, both caches in A and B contains the value 0 for location X. However, when CPU A writes a 1 to X, a write broadcast is issued to all processors connected to the memory bus. All CPUs (CPU B in this case) with location X in cache takes the new broadcasted value and update their own private caches. A cache miss would not need to be issued when CPU B later reads X. The disadvantage of write-update protocols is the performance overhead resulted from process migration. Allowing the processes to be switched among the processors produces false sharing when logically private data become physically shared. So, actual sharing increases and unnecessary coherency overhead is induced. It can be concluded that the write-update protocol scheme is suited for applications with tighter sharing of data. Performance can be seriously impeded with frequent unnecessary write broadcast in the case of process migration and sequential sharing.

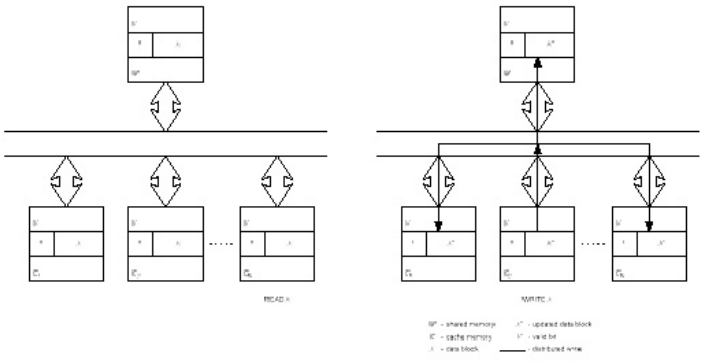

Figure 4-6 Write strategy in Write-update Snoopy protocols

Adaptive protocols

It can be observed that write-update protocols are suitable for applications with tighter sharing. Write-invalidate protocol excels for sequential sharing as multiple writes to the same block in cache only results in one invalidation broadcast. To support a wider range of workloads, the adaptive protocol is used. The adaptive protocol has been proposed that combined invalidation and update policy in a suitable way. The scheme starts with write broadcast whenever a processor writes a shared-memory. However, when a longer sequence of local writes is encountered or predicted, the invalidation signal for that particular block is sent. This solution is known as adaptive because it attempts to adapt the coherence mechanisms to the observed and predicted data use, in the effort to achieve the optimal performance.

Lock-based protocols

The coherence problem is very closely related to synchronization and mutual exclusion in accessing shared data. The snoopy coherence schemes enforce a strict consistency model and do not address these issues explicitly. Another method to obtain exclusive access during write to shared data is to use the system bus as a semaphore. This protocol employs to additional states dedicated to lock handling; one for lock possession and the other for lock waiting. Thus, the lock-waiter is able to work in the background while waiting for the lock to be released. Unlock is broadcast on the system bys, so the snoop busywait register of waiter can recognize it upon match. The lock is obtained after a prioritized bus arbitration. The processor is then interrupted to use the locked data. Waiting for the lock is organized through distributed, hardware implemented FIFO queues. Requesting caches are organized in a waiting list. What makes this protocol stand out from the 3 previous protocols is that the lock-based protocol is able to distinguish between shared and exclusive locks.

Multi-level Caches

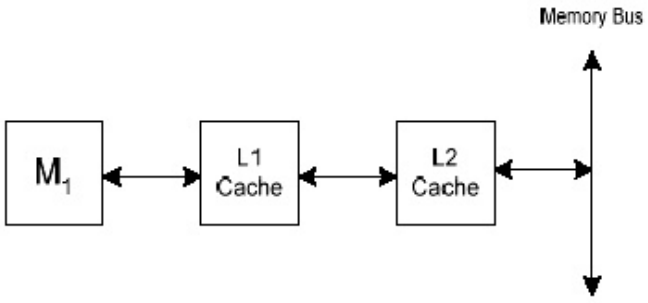

Figure 4-7 An example of a multi-level cache configuration

The previous chapters discuss based on a single level cache architecture. A multi-level cache is introduced to further reduce the latency when fetching data from memory. A faster cache (but smaller in size) will be placed closer to the processor to speed up performance when fetching local data. The introduction of multi-level caches introduces the cache coherence problem among the cache levels. Should each cache layer snoop the bus to maintain coherency? This can be avoided if the system uses the inclusion cache property whereby the data in L1 is a subset of the data in the L2 cache. L2 includes all entries in L1. All transactions relevant to L1 are relevant to L2. Thus, only L2 is responsible for snooping the bus.

