UNIT 6 – INTRODUCTION TO PARALLEL ARCHITECTURE

Pipelining

The speed of execution of programs is influenced by many factors:

- One way to improve performance is to use faster circuit technology to implement the processor and the main memory
- Another possibility is to arrange the hardware so that more than one operation can be performed at the same time
- In this way, the number of operations performed per second is increased, even though the time needed to perform any one operation is not changed.

Pipelining

- Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
- A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned.
- The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

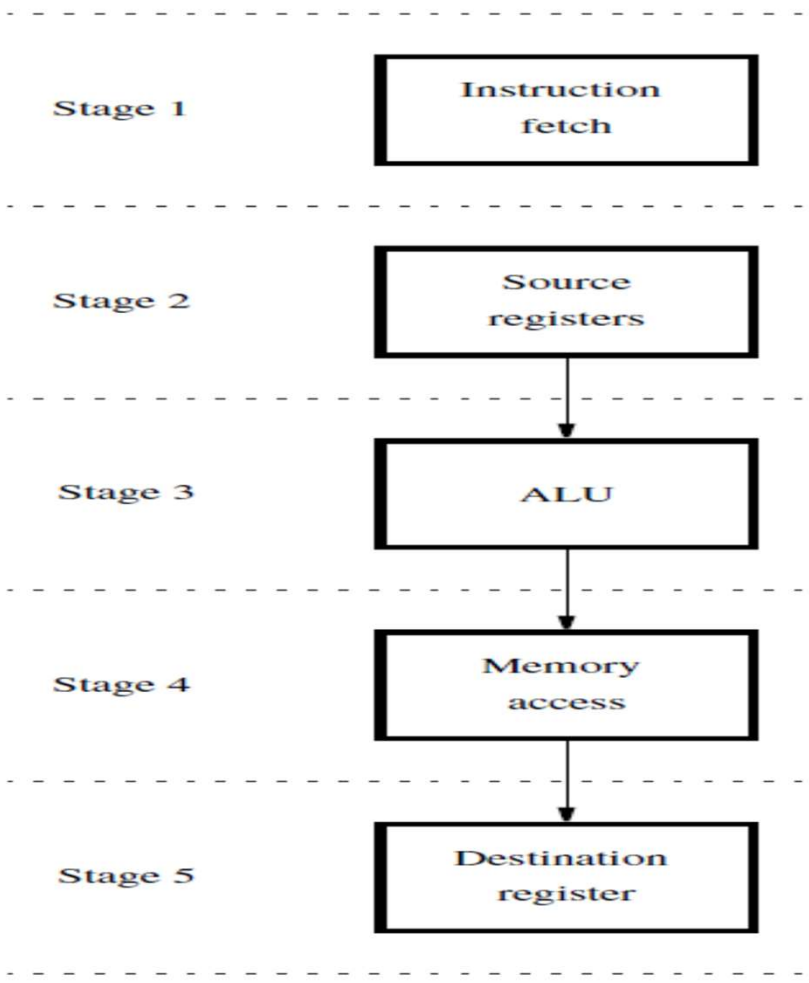


Figure 6.1 Five Stage Organization

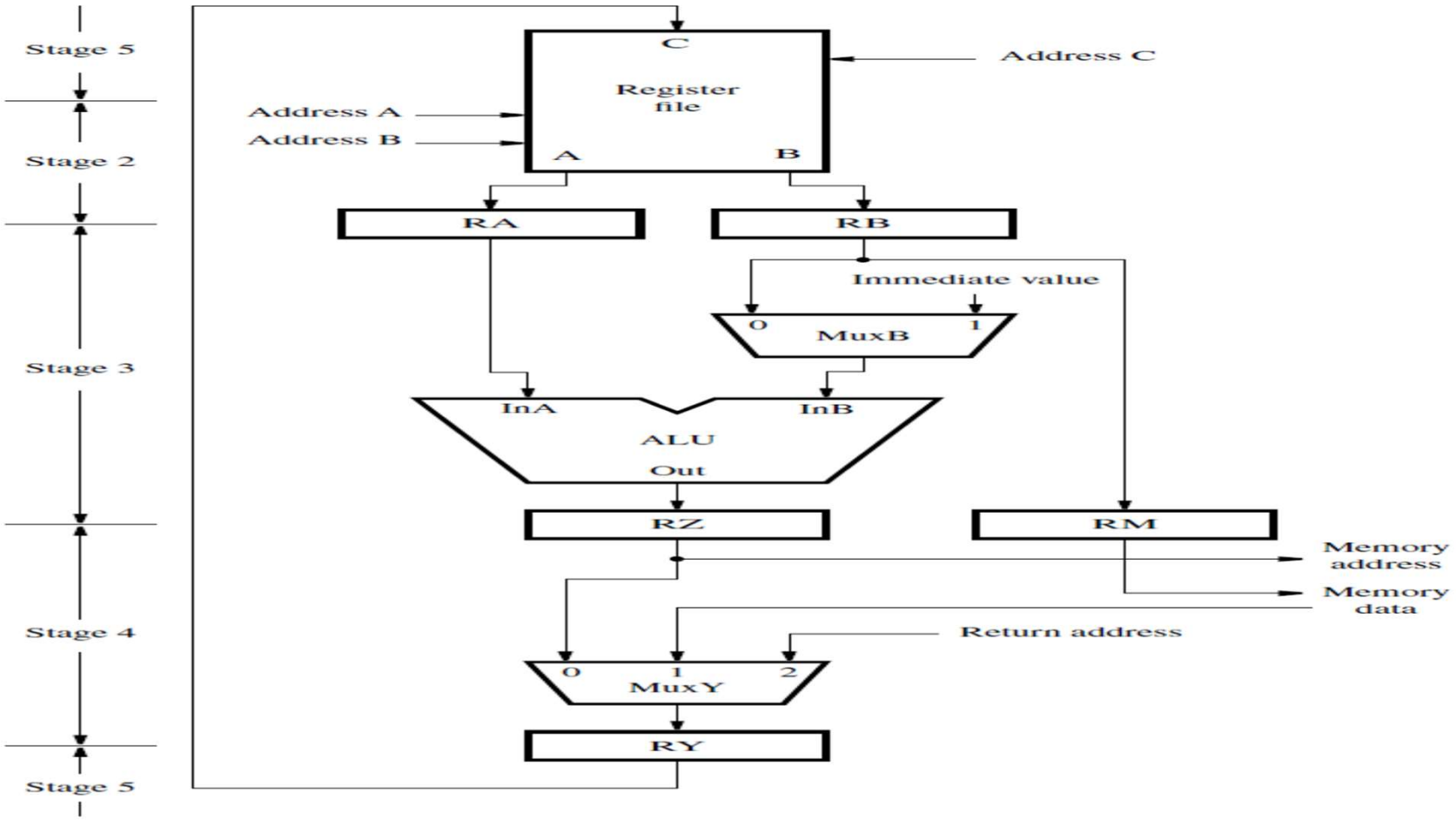


Figure 6.2 Datapath in a Processor

Pipelining

- The earlier discussed processor organization as shown in the figure 6.1 and 6.2 allow instructions to be fetched and executed one at a time.
- In the pipelining approach, rather than waiting until each instruction is completed, instructions can be fetched and executed in a pipelined manner.
- Instruction I_j is fetched in the first cycle and moves through the remaining stages in the following cycles
- In the second cycle, instruction I_{j+1} is fetched while instruction I_j is in the Decode stage where its operands are also read from the register file
- In the third cycle, instruction I_{j+2} is fetched while instruction I_{j+1} is in the Decode stage and instruction I_j is in the Compute stage where an arithmetic or logic operation is performed on its operands
- Ideally, this overlapping pattern of execution would be possible for all instructions.

