COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

INTRODUCTION TO PARALLEL ARCHITECTURE

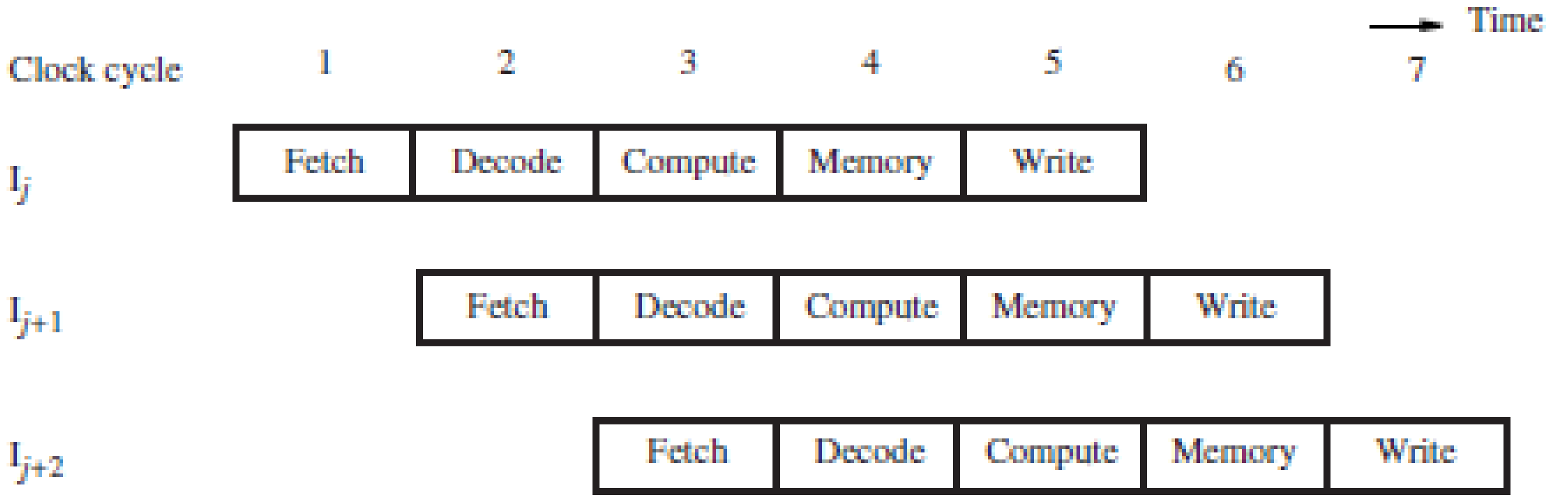
MODULE 7

PIPELINING: OVERVIEW

- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

PIPELINING: BASIC CONCEPTS

- Pipelined Execution:



Clock cycle	1	2	3	4	5	6	7
I_j	Fetch	Decode	Compute	Memory	Write		
I_{j+1}		Fetch	Decode	Compute	Memory	Write	
I_{j+2}			Fetch	Decode	Compute	Memory	Write

Figure 6.1 Pipelined execution—the ideal case.

PIPELINE ORGANIZATION

- The interstage buffers (for figures 5.7 and 5.8 from the textbook) are used as follows:
  - Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.
  - Interstage buffer B2 feeds the Compute stage with the operands read from the register file
  - Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage. In case of a write access to memory, buffer B3 holds the data to be written
  - Interstage buffer B4 feeds the Write stage with a value to be written into the register file.

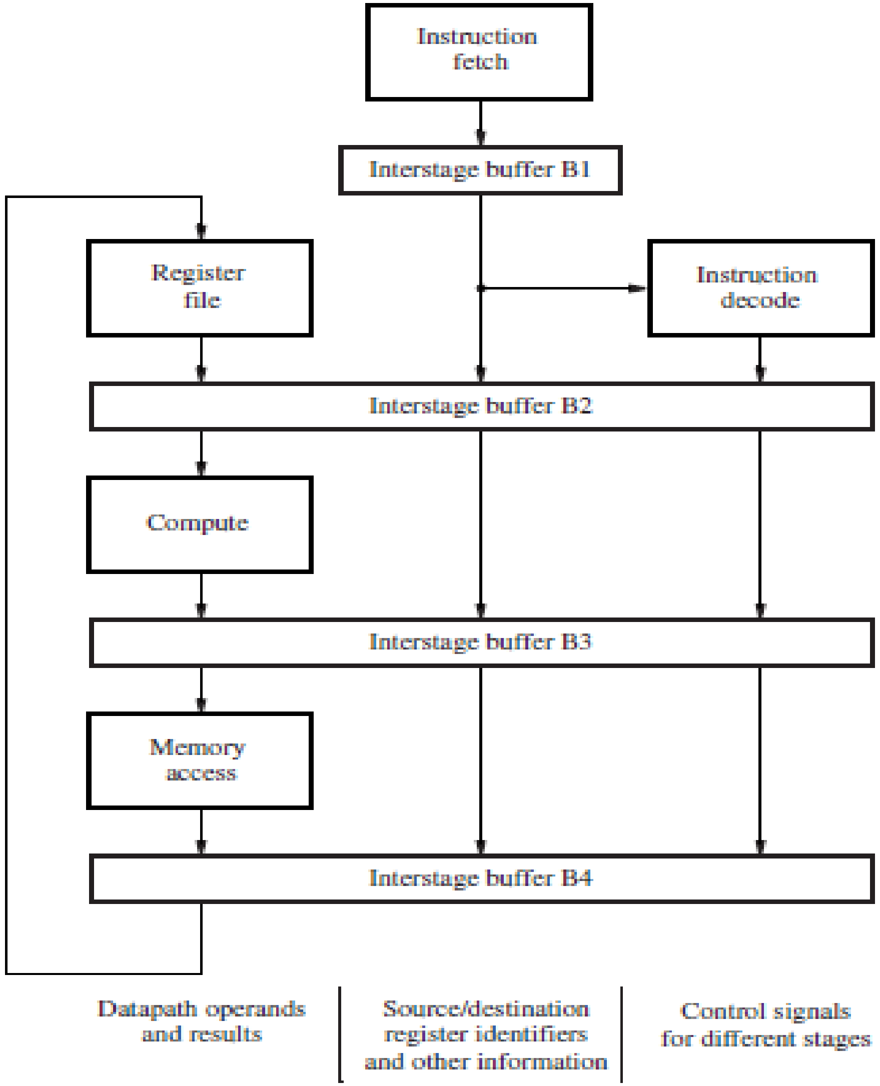


Figure 6.2 A five-stage pipeline.

PIPELINING ISSUES

- Consider the case of two instructions, I_j and I_{j+1}, where the destination register for instruction I_j is a source register for instruction I_{j+1}.
- To obtain the correct result it is necessary to wait until the new value is written into the register by instruction I_j .
- Hence, instruction I_{j+1} cannot read its operand until cycle 6, which means it must be stalled in the Decode stage for three cycles. While instruction I_{j+1} is stalled, instruction I_{j+2} and all subsequent instructions are similarly delayed. New instructions cannot enter the pipeline, and the total execution time is increased.

HAZARD

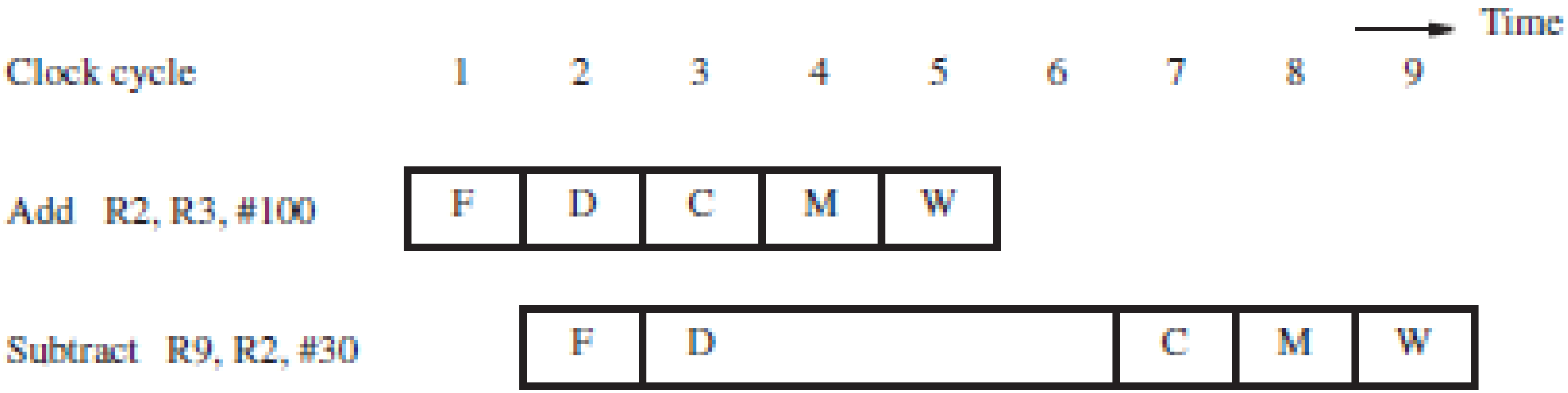
- Any condition that causes the pipeline to stall is called a hazard.
  - Data Hazard
  - Memory delays
  - Branch instructions

DATA HAZARDS

- Consider the two instructions in Figure 6.3:

  Add R2, R3, #100

  Subtract R9, R2, #30



Clock cycle	1	2	3	4	5	6	7	8	9
Add R2, R3, #100	F	D	C	M	W				
Subtract R9, R2, #30		F	D				C	M	W

Figure 6.3 Pipeline stall due to data dependency.