MESI protocol

The MESI protocol is a widely used cache coherency and memory coherence protocol, which was later introduced by Intel in the Pentium processor to "support the more efficient writeback cache in addition to the write-through cache previously used by the Intel 486 processor."

The MESI protocol is also known as the Illinois protocol and is an example of an invalidate cache coherency protocol.

Every cache line is marked with one of the four following states (coded in two additional bits):

M – Modified: Indicates that this cache line was modified and therefore the underlying data in the main memory is no longer valid

E – Exclusive: Indicates that this cache line is only stored in this cache and hasn't been changed by a write access yet

S – Shared: Indicates that this cache line may be stored in other caches of the machine,

ALL copies identical to memory copy

I – Invalid: Indicates that this cache line is invalid.

Example of processors using this protocol : PowerPC, Pentium

Writes to SHARED cache lines are write through whereas writes to EXCLUSIVE cache lines are write-back. If a cache observes a bus transaction for an address which it contains, it asserts

the SHARED bus line.

Here is the state transition diagram for a cache line:

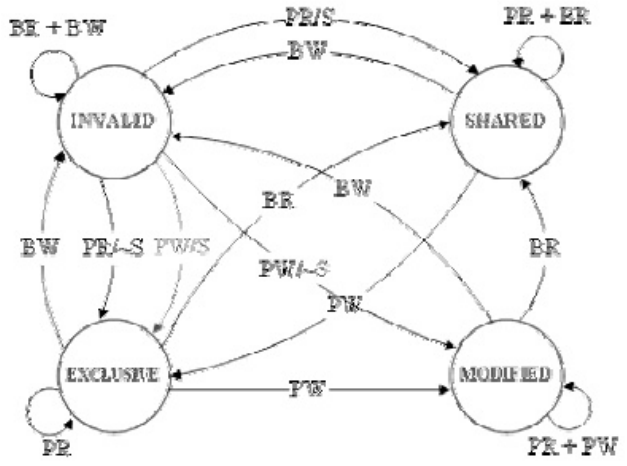

Figure 4-8 State diagram of MESI protocol

In the PW transitions from the INVALID state: On a write-miss, the cache line is first read from memory and then over written.. If the read result is an EXCLUSIVE cache line, the write is write-back, resulting in a MODIFIED cache line. If, however, the read results in a

SHARED cache line, the write is write-through, resulting in an EXCLUSIVE cache line.

Software Solution

Hardware approaches are normally used as they make the consistency maintenance fully transparent for all levels of software, which simplifies the programming model of the multiprocessor system with private cache memories. However, hardware complexity is increased.

Software solutions decrease the complexity of the hardware support. Moreover, software approaches tend to be more efficient than hardware approaches, in a class of non-numeric applications characterized with the predominantly migratory data.

There are two major classifications of software solutions namely static schemes and dynamic schemes. The implementations details are not discussed in this report but just the overview of the schemes employed.

Static Schemes

Static schemes are usually associated with compilation of the program prior to runtime. An analysis in perform on the program to find the potential inconsistencies that may arise during execution and additional information is added into the program file. Additional information includes marking of data, marking of references and insertion of special instructions. With such analysis, this eliminates the need of the processors to communicate during the execution of the critical parts of the codes. Thus, the system can be made more scalable than more hardware approaches. However, precise prediction of memory conflicts is impossible and will thus degrade the system performance as a whole. Two examples of static schemes are provided.

C.mmp page marking

One early solution to the cache coherence problem using a static software solution is to use the C.mmp page marking during compilation, The purpose of this marking is to keep the read-write shared data out of the cache at all times. The read-only shared pages can only be cached. Consequently, read-only shared instructions will be cached from memory. A relocation register is used to mark the pages which can be cached using only a 1 bit value. It can be obviously seen that this scheme is very restrictive and thus reduces performance. Another method is shown below.

Version Control

This scheme is a software approach with hardware support by the system. Whenever a write is made to a shared variable, a new 'version' of the content is formed. When tasks are on the same level on the execution graph, there is no data dependency between the iterations. Thus, there is no hazard of another processor needing the new version of the shared variable. When the processor passes the task level boundary, the most recent version of the variable in cache must be stored into the main memory. Each processor keeps tracks of the version of the data. A current Version Number (CVN) is maintained. An instruction is inserted into the program by the compiler to increment the CVN for variables modified at previous task level when the processor passes to the next task level. The Birth Version Number (BVN) is loaded during the CVN increment and updated when each cache line from main memory is loaded. When each cached data is written, the BVN equals the number of the next version. CVN and BVN are compared when a shared data is accessed. If BVN is smaller than CVN, data in cache is stale and a read miss occurs. Else, data is valid and a hit occurs.