Pipelining

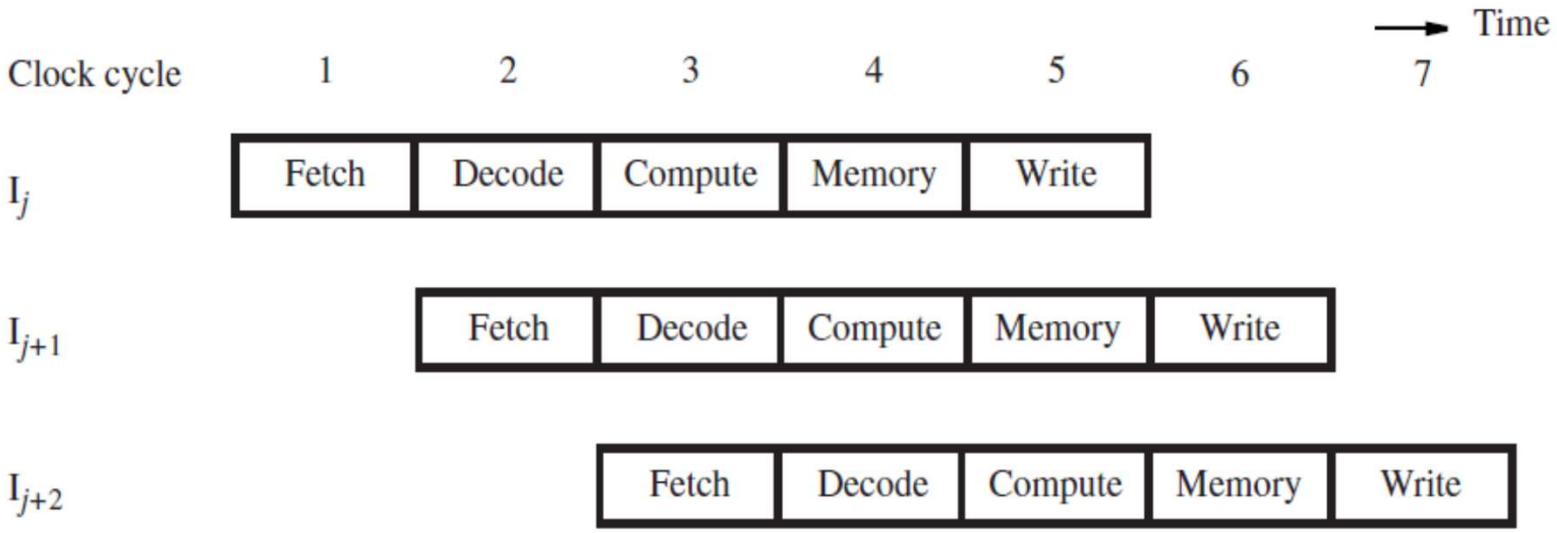


Figure 6.3 Pipelined Execution

Pipeline Organization

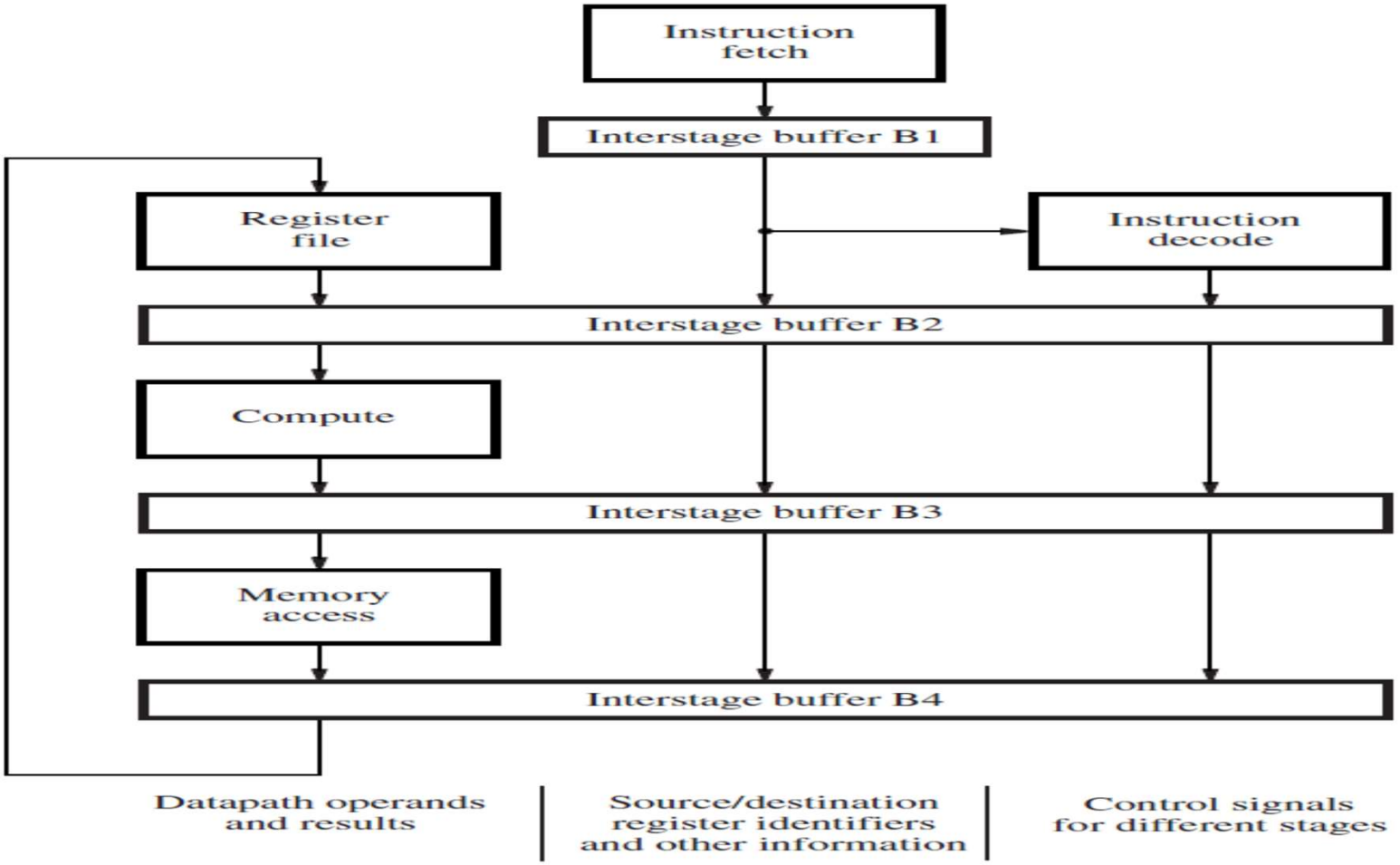


Figure 6.4 A five-stage Pipeline

Pipeline Organization

- In the first stage of the pipeline, the program counter (PC) is used to fetch a new instruction.
- As other instructions are fetched, execution proceeds through successive stages.
- At any given time, each stage of the pipeline is processing a different instruction
- Information such as register addresses, immediate data, and the operations to be performed must be carried through the pipeline as each instruction proceeds from one stage to the next
- This information is held in interstage buffers.
- These include registers RA, RB, RM, RY, and RZ, the IR and PC-Temp registers and additional storage.

Pipeline Organization

- The interstage buffers 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 two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder
- The settings for control signals move through the pipeline to determine the ALU operation, the memory operation, and a possible write into the register file.

Pipeline Organization

- 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 the case of a write access to memory, buffer B3 holds the data to be written
- These data were read from the register file in the Decode stage
- The buffer also holds the incremented PC value passed from the previous stage, in case it is needed as the return address for a subroutine-call instruction.
- Interstage buffer B4 feeds the Write stage with a value to be written into the register file
- This value may be the ALU result from the Compute stage, the result of the Memory access stage, or the incremented PC value that is used as the return address for a subroutine-call instruction.

Pipelining Issues

- Figure 6.3 depicts the ideal overlap of three successive instructions.
- But, there are times when it is not possible to have a new instruction enter the pipeline in every cycle
- 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}
- The result of instruction I_j is not written into the register file until cycle 5, but it is needed earlier in cycle 3 when the source operand is read for instruction I_{j+1}
- If execution proceeds as shown in Figure 6.1, the result of instruction I_{j+1} would be incorrect because the arithmetic operation would be performed using the old value of the register in question
- To obtain the correct result, it is necessary to wait until the new value is written into the register by instruction I_j

Pipelining Issues

- 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.

Pipelining Issues

- Any condition that causes the pipeline to stall is called a *hazard*