OPERAND FORWARDING

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.
- A special arrangement needs to be made to "forward" the output of ALU to the input of ALU.

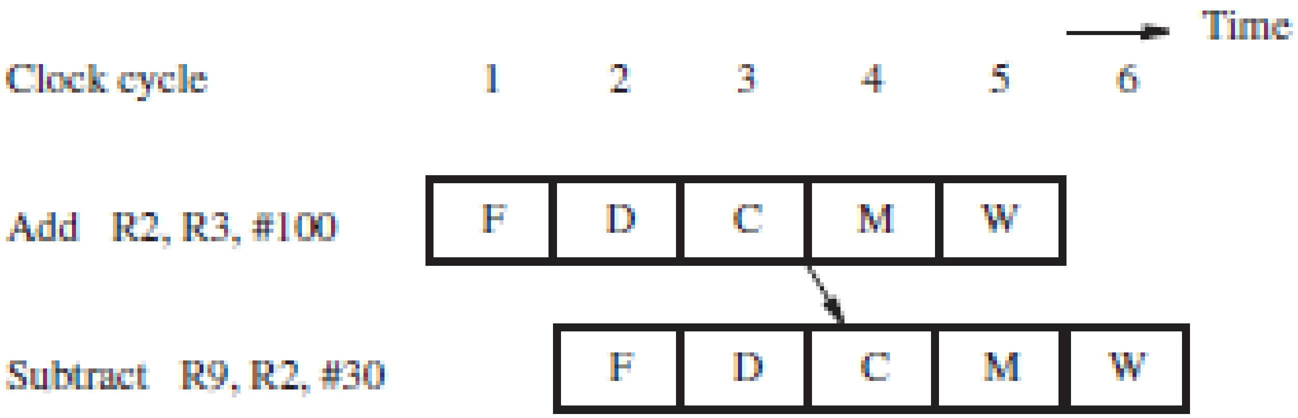


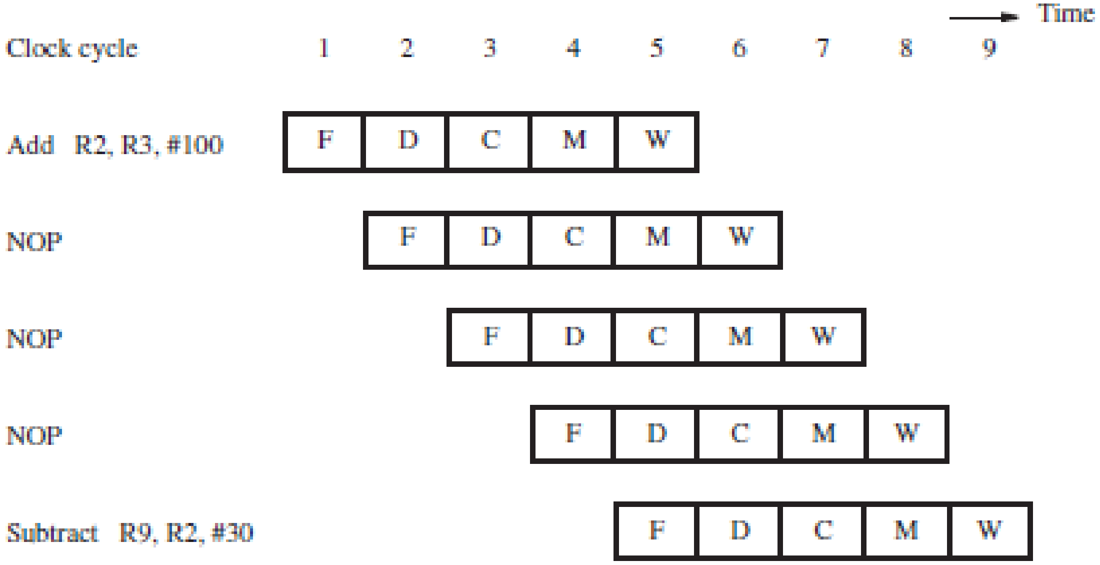
Figure 6.4 Avoiding a stall by using operand forwarding.

HANDLING DATA DEPENDENCIES IN SOFTWARE

- Compiler identifies a data dependency between two successive instructions I_j and I_{j+1}, it can insert three explicit NOP (No-operation) instructions between them.
- The NOPs introduce the necessary delay to enable instruction I_{j+1} to read the new value from the register file after it is written.
- The compiler can reorder the instructions to perform some useful work during the NOP slots

```
Add        R2, R3, #100
NOP
NOP
NOP
Subtract   R9, R2, #30
```

(a) Insertion of NOP instructions for a data dependency



Clock cycle	1	2	3	4	5	6	7	8	9
Add R2, R3, #100	F	D	C	M	W				
NOP		F	D	C	M	W			
NOP			F	D	C	M	W		
NOP				F	D	C	M	W	
Subtract R9, R2, #30					F	D	C	M	W

(b) Pipelined execution of instructions

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 6: 6.1- 6.4

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code	: CSE 2151
Credits	: 04

MEMORY DELAYS

- Delays arising from memory accesses are another cause of pipeline stalls.
- Load instruction may require more than one clock cycle to obtain its operand from memory.
- This may occur because the requested instruction or data are not found in the cache, resulting in a cache miss.
- A memory access may take ten or more cycles. For simplicity, the figure shows only three cycles.

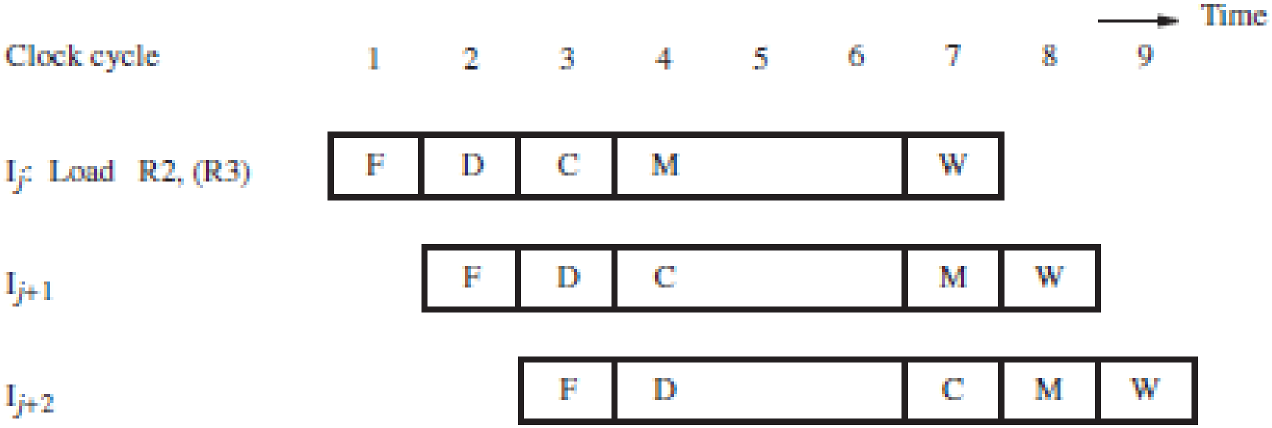


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

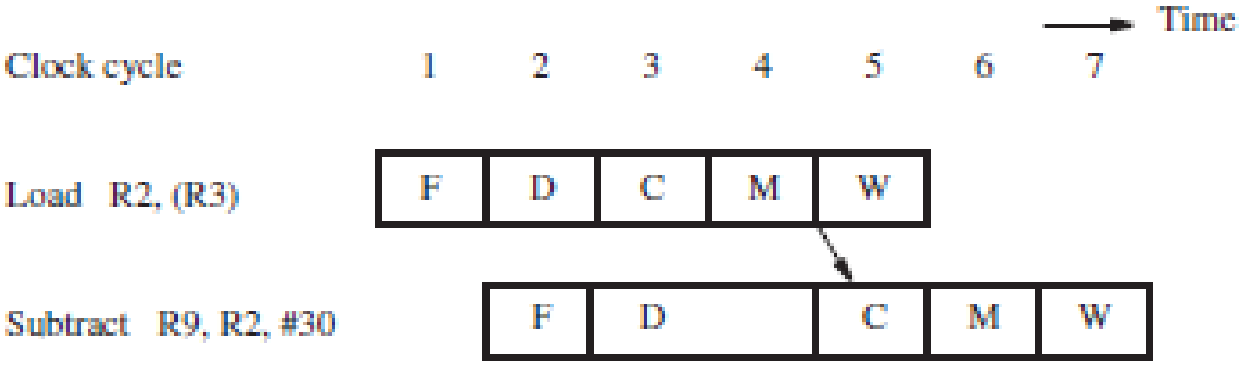
MEMORY DELAYS

- Consider the instructions:

  Load R2, (R3)

  Subtract R9, R2, #30

- Assume that the data for the Load instruction is found in the cache, requiring only one cycle to access the operand.
- The destination register R2 for the Load instruction is a source register for the Subtract instruction.
- Operand forwarding cannot be done because the data read from memory (the cache, in this case) are not available until they are loaded into register
- the Subtract instruction must be stalled for one cycle



Clock cycle	1	2	3	4	5	6	7
Load R2, (R3)	F	D	C	M	W		
Subtract R9, R2, #30		F	D		C	M	W

Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

INSTRUCTION HAZARDS: OVERVIEW

- Whenever the stream of instructions supplied by the instruction fetch unit is interrupted, the pipeline stalls.
  - Branch

UNCONDITIONAL BRANCH

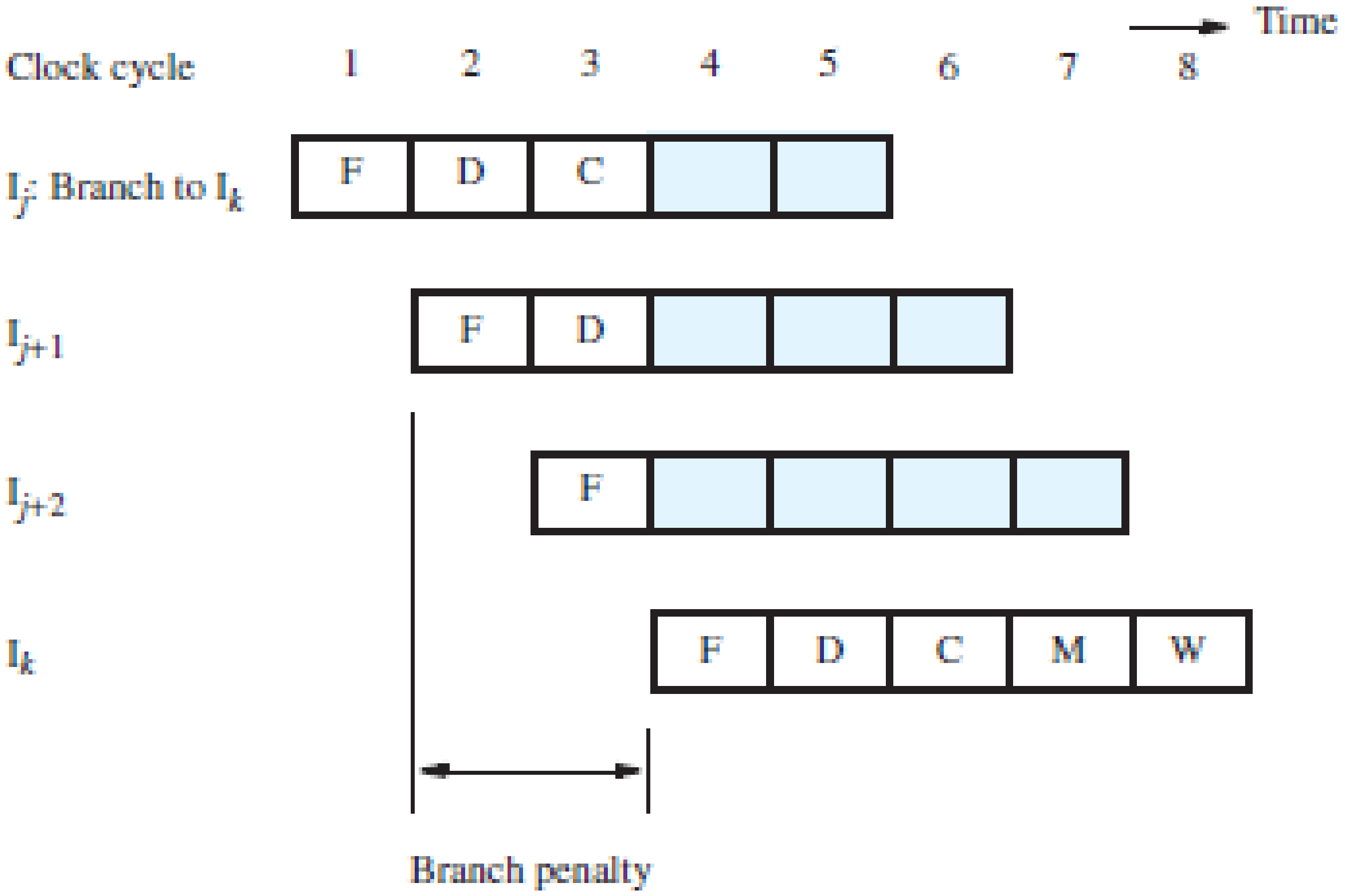


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

UNCONDITIONAL BRANCH

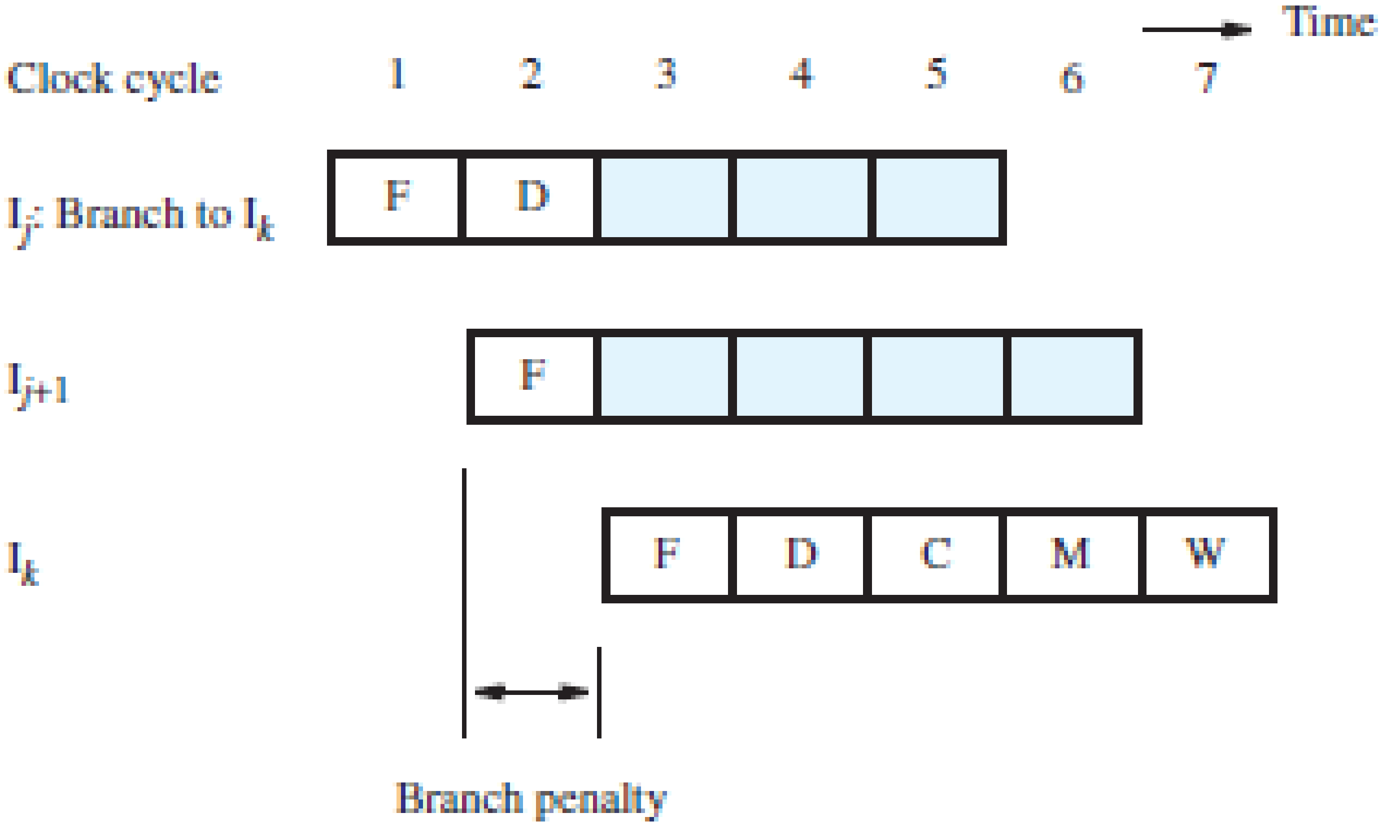


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

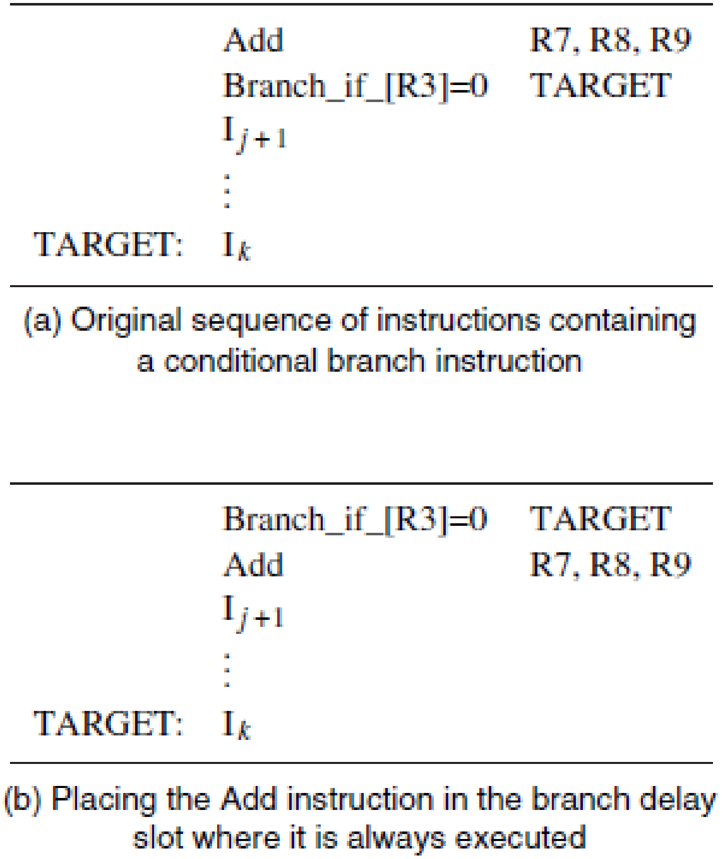
CONDITIONAL BRANCHES

Branch_if_[R5]=[R6] LOOP

- A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
- The result of the comparison determines whether the branch is taken
- The branch condition must be tested as early as possible to limit the branch penalty.
- The decision to branch cannot be made until the execution of that instruction has been completed.
- The comparator that tests the branch condition can also be moved to the Decode stage, enabling the conditional branch decision to be made at the same time that the target address is determined.
- In this case, the comparator uses the values from outputs A and B of the register file directly.

DELAYED BRANCH

- The location that follows a branch instruction is called the branch delay slot
- The instructions in the delay slots are always fetched. Therefore, arrange for them to be fully executed whether or not the branch is taken.
- Place useful instructions in these slots.
- The effectiveness depends on how often it is possible to reorder instructions.
- Branching takes place one instruction later than where the branch instruction appears in the instruction sequence. This technique is called delayed branching.



```
            Add                 R7, R8, R9
            Branch_if_[R3]=0    TARGET
            I_{j+1}
            :
TARGET:     I_k
```

(a) Original sequence of instructions containing a conditional branch instruction

```
            Branch_if_[R3]=0    TARGET
            Add                 R7, R8, R9
            I_{j+1}
            :
TARGET:     I_k
```

(b) Placing the Add instruction in the branch delay slot where it is always executed

Figure 6.11 Filling the branch delay slot with a useful instruction.

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 6: 6.5- 6.6

COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

PARALLEL PROCESSING

Multithreading

Vector Processing

multiprocessing

HARDWARE MULTITHREADING

- Operating system (OS) software enables multitasking of different programs in the same processor by performing context switches among programs
- A program, together with any information that describes its current state of execution, is called a process
- Information about the memory and other resources allocated by the OS is maintained with each process.
- Processes may be associated with applications such as Web-browsing, word-processing, and music-playing programs that a user has opened in a computer
- Each process has a corresponding thread, which is an independent path of execution within a program.

HARDWARE MULTITHREADING -THREADS

- The term thread is used to refer to a thread of control whose state consists of the contents of the program counter and other processor registers
- It is possible for multiple threads to execute portions of one program and run in parallel as if they correspond to separate programs.
- Two or more threads can be running on different processors, executing either the same part of a program on different data, or executing different parts of a program.
- Threads for different programs can also execute on different processors.
- All threads that are part of a single program run in the same address space and are associated with the same process
- We focus on multitasking where two or more programs run on the same processor and each program has a single thread
- OS selects a process among those that are not presently blocked and allows this process to run for a short period of time.
- Only the thread corresponding to the selected process is active during the time slice.

HARDWARE MULTITHREADING -THREADS

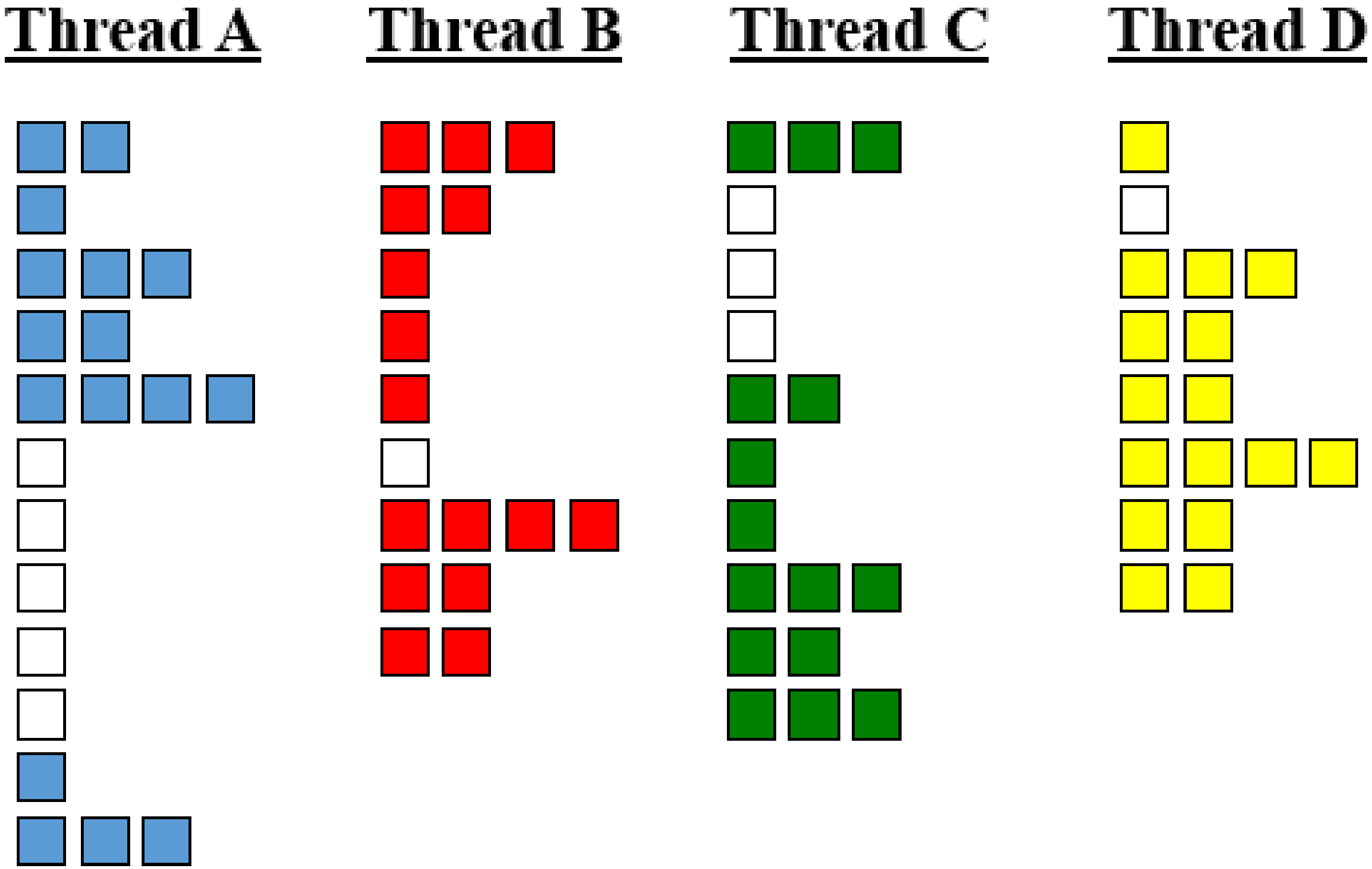
- Context switching at the end of the time slice causes the OS to select a different process, whose corresponding thread becomes active during the next time slice.
- A timer interrupt invokes an interrupt-service routine in the OS to switch from one process to another
- Processor is implemented with several identical sets of registers, including multiple program counters.
- Each set of registers can be dedicated to a different thread.
- Thus, no time is wasted during a context switch to save and restore register contents.
- The processor is said to be using a technique called hardware multithreading.
- With multiple sets of registers, context switching is simple and fast.
- All that is necessary is to change a hardware pointer in the processor to use a different set of registers to fetch and execute subsequent instructions.
- Switching to a different thread can be completed within one clock cycle.
- The state of the previously active thread is preserved in its own set of registers.