Dynamic Schemes

In this scheme, the consistencies of private caches are maintained entirely at runtime by the kernel of the operating system. Thus, this does not affect the complexity of compilers for such multiprocessor systems. Dynamic approaches do not have to do any preventive actions as it is possible that only the actions which are absolutely necessary will be performed. Parallelisms modeled via parallel loops are hard to apply in dynamic approaches. Thus, it is not suitable for numeric applications. Numeric applications are normally based on parallel loops (i.e. FFT etc). However, non-numeric concurrent applications are mostly suitable for such type of schemes especially for such applications which uses critical regions for explicit management sharing.

One time identifiers

One way to maintenance cache consistency is to use operating system level primitive operations for control of the exclusive access to critical regions. In a page-organized memory, the real page address is calculated from the virtual page address at each access to a page, using the hashing function. To avoid unnecessary repetitions of this operation, a translation lookaside buffer (TLB) can be introduced, containing the virtual page address and the real page address. Each TLB entry and each line in the processor cache are expanded with another field, representing a unique marking of a shared data page. This field is referred to as OTI (one time identifier). When a new TLB entry is loaded, a new and unique value is read from a special incrementing register and placed into its OTI field. When an entry for a SHARED page id accessed for the first time, the entry is loaded into the cache from the main memory. The OTI field value in the TLB is passed to the OTI field in the cache. All subsequent accesses to this variable check if the value of the OTI field in cache matches the value of the OTI field in the TLB. When a match occurs, the access is a hit. After the exit from a critical region, the processor invalidates all TLB entries of the pages which belong to the shared data. The corresponding TLB entry is loaded again when the next access to shared data is performed. Consistency is maintained at each processor autonomously, without any communications with other processors in the system. However, complex hardware support is needed for this method.

Consistency using Interrupt Request

This is a hard-software solution which utilizes the hardware based bus monitor to detect the inconsistency conditions. It then informs the local processor about that, by issuing an interrupt request. The processor executes the interrupt routine and performs the consistency maintenance related activities. This consistency mechanism is used for both shared pages and

the page that contains entries of the page table.

The main memory contains a sequence of cache page frames where each frame can either be shared or exclusive (private). In the shared state, the page contains the last recently written page value. The other local cache memories can contain multiple copies of that particular page. In the private stage, only one copy can exist in any on cache. Each bus monitor contains the private table of actions to be performed when that address appears on the bus. No action is taken if the page frame is not in the private cache memory. If the page is shared, the monitor has to ignore all read-shared requests. All read private and assert-ownership requests have to be aborted. The bus monitor then interrupts the local processor. The processor than invalidates the page in cache memory. When the page is private, the bus monitor has to interrupt the processor for every transaction to maintain the consistency of data in the cache. When the page is dirty, the processor has to perform a write-back to main memory. Thus, the main memory always has the updated value and a request for a shared copy can be done immediately. When a private copy is requested, this triggers the invalidation of all other copies of that page in memory.

Challenges of Parallel Processing

Amdahl's Law is expressed as

$$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

The limited parallelism in programs will impede and limit the speedup of the multiprocessor system as a whole. The speedup obtained is determined by the fraction of the program code which is able to run in parallel. The sequential code will still run in the same time. So, it can not be assumed that if there are 2 processors in the system, the performance would definitely be doubled.

In conclusion, programs in the first place must contain significant number of algorithms that is able to process data in parallel in order to take advantage of the multiprocessor system.

Another challenge to multiprocessor design is the latency imposed when accessing the main memory, particularly with centralized shared-memory multiprocessor systems. As the number of microprocessors increase, the number of interconnection between the processors and the main memory has to be increased as well. This adds to the logic design and thus, introduces latency overall.

Questions

1.Explain multi level caches.

2.Explain challenges of parrallel processing.

3.Explain multiprocessor cache coherence.

4.Explain adaptive protocols.

References

http://www.cs.iastate.edu/~prabhu/Tutorial/title.html

http://www.rdrop.com/~cary/html/computer_architecture.html

Computer Architecture:A Quantitative Approach By David.a.patterson

Parrallel Computer Architecture:A Hardware/Software Approach By David Culler &J.p Singh

High Performance Computer Architecture By Harol Stone

REVISION

REVISION

REVISION

REVISION

PRE SEMESTER EXAM

PREPARATORY LEAVE

END TERM EXAMS