- We have just described an example of a *data hazard*, where the value of a source operand of an instruction is not available when needed
- Other hazards arise from memory delays, branch instructions, and resource limitations
- The next several sections describe these hazards in more detail, along with techniques to mitigate their impact on performance

Data Dependencies

Consider the two instructions in Figure 6.5:

Add R2, R3, #100

Subtract R9, R2, #30

- The destination register R2 for the Add instruction is a source register for the Subtract instruction
- There is a *data dependency* between these two instructions, because register R2 carries data from the first instruction to the second
- Pipelined execution of these two instructions is depicted in Figure 6.5
- The Subtract instruction is stalled for three cycles to delay reading register R2 until cycle 6 when the new value becomes available.

Data Dependencies

- We now explain the stall in more detail.
- The control circuit must first recognize the data dependency when it decodes the Subtract instruction in cycle 3 by comparing its source register identifier from interstage buffer B1 with the destination register identifier of the Add instruction that is held in interstage buffer B2.
- Then, the Subtract instruction must be held in interstage buffer B1 during cycles 3 to 5
- . Meanwhile, the Add instruction proceeds through the remaining pipeline stages
- In cycles 3 to 5, as the Add instruction moves ahead, control signals can be set in interstage buffer B2 for an implicit NOP (No-operation) instruction that does not modify the memory or the register file.
- Each NOP creates one clock cycle of idle time, called a *bubble*, as it passes through the Compute, Memory, and Write stages to the end of the pipeline

Data Dependencies

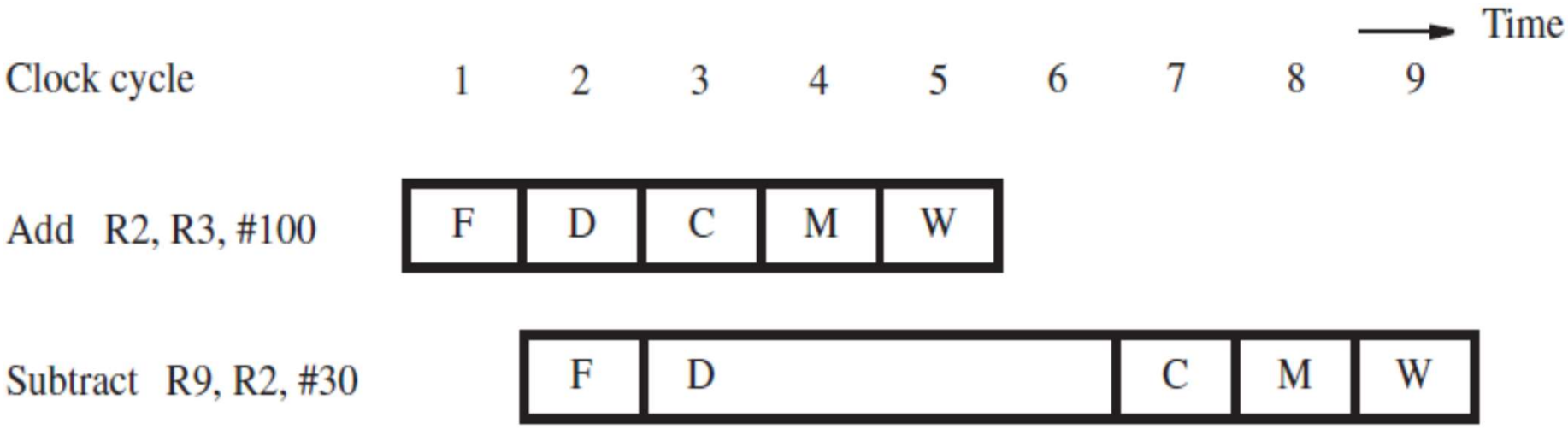


Figure 6.5 Pipeline Stall due to data independency

Operand Forwarding

- Pipeline stalls due to data dependencies can be alleviated through the use of operand forwarding.
- Consider the pair of instructions discussed above, where the pipeline is stalled for three cycles to enable the Subtract instruction to use the new value in register R2
- The desired value is actually available at the end of cycle 3, when the ALU completes the operation for the Add instruction
- This value is loaded into register RZ in Figure 6.2, which is a part of interstage buffer B3
- Rather than stall the Subtract instruction, the hardware can forward the value from register RZ to where it is needed in cycle 4, which is the ALU input. Figure 6.6 shows pipelined execution when forwarding is implemented
- The arrow shows that the ALU result from cycle 3 is used as an input to the ALU in cycle 4.