COARSE-GRAINED MULTITHREADING

- Switching to a different thread may be triggered at any time by the occurrence of a specific event, rather than at the end of a fixed time interval
- a cache miss may occur when a Load or Store instruction is being executed for the active thread. Instead of stalling while the slower main memory is accessed to service the cache miss, a processor can quickly switch to a different thread and continue to fetch and execute other instructions.
- This is called coarse-grained multithreading because many instructions may be executed for one thread before an event such as a cache miss causes a switch to another thread

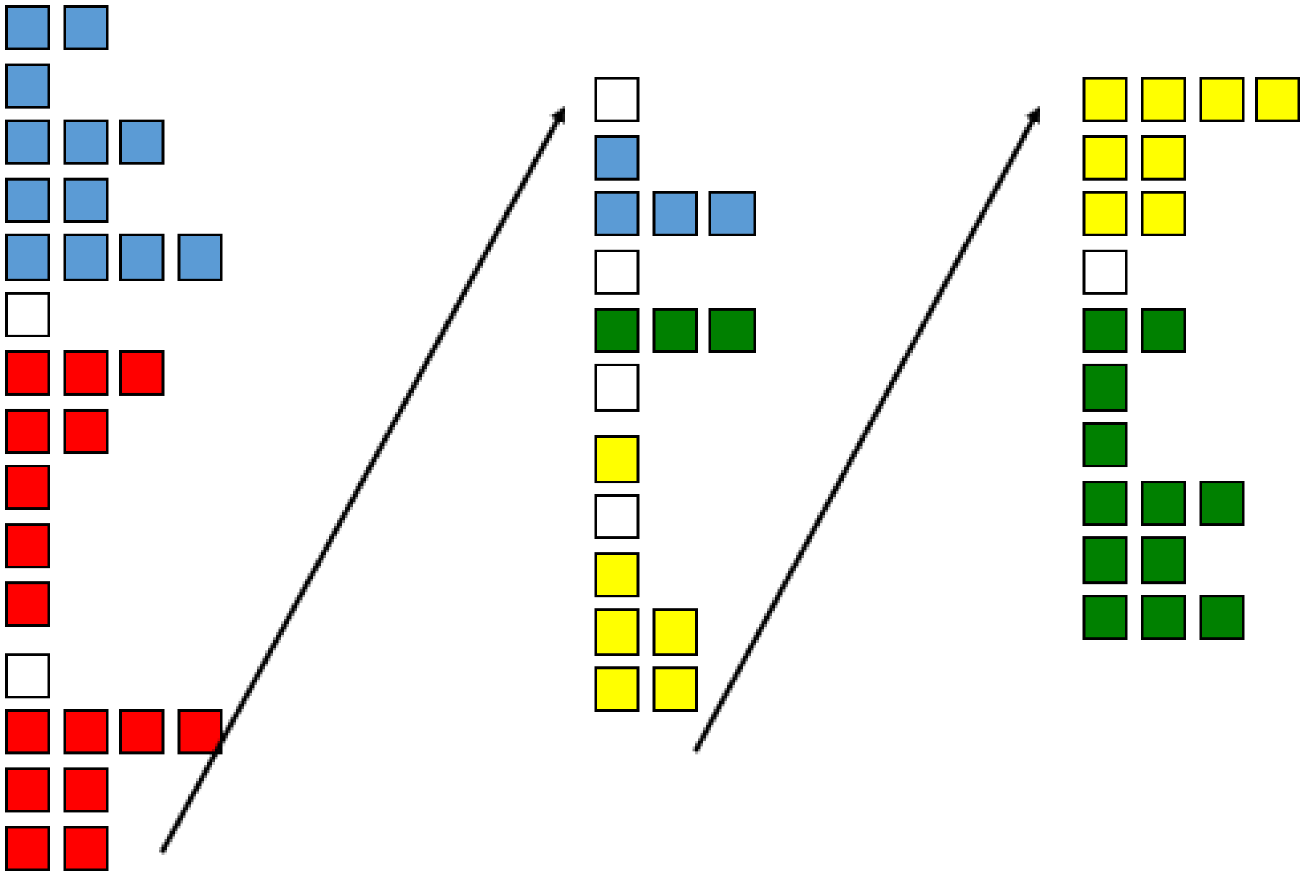
COARSE-GRAINED MULTITHREADING

- Conceptual Diagram



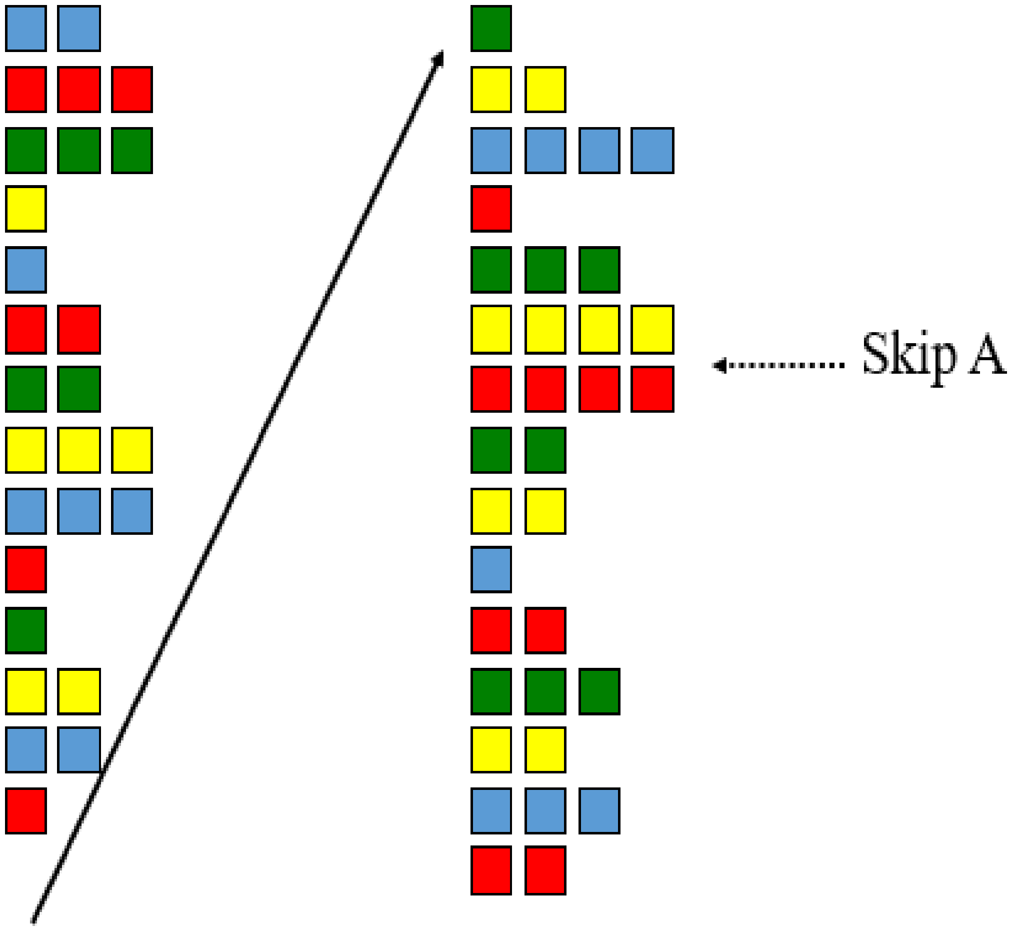
COARSE-GRAINED MULTITHREADING

- Stalls for A and C would be longer than indicated in previous slide
  Assume long stalls at end of each thread indicated in previous slide



FINE-GRAINED OR INTERLEAVED MULTITHREADING

- Switch after every instruction is fetched.
- This is called fine-grained or interleaved multithreading.
- The intent is to increase the processor throughput.
- Each new instruction is independent of its predecessors from other threads.
- This should reduce the occurrence of stalls due to data dependencies.
- Throughput is increased by interleaving instructions from many threads
- It takes longer for a given thread to complete all of its instructions.



VECTOR (SIMD)PROCESSING

- programs use loops to perform operations on vectors of data, where a vector is an array of elements such as integers or floating-point numbers.
- When a processor executes the instructions in such a loop, the operations are performed one at a time on individual vector elements.
- Many instructions need to be executed to process all vector elements.
- A processor can be enhanced with multiple ALUs.
- It is possible to operate on multiple data elements in parallel using a single instruction.
- Such instructions are called single-instruction multiple-data (SIMD) instructions. They are also called vector instructions.
- These instructions can only be used when the operations performed in parallel are independent. This is known as data parallelism.
- The data for vector instructions are held in vector registers, each of which can hold several data elements. The number of elements, L, in each vector register is called the vector length.
- It determines the number of operations that can be performed in parallel on multiple ALUs.

VECTOR (SIMD)PROCESSING

- The vector instruction

  VectorAdd.S Vi, Vj, Vk

- computes L sums using the elements in vector registers Vj and Vk, and places the resulting sums in vector register Vi.
- Suffix S denotes the size of each data element
- Special instructions are needed to transfer multiple data elements between a vector register and the memory. The instruction

  VectorLoad.S Vi, X(Rj)

- causes L consecutive elements beginning at memory location X + [Rj] to be loaded into vector register Vi. Similarly, the instruction

  VectorStore.S Vi, X(Rj)

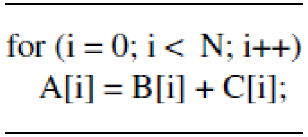
- causes the contents of vector register Vi to be stored as L consecutive locations in the memory.

VECTORIZATION

- In a source program written in a high-level language, loops that operate on arrays of integers or floating-point numbers are vectorizable if the operations performed in each pass are independent of the other passes.
- Using vector instructions reduces the number of instructions that need to be executed
- Enables the operations to be performed in parallel on multiple ALUs.
- A vectorizing compiler can recognize such loops, if they are not too complex, and generate vector instructions.

VECTORIZATION EXAMPLE

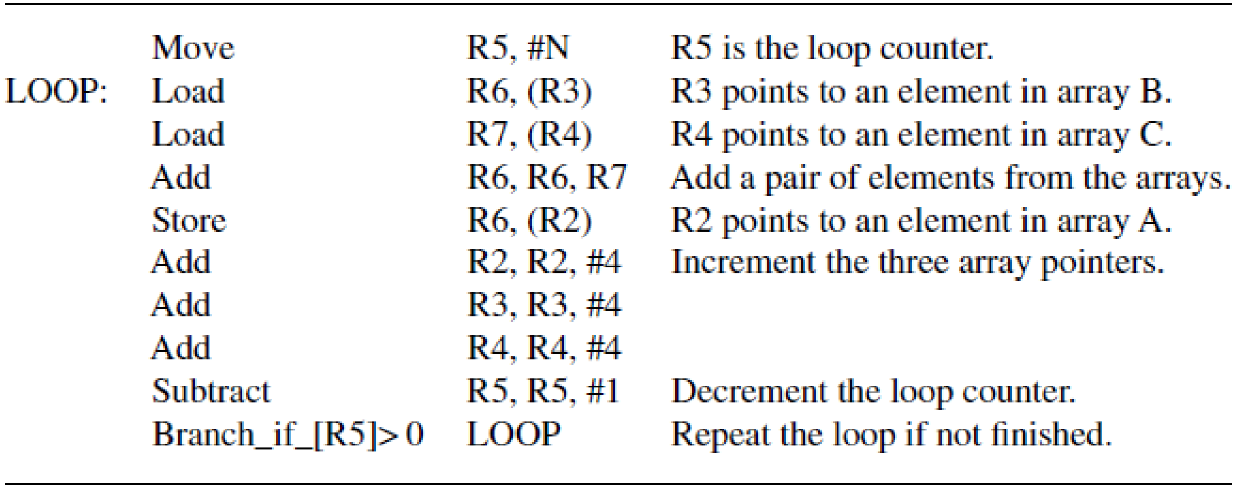
- Consider vectorization of the loop given below



```
for (i = 0; i <  N; i++)
   A[i] = B[i] + C[i];
```

(a) A C-language loop to add vector elements

- Assume that the starting locations in memory for arrays A, B, and C are in registers R2, R3, and R4. Using conventional assembly-language instructions, the compiler may generate the loop.



```
          Move                R5, #N        R5 is the loop counter.
LOOP:     Load                R6, (R3)      R3 points to an element in array B.
          Load                R7, (R4)      R4 points to an element in array C.
          Add                 R6, R6, R7    Add a pair of elements from the arrays.
          Store               R6, (R2)      R2 points to an element in array A.
          Add                 R2, R2, #4    Increment the three array pointers.
          Add                 R3, R3, #4
          Add                 R4, R4, #4
          Subtract            R5, R5, #1    Decrement the loop counter.
          Branch_if_[R5]> 0   LOOP          Repeat the loop if not finished.
```

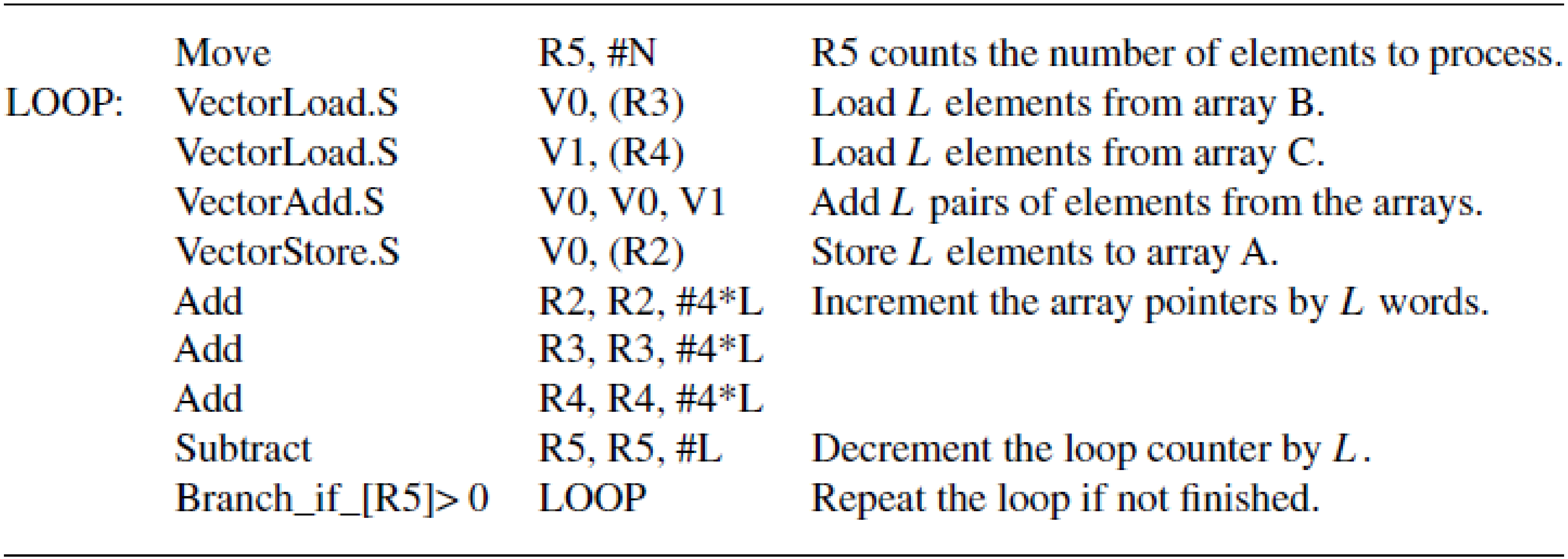
(b) Assembly-language instructions for the loop

VECTORIZATION EXAMPLE CONTD..

- The Load, Add, and Store instructions at the beginning of the loop are replaced by corresponding vector instructions that operate on L elements at a time.
- The vectorized loop requires only N/L passes to process all of the data in the arrays.
- With L elements processed in each pass through the loop, the address pointers in registers R2, R3, and R4 are incremented by 4L, and the count in register R5 is decremented by L.

VECTORIZATION EXAMPLE CONTD..

- Vectorized form of the loop



```
            Move                 R5, #N          R5 counts the number of elements to process.
LOOP:       VectorLoad.S         V0, (R3)        Load L elements from array B.
            VectorLoad.S         V1, (R4)        Load L elements from array C.
            VectorAdd.S          V0, V0, V1      Add L pairs of elements from the arrays.
            VectorStore.S        V0, (R2)        Store L elements to array A.
            Add                  R2, R2, #4*L    Increment the array pointers by L words.
            Add                  R3, R3, #4*L
            Add                  R4, R4, #4*L
            Subtract             R5, R5, #L      Decrement the loop counter by L.
            Branch_if_[R5]> 0    LOOP            Repeat the loop if not finished.
```

(c) Vectorized form of the loop

GPU

- specialized chips called Graphics Processing Units (GPUs).
- to accelerate the large number of floating-point calculations needed in high-resolution, three-dimensional graphics, such as in video games.
- a large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel.
- A small program is written for the processing cores in the GPU chip.
- A large number of cores execute this program in parallel.
- The cores execute the same instructions but operate on different data elements.
- A separate controlling program runs in the general-purpose processor of the host computer and invokes the GPU program when necessary.
- Before initiating the GPU computation, the program in the host computer must first transfer the data needed by the GPU program from the main memory into the dedicated GPU memory.
- After the computation is completed, the resulting output data in the dedicated memory are transferred back to the main memory.