Operand Forwarding

- Figure 6.7 shows the modification needed in the datapath of Figure 5.8 to make this forwarding possible
- A new multiplexer, MuxA, is inserted before input InA of the ALU,and the existing multiplexer MuxB is expanded with another input
- The multiplexers select either a value read from the register file in the normal manner, or the value available in register RZ

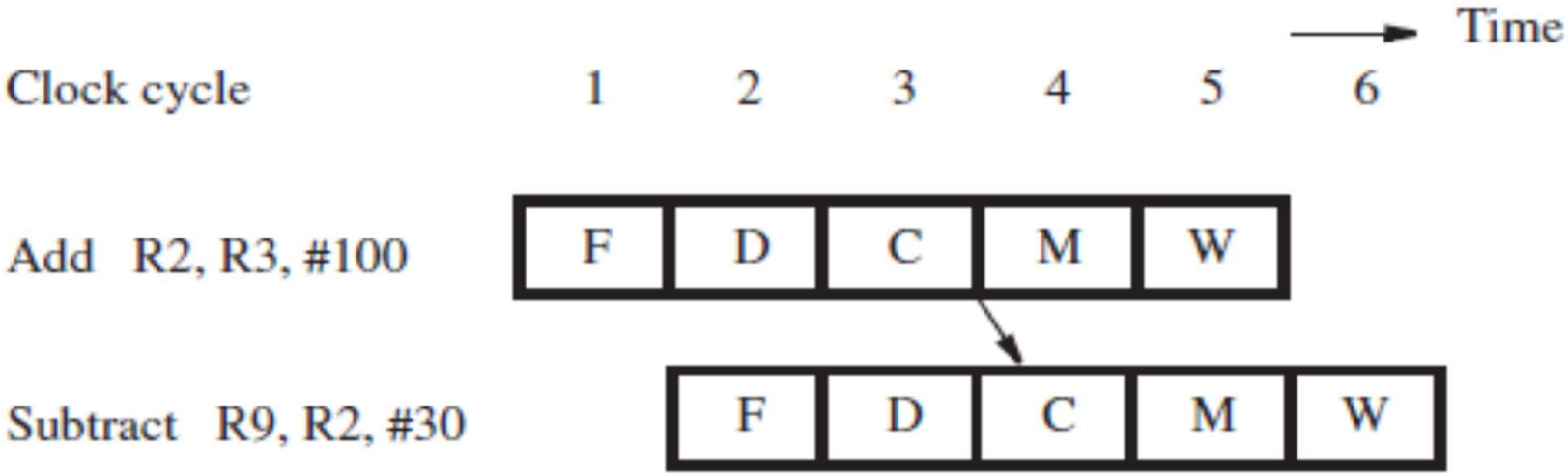


Figure 6.6 Avoiding a stall using Operand Forwarding

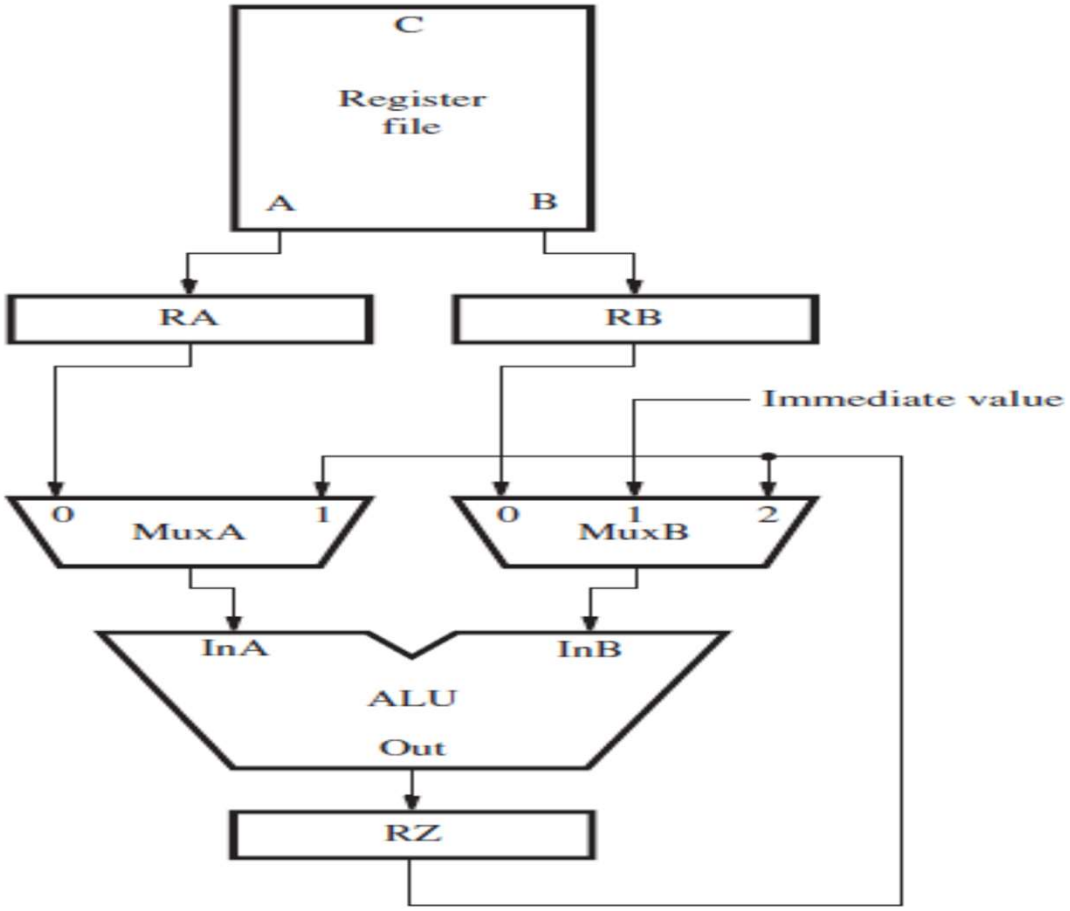


Figure 6.7 Modification of the datapath of Figure 6.2 to support data forwarding fromregister RZ to the ALU inputs.