GPU

- The processing cores in a GPU chip have a specialized instruction set and hardware architecture, which are different from those used in a general-purpose processor.
  - e.g Compute Unified Device Architecture (CUDA) that NVIDIA Corporation uses for the cores in its GPU chips
- To facilitate writing programs that involve a general-purpose processor and a GPU, an extension to the C programming language, called CUDA C, has been developed by NVIDIA
- This extension enables a single program to be written in C, with special keywords used to label the functions executed by the processing cores in a GPU chip.
- The compiler and related software tools automatically partition the final object program into the portions that are translated into machine instructions for the host computer and the GPU chip.
- Library routines are provided to allocate storage in the dedicated memory of a GPU-based video card and to transfer data between the main memory and the dedicated memory.
- An open standard called OpenCL has also been proposed by industry as a programming framework for systems that include GPU chips from any vendor

SHARED-MEMORY MULTIPROCESSORS

- All processors have access to the same memory.
- Tasks running in different processors can access shared variables in the memory using the same addresses.
- The size of the shared memory is likely to be large.
- bottleneck when many processors make requests to access the memory simultaneously.
- Distributed across multiple modules so that simultaneous requests from different processors are more likely to access different memory modules, depending on the addresses of those requests.

SHARED-MEMORY MULTIPROCESSORS-UMA

- Interconnection networks-enables any processor to access any module that is a part of the shared memory.
- All requests to access memory must pass through the network, which introduces latency.
- A system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access (UMA) multiprocessor.

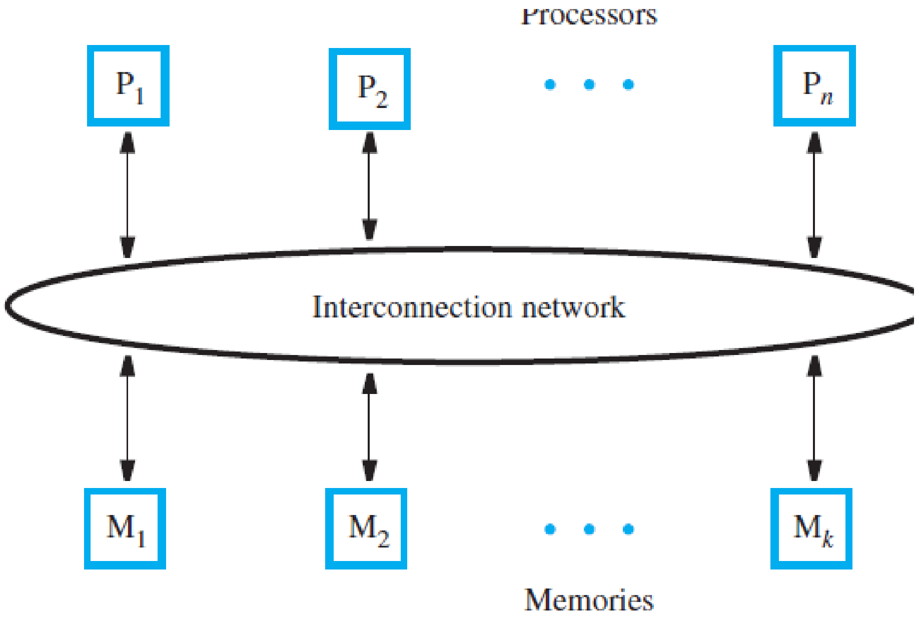


Figure 12.2 A UMA multiprocessor.

NUMA

- It is desirable to place a memory module close to each processor.
- The result is a collection of nodes, each consisting of a processor and a memory module.
- Nodes are then connected to the network, as shown in Figure 12.3.
- The n/w latency is avoided when a processor makes a request to access its local memory.
- a request to access a remote memory module must pass through the n/w
- Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called Non-Uniform Memory Access (NUMA) multiprocessors.

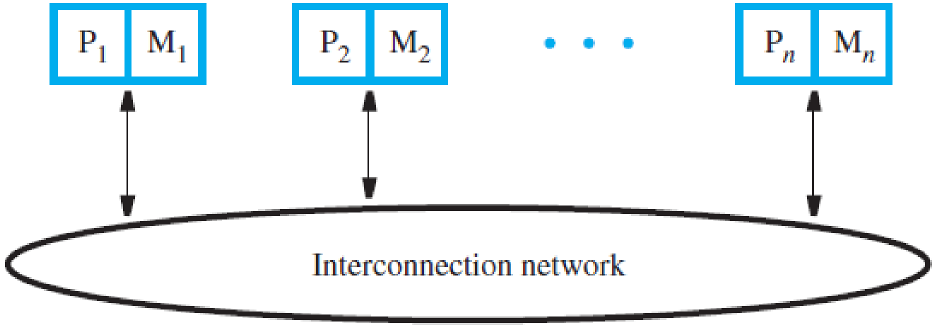


Figure 12.3 A NUMA multiprocessor.

INTERCONNECTION NETWORKS

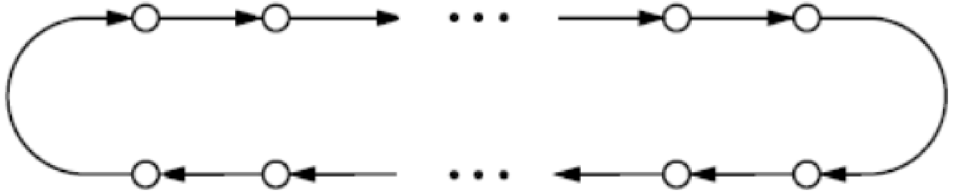
- must allow information transfer between any pair of nodes in the system.
- may also be used to broadcast information from one node to many other nodes.
- The traffic in the network consists of requests (such as read and write) and data transfers.
- The suitability of a particular network is judged in terms of cost, bandwidth, effective throughput, and ease of implementation.
- The term bandwidth refers to the capacity of a transmission link to transfer data and is expressed in bits or bytes per second.
- The effective throughput is the actual rate of data transfer.
  - This rate is less than the available bandwidth because a given link must also carry control information that coordinates the transfer of data

BUS

- Set of lines (wires) that provide a single shared path for information transfer
- Buses are most commonly used in UMA multiprocessors to connect a number of processors to several shared-memory modules.
- Arbitration is necessary to ensure that only one of many possible requesters is granted use of the bus at any time.
- The bus is suitable for a relatively small number of processors
- contention for access to the bus
- increased propagation delays when many processors are connected.

INTERCONNECTION NETWORKS-RING BASED

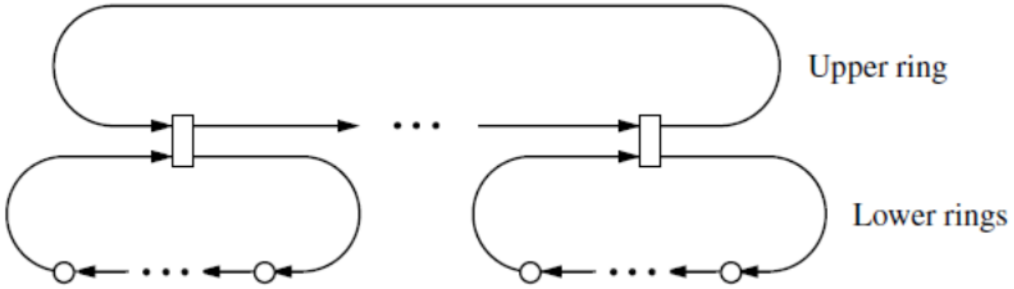
- A ring network is formed with point-to-point connections between nodes
- A long single ring results in high average latency for communication between any two nodes.
- This high latency can be mitigated in two different ways.
- A second ring can be added to connect the nodes in the opposite direction. The resulting bidirectional ring halves the average latency and doubles the bandwidth. However, handling of communications is more complex



(a) Single ring

HIERARCHY OF RINGS

- A two-level hierarchy is shown
- The upper-level ring connects the lower-level rings. The average latency for communication between any two nodes on lower-level rings is reduced with this arrangement.
- Transfers between nodes on the same lower-level ring need not traverse the upper-level ring.
- Transfers between nodes on different lower-level rings include a traversal on part of the upper-level ring.
- The drawback of the hierarchical scheme is that the upper-level ring may become a bottleneck when many nodes on different lower-level rings communicate with each other frequently.