Operand Forwarding

- Forwarding can also be extended to a result in register RY in Figure 6.2

This would handle a data dependency such as the one involving register R2 in the following sequence of instructions:

Add R2, R3, #100

Or R4, R5, R6

Subtract R9, R2, #30

Operand Forwarding

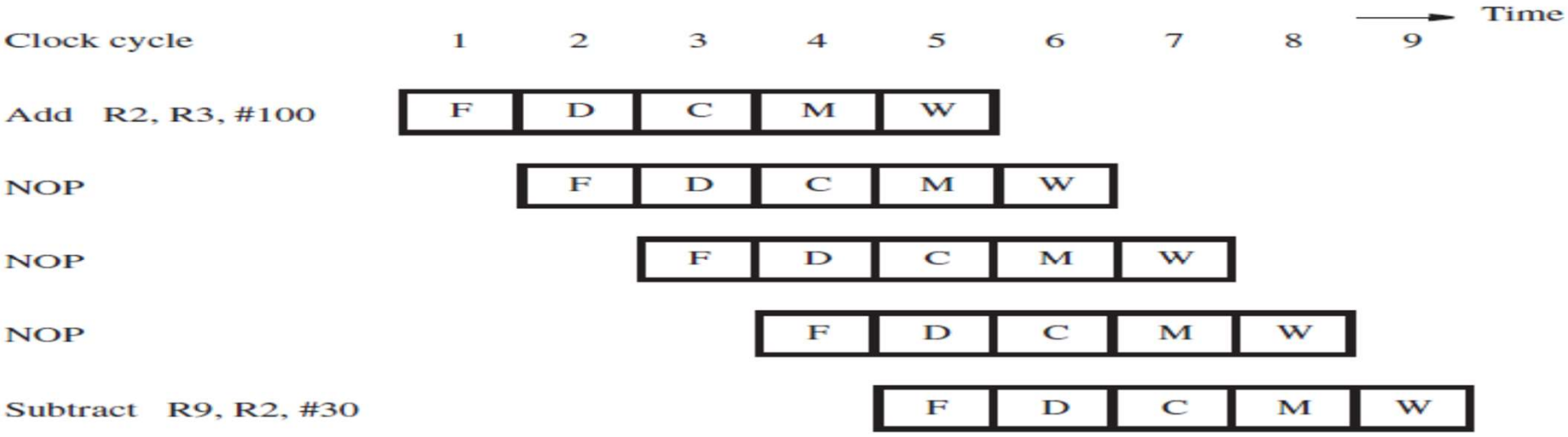
- When the Subtract instruction is in the Compute stage of the pipeline, the or instruction is in the Memory stage (where no operation is performed), and the Add instruction is in the Write stage
- The new value of register R2 generated by the Add instruction is now in register RY
- Forwarding this value from register RY to ALU input InA makes it possible to avoid stalling the pipeline
- MuxA requires another input for the value of RY. Similarly, MuxB is extended with another input.

Handling Data Dependencies in Software

- Data dependencies may be handled by the processor hardware, either by stalling the pipeline or by forwarding data
- An alternative approach is to leave the task of detecting data dependencies and dealing with them to the compiler
- When the 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

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

(a) Insertion of NOP instructions for a data dependency



(b) Pipelined execution of instructions

Figure 6.8 Using NOP instructions to handle a data dependency in software

Handling Data Dependencies in Software

- Requiring the compiler to identify dependencies and insert NOP instructions simplifies the hardware implementation of the pipeline
- However, the code size increases, and the execution time is not reduced as it would be with operand forwarding
- The compiler can attempt to *optimize* the code to improve performance and reduce the code size by reordering instructions to move useful instructions into the NOP slots
- In doing so, the compiler must consider data dependencies between instructions, which constrain the extent to which the NOP slots can be usefully filled

Memory Delays

- Delays arising from memory accesses are another cause of pipeline stalls
- For example, a 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*
- Figure 6.9 shows the effect of a delay in accessing data in the memory on pipelined execution
- A memory access may take ten or more cycles
- For simplicity, the figure shows only three cycles
- A cache miss causes all subsequent instructions to be delayed
- A similar delay can be caused by a cache miss when fetching an instruction

Memory Delays

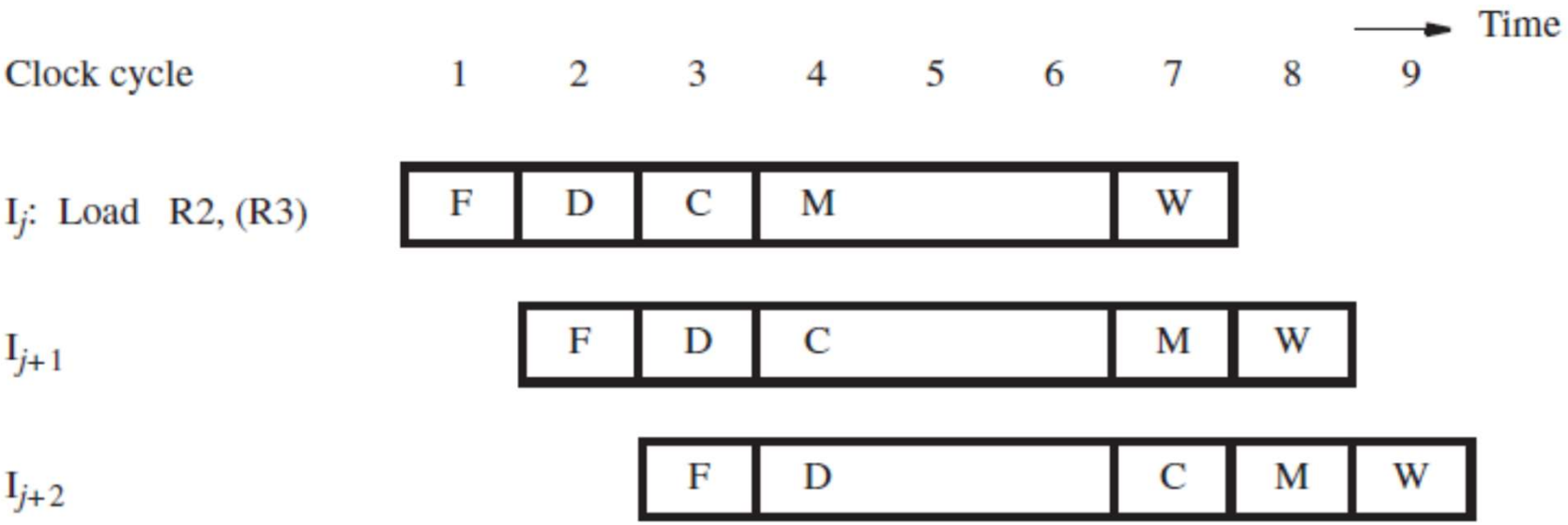


Figure 6.9 Stall caused by a memory access delay for a Load instruction

Memory Delays

- There is an additional type of memory-related stall that occurs when there is a data dependency involving a Load instruction
- Consider the instructions:

  Load R2, (R3)

  Subtract R9, R2, #30

Memory Delays

- 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 in the same manner as Figure 6.6, because the data read from memory (the cache, in this case) are not available until they are loaded into register RY at the beginning of cycle 5
- Therefore, the Subtract instruction must be stalled for one cycle, as shown in Figure 6.10, to delay the ALU operation
- The memory operand, which is now in register RY, can be forwarded to the ALU input in cycle 5.

Memory Delays

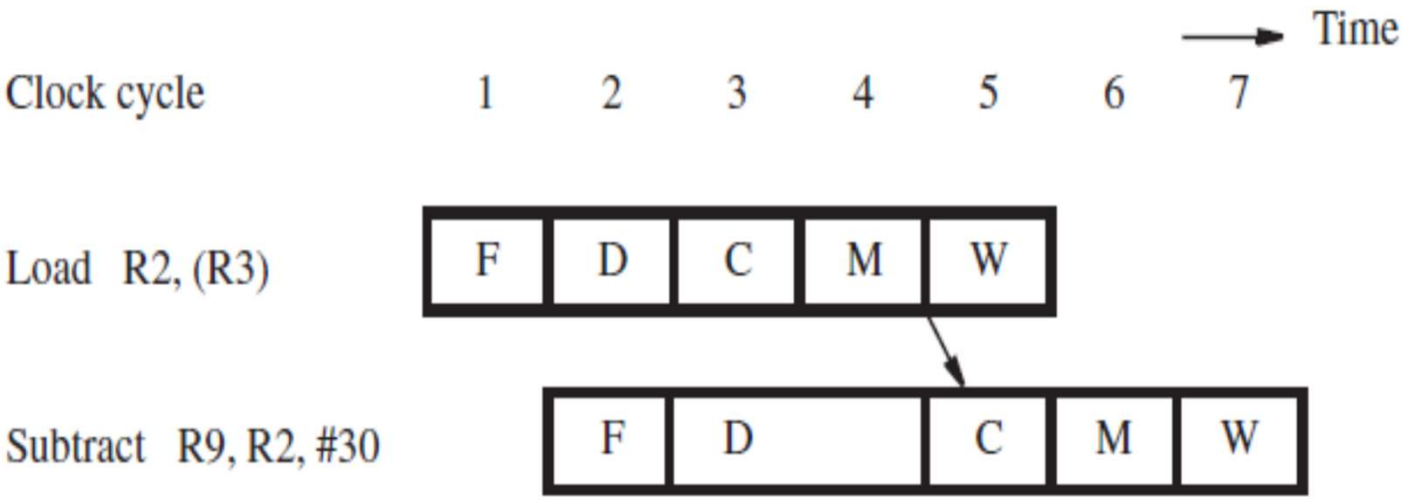


Figure 3.10

Parallel Processing

- Three additional techniques for improving performance, namely multithreading, vector processing, and multiprocessing
- These techniques are implemented in the multicore processor chips that are used in general-purpose computers
- They increase performance by improving the utilization of processing resources and by performing more operations in parallel

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 regarded by the OS as an entity 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

Hardware MultiThreading

- Each process has a corresponding thread, which is an independent path of execution within a program
- More precisely, 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.

Hardware MultiThreading

- In this section, we focus on multitasking where two or more programs run on the same processor and each program has a single thread
- Time Slicing is the technique where the 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
- 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

Hardware MultiThreading

- To deal with multiple threads efficiently, a 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*.

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

Hardware 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
- For example, 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.

Hardware MultiThreading

- An alternative to switching between threads on specific events is to 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.
- Thus, throughput may be increased by interleaving instructions from many threads, but it takes longer for a given thread to complete all of its instructions

Vector(SIMD) Processing

- Many computationally demanding applications involve programs that 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
- As a result, many instructions need to be executed to process all vector elements

Vector(SIMD) Processing

- A processor can be enhanced with multiple ALUs
- In such a processor, 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*.

Vector(SIMD) Processing

- 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
- If vector instructions are provided for different sizes of data elements using the same vector registers, *L* may vary.
  - For example, the Intel IA-32 architecture has 128-bit vector registers that are used by instructions for vector lengths ranging from $L = 2$ up to $L = 16$, corresponding to integer data elements with sizes ranging from 64 bits down to 8 bits.

Vector(SIMD) Processing

- Some typical examples of vector instructions are given in next slides to illustrate how vector registers are used
- We assume that the OP-code includes a suffix S which specifies the size of each data element
- This determines the number of elements, *L*, in a vector.
- For instructions that access the memory, the contents of a conventional register are used in the calculation of the effective address.

Vector(SIMD) Processing

- The vector instruction
  VectorAdd.S V*i*, V*j*, V*k*

  computes *L* sums using the elements in vector registers V*j* and V*k*, and places the resulting sums in vector register V*i*.
- Similar instructions are used to perform other arithmetic operations
- Special instructions are needed to transfer multiple data elements between a vector register and the memory
- The instruction
  VectorLoad.S V*i*, X(R*j*)

causes *L* consecutive elements beginning at memory location X + [R*j*] to be loaded into vector register V*i*.

Vector(SIMD) Processing

- Similarly, the instruction
  VectorStore.S V*i*, X(R*j*)
  causes the contents of vector register V*i* to be stored as *L* consecutive elements 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 and 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

- Vectorization of a loop is illustrated with the simple example of C-language code shown in Figure 3.11*a*
- 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 shown in Figure 3.11*b*
- The loop body has nine instructions, hence a total of 9*N* instructions are executed for *N* passes through the loop.

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

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

	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

	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

Figure 3.11 Example for Loop Vectorization

Vectorization

- To vectorize the loop, the compiler must recognize that the computation in each pass through the loop is independent of the other passes, and that the same operation can be performed simultaneously on multiple elements
- Assume for simplicity that the number of passes, *N*, is evenly divisible by the vector length, *L*
- 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
- Consequently, 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 4*L*, and the count in register R5 is decremented by *L*
- The vectorized loop is shown in Figure 3.11c.

Vectorization

- We assume that the assembler calculates the value of 4*L* for the expression given as the immediate operand in the Add instructions
- There are still nine instructions in the loop body, but because the number of passes is now *N/L*, the total number of instructions that- need to be executed is only 9*N/L*.

Vectorization

- Vectorizable loops exist in programs for applications such as computer graphics and digital signal processing
- Such loops have many independent computations that can be performed simultaneously on several data elements
- If a large portion of the execution time for an application is spent executing loops of this type, then vectorization of these loops can significantly reduce the total execution time
- The extent of the performance improvement is limited by the vector length *L*, which determines the number of ALUs that can operate in parallel

Graphics Processing Units

- The increasing demands of processing for computer graphics has led to the development of specialized chips called graphics processing units (GPUs)
- The primary purpose of GPUs is to accelerate the large number of floating-point calculations needed in high-resolution three-dimensional graphics, such as in video games
- Since the operations involved in these calculations are often independent, a large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel

Graphics Processing Units

- A GPU chip and a dedicated memory for it are included on a video card. Such a card is plugged into an expansion slot of a host computer using an interconnection standard such as the PCIe standard discussed
- 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

Graphics Processing Units

- 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
- 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
- An example is the *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

Graphics Processing Units

- 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

- A multiprocessor system consists of a number of processors capable of simultaneously executing independent tasks
- The granularity of these tasks can vary considerably
- A task may encompass a few instructions for one pass through a loop, or thousands of instructions executed in a subroutine
- In a shared-memory multiprocessor, 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

Shared Memory MultiProcessors

- Implementing a large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously
- Solution: Distributing the memory 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.