(b) Hierarchy of rings

CROSSBAR NETWORK

- A crossbar is a network that provides a direct link between any pair of units connected to the network.
- It is typically used in UMA multiprocessors to connect processors to memory modules.
- It enables many simultaneous transfers if the same destination is not the target of multiple requests.

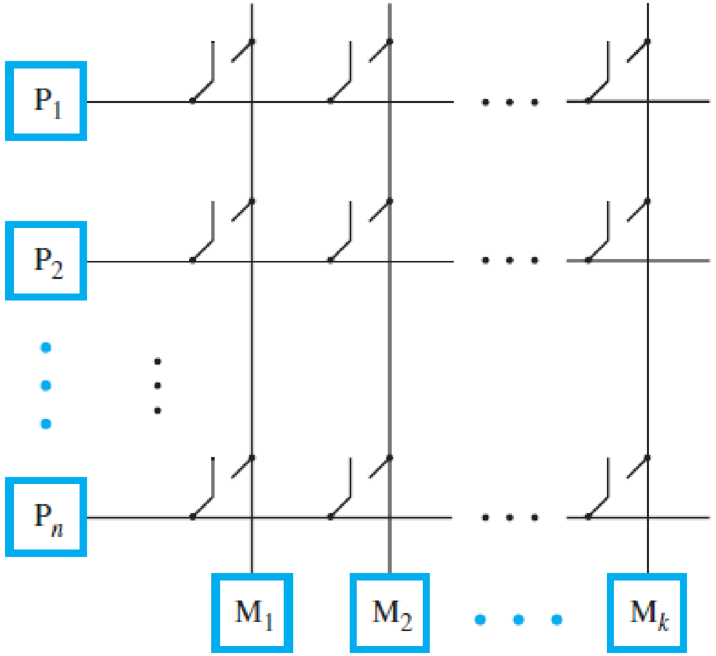


Figure 12.5 Crossbar interconnection network.

MESH

- A natural way of connecting a large number of nodes is with a two-dimensional mesh
- Each internal node of the mesh has four connections, one to each of its horizontal and vertical neighbors.
- Nodes on the boundaries and corners of the mesh have fewer neighbors and hence fewer connections.
- To reduce latency for communication between nodes that would otherwise be far apart in the mesh, wraparound connections may be introduced between nodes at opposite boundaries of the mesh.
- A network with such connections is called a **torus.**
- All nodes in a torus have four connections. Average latency is reduced, but the implementation complexity for routing requests and responses through a torus is somewhat higher than in the case of a simple mesh

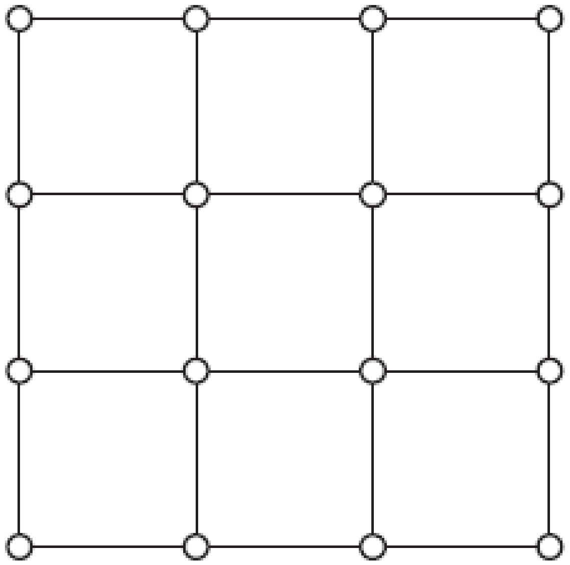


Figure 12.6 A two-dimensional mesh network.

CACHE COHERENCE

- Possibility that copies of shared data may reside in several caches.
- When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value.
- They must be informed of the change so that they can either update their copy to the new value or invalidate it.
- This is the issue of maintaining cache coherence, which requires having a consistent view of shared data in multiple caches.

WRITE-THROUGH PROTOCOL

- updating the values in other caches
  - When a processor writes a new value to block of data in its cache, the new value is also written into the memory module containing the block being modified.
  - Since copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the Write operation. The simplest way
    - broadcast the written data to the caches of all processors in the system.
  - As each processor receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache
- invalidating the copies in other caches
  - When a processor writes a new value into its cache, this value is also sent to the appropriate location in memory, and all copies in other caches are invalidated.
  - broadcasting can be used to send the invalidation requests throughout the system

WRITE-BACK PROTOCOL

- Based on the concept of ownership of a block of data in the memory.
- Initially, the memory is the owner of all blocks, and the memory retains ownership of any block that is read by a processor to place a copy in its cache.
- If some processor wants to write to a block in its cache, it must first become the exclusive owner of this block. To do so, all copies in other caches must first be invalidated with a broadcast request.
- The new owner of the block may then modify the contents at will without having to take any other action
- When another processor wishes to read a block that has been modified, the request for the block must be forwarded to the current owner. The data are then sent to the requesting processor by the current owner.
- The data are also sent to the appropriate memory module, which reacquires ownership and updates the contents of the block in the memory.
- The cache of the processor that was the previous owner retains a copy of the block. Hence, the block is now shared with copies in two caches and the memory.
- Subsequent requests from other processors to read the same block are serviced by the memory module containing the block.

SNOOPY CACHES

- In a single-bus system, all transactions between processors and memory modules occur via requests and responses on the bus.
- They are broadcast to all units connected to the bus.
- Suppose that each processor cache has a controller circuit that observes, or snoops, all transactions on the bus.
- scenarios for the write-back protocol

SCENARIO 1

- Consider a processor that has previously read a copy of a block from the memory into its cache.
- Before writing to this block for the first time, the processor must broadcast an invalidation request to all other caches, whose controllers accept the request and invalidate any copies of the same block.
- This action causes the requesting processor to become the new owner of the block.
- The processor may then write to the block and mark it as being modified.
- No further broadcasts are needed from the same processor to write to the modified block in its cache.
- if another processor broadcasts a read request on the bus for the same block, the memory must not respond because it is not the current owner of the block.

SCENARIO 1

- The processor owning the requested block snoops the read request on the bus. Because it holds a modified copy of the requested block in its cache, it asserts a special signal on the bus to prevent the memory from responding.
- The owner then broadcasts a copy of the block on the bus and marks its copy as clean (unmodified).
- The data response on the bus is accepted by the cache of the processor that issued the read request.
- The data response is also accepted by the memory to update its copy of the block. In this case, the memory reacquires ownership of the block, and the block is said to be in a shared state because copies of it are in the caches of two processors.
- Coherence is maintained because the two cached copies and the copy of the block in the memory contain the same data.
- Subsequent requests from any processor are serviced by the memory.

SCENARIO 2

- Two processors have copies of the same block in their respective caches, and both processors attempt to write to the same cache block at the same time.
- Since the block is in the shared state, the memory is the owner of the block.
- Both processors request the use of the bus to broadcast an invalidation message.
- One of the processors is granted the use of the bus first. That processor broadcasts its invalidation request and becomes the new owner of the block.
- Through snooping, the copy of the block in the cache of the other processor is invalidated.
- When the other processor is later granted the use of the bus, it broadcasts a read-exclusive request. This request combines a read request and an invalidation request for the same block.
- The controller for the first processor snoops the read-exclusive request, provides a data response on the bus, and invalidates the copy in its cache.
- Ownership of the block is therefore transferred to the second processor making the request.
- The memory is not updated because the block is being modified again.
- Since the requests from the two processors are handled sequentially, cache coherence is always maintained.
- This scheme is based on the ability of cache controllers to observe the activity on the bus and take appropriate actions. Such schemes are called snoopy-cache techniques

PERFORMANCE

- the snooping function should not interfere with the normal operation of a processor and its cache.
- Such interference occurs if the cache controller accesses the tags of the cache for every request that appears on the bus.
- In most cases, the cache would not contain a valid copy of the block that is relevant to a request.
- To eliminate unnecessary interference, each cache can be provided with a set of duplicate tags, which maintain the same status information about the blocks in the cache but can be accessed separately by the snooping circuitry.

DIRECTORY-BASED CACHE COHERENCE

- In systems using interconnection networks such as rings and meshes broadcasting every single request to the caches of all processors is inefficient
- A scalable, but more complex, solution to this problem uses directories in each memory module to indicate which nodes may have copies of a given block in the shared state.
- If a block is modified, the directory identifies the node that is the current owner.
- Each request from a processor must be sent first to the memory module containing the relevant block.
- The directory information for that block is used to determine the action that is taken.
- A read request is forwarded to the current owner if the block is modified. In the case of a write request for a block that is shared, individual invalidations are sent only to nodes that may have copies of the block in question.
- The cost and complexity of the directory-based approach for enforcing cache coherence limits its use to large systems.
- Small multiprocessors, including current multicore chips, use snooping.

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 12: 12.1- 12.4