Uniform Memory Access(UMA) Multiprocessor

- An interconnection network enables any processor to access any module that is a part of the shared memory
- When memory modules are kept physically separate from the processors, all requests to access memory must pass through the network, which introduces latency
- Figure 3.12 shows such an arrangement
- 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
- Although the latency is uniform, it may be large for a network that connects many processors and memory modules

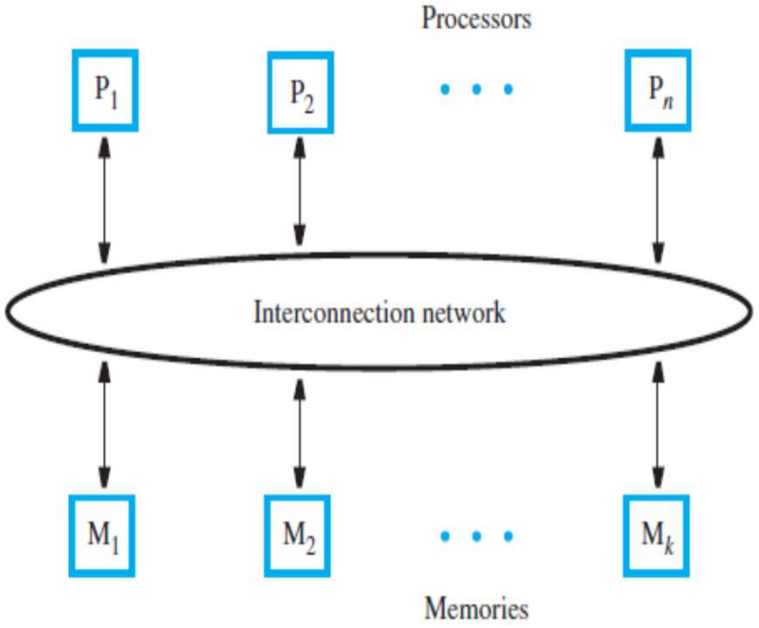


Figure 3.12 UMA Multiprocessor

Non-Uniform Memory Access(NUMA) Multiprocessor

- For better performance, 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
- The nodes are then connected to the network, as shown in Figure 3.13
- The network latency is avoided when a processor makes a request to access its local memory
- However, a request to access a remote memory module must pass through the network
- 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

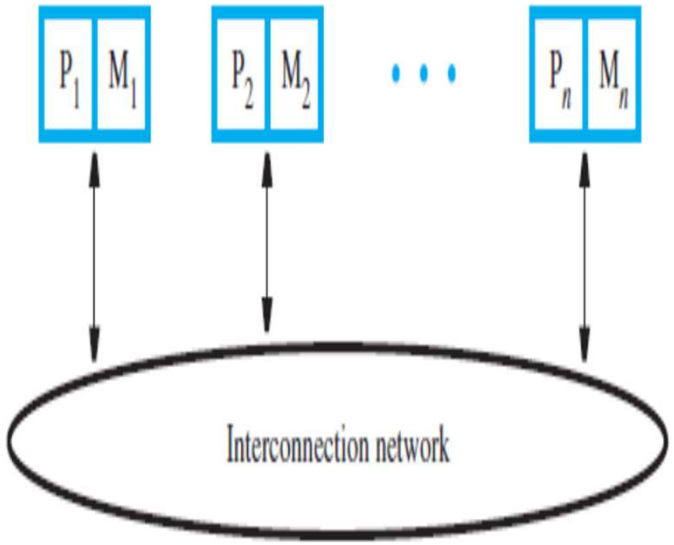


Figure 3.13 NUMA Multiprocessor

Interconnection Network

- The interconnection network must allow information transfer between any pair of nodes in the system
- The network 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

Interconnection Network

- Information transfer through the network usually takes place in the form of *packets* of fixed length and specified format
- For example, a read request is likely to be a single packet sent from a processor to a memory module
- The packet contains the node identifiers for the source and destination, the address of the location to be read, and a command field that indicates what type of read operation is required
- A write request that writes one word in a memory module is also likely to be a single packet that includes the data to be written
- On the other hand, a read response may involve an entire cache block requiring several packets for the data transfer.

Interconnection Network

- Ideally, a complete packet would be handled in parallel in one clock cycle at any node or switch in the network
- This implies having wide links, comprising many wires
- However, to reduce cost and complexity, the links are often considerably narrower
- In such cases, a packet must be divided into smaller pieces, each of which can be transmitted in one clock cycle

Bus

- A *bus* is a 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 because of the contention for access to the bus and the increased propagation delays caused by electrical loading when many processors are connected
- A simple bus does not allow a new request to appear on the bus until the response for the current request has been provided
- However, if the response latency is high, there may be considerable idle time on the bus

Split-Transaction Bus

- Higher performance can be achieved by using a *split-transaction* bus, in which a request and its corresponding response are treated as separate events
- Other transfers may take place between them
- Consider a situation where multiple processors need to make read requests to the memory
- Arbitration is used to select the first processor to be granted use of the bus for its request
- After the request is made, a second processor is selected to make its request, instead of leaving the bus idle
- Assuming that this request is to a different memory module, the two read accesses proceed in parallel
- If neither module has finished with its access, a third processor is selected to make its request, and so on

Split-Transaction Bus

- Eventually, one memory module completes its read access
- It is granted the use of the bus to transfer the data to the requesting processor
- As other modules complete their accesses, the bus is used to transfer their responses
- The actual length of time between each request and its corresponding response may vary as requests and responses for different transactions with the memory are interleaved on the bus to make efficient use of the available bandwidth
- The split-transaction bus requires a more complex bus protocol
- The main source of complexity is the need to match each response with its corresponding request

Split-Transaction Bus

- This is usually handled by associating a unique tag with each request that appears on the bus
- Each response then appears with the appropriate tag so that the source can match it to its original request

RING

- A ring network is formed with point-to-point connections between nodes
- A single ring is shown in the Figure below
- 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.

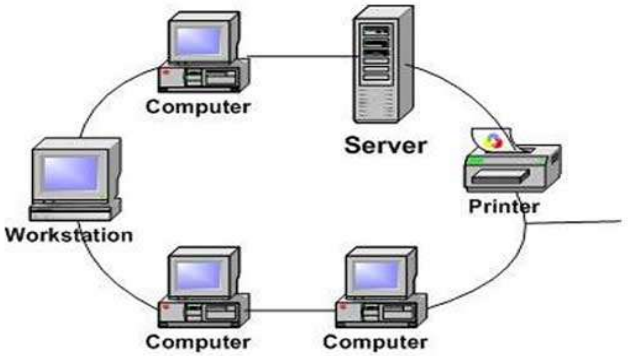


Fig 3.14 Ring Network

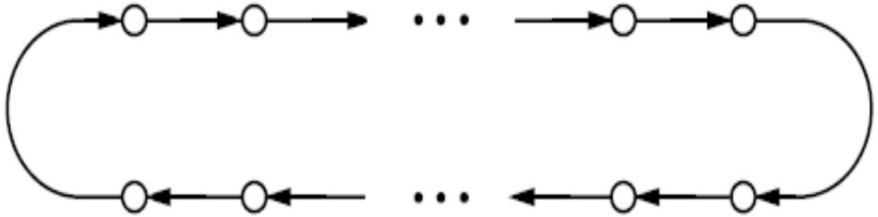


Fig 3.15 Single Ring

RING

- 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

RING

- Another approach is to use a *hierarchy* of rings. A two-level hierarchy
- 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 ringscommunicate with each other frequently.

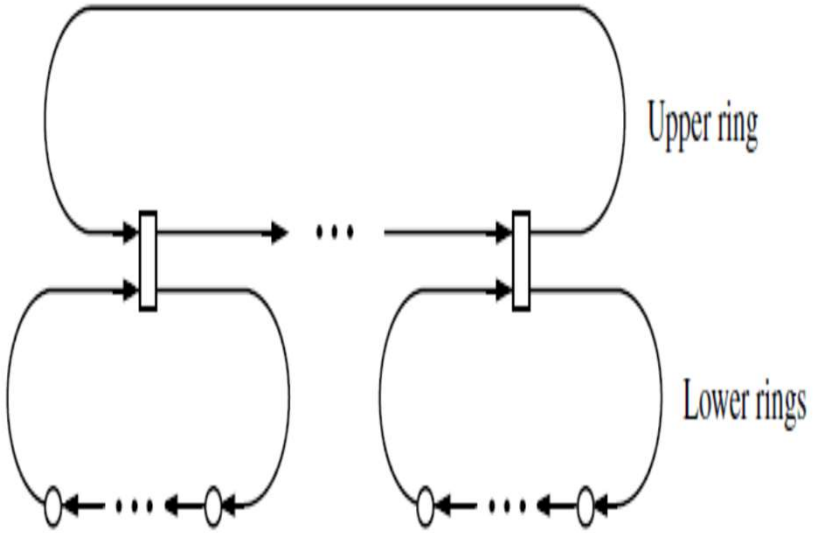


Fig 3.15 Hierarchy of Ring

CrossBar

- 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.
- For n processors and k memories, n × k switches are needed

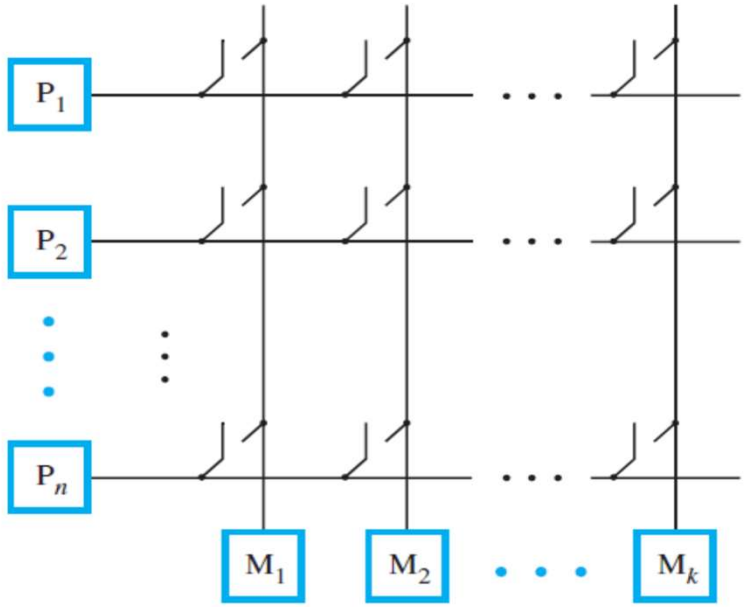


Fig 3.16 Crossbar Interconnection Network

Mesh

- A natural way of connecting a large number of nodes is with a two-dimensional *mesh*, as shown in Figure 3.17
- 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, wrap around 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.

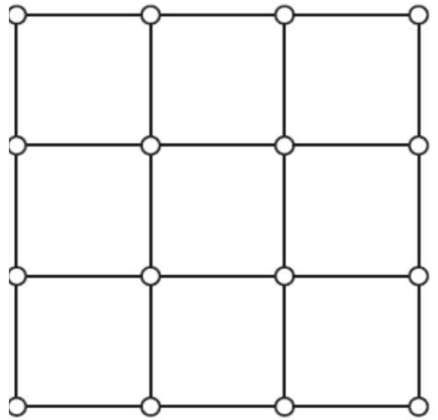


Fig 3.17 A two dimensional Mesh Network

Cache Coherence

- A shared-memory multiprocessor is easy to program
- Each variable in a program has a unique address location in the memory, which can be accessed by any processor
- However, each processor has its own cache
- Therefore, it is necessary to deal with the 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

Cache Coherence

- This is the issue of maintaining *cache coherence*, which requires having a consistent view of shared data in multiple caches.
- Two basic approaches for performing write operations on data in a cache is already discussed
- The write-through approach changes the data in both the cache and the main memory
- The write-back approach changes the data only in the cache; the main memory copy is updated when a modified data block in the cache has to be replaced
- Similar approaches can be used to address cache coherence in a multiprocessor system

Write Through Protocol

- A write-through protocol can be implemented in one of two ways
- One version is based on updating the values in other caches
- Second relies on invalidating the copies in other caches

Update Protocol

- When a processor writes a new value to a 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 of doing this is to 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

Invalidation

- 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
- Again, broadcasting can be used to send the invalidation requests throughout the system.

Write Back Protocol

- Maintaining coherence with the write-back protocol is 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.

Write Back Protocol - Read

- 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.

Write Back Protocol - Write

- When another processor wishes to write to a block that has been modified, the current owner sends the data to the requesting processor
- It also transfers ownership of the block to the requesting processor and invalidates its cached copy
- Since the block is being modified by the new owner, the contents of the block in the memory are not updated
- The next request for the same block is serviced by the new owner

Write Back Protocol

- The write-back protocol has the advantage of creating less traffic than the write-through protocol
- This is because a processor is likely to perform several writes to a cache block before this block is needed by another processor
- With the write-back protocol, these writes are performed only in the cache, once ownership is acquired with an invalidation request
- With the write-through protocol, each write must also be performed in the appropriate memory module and broadcast to other caches

Directory Based Cache Coherence

- Large shared memory multiprocessors use interconnection networks such as rings and meshes
- In such systems, 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

Directory Based Cache Coherence

- 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

END of UNIT 6