ARITHMETIC PIPELINE

ARITHMETIC PIPELINE

- Pipeline arithmetic units are usually found in very high speed computers.
- They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations
- A pipeline multiplier is essentially an array multiplier with special adders designed to minimize the carry propagation time through the partial products.
- Floating-point operations are easily decomposed into sub-operations

Steps:

- $X = A \times 2^a$
- $Y = B \times 2^b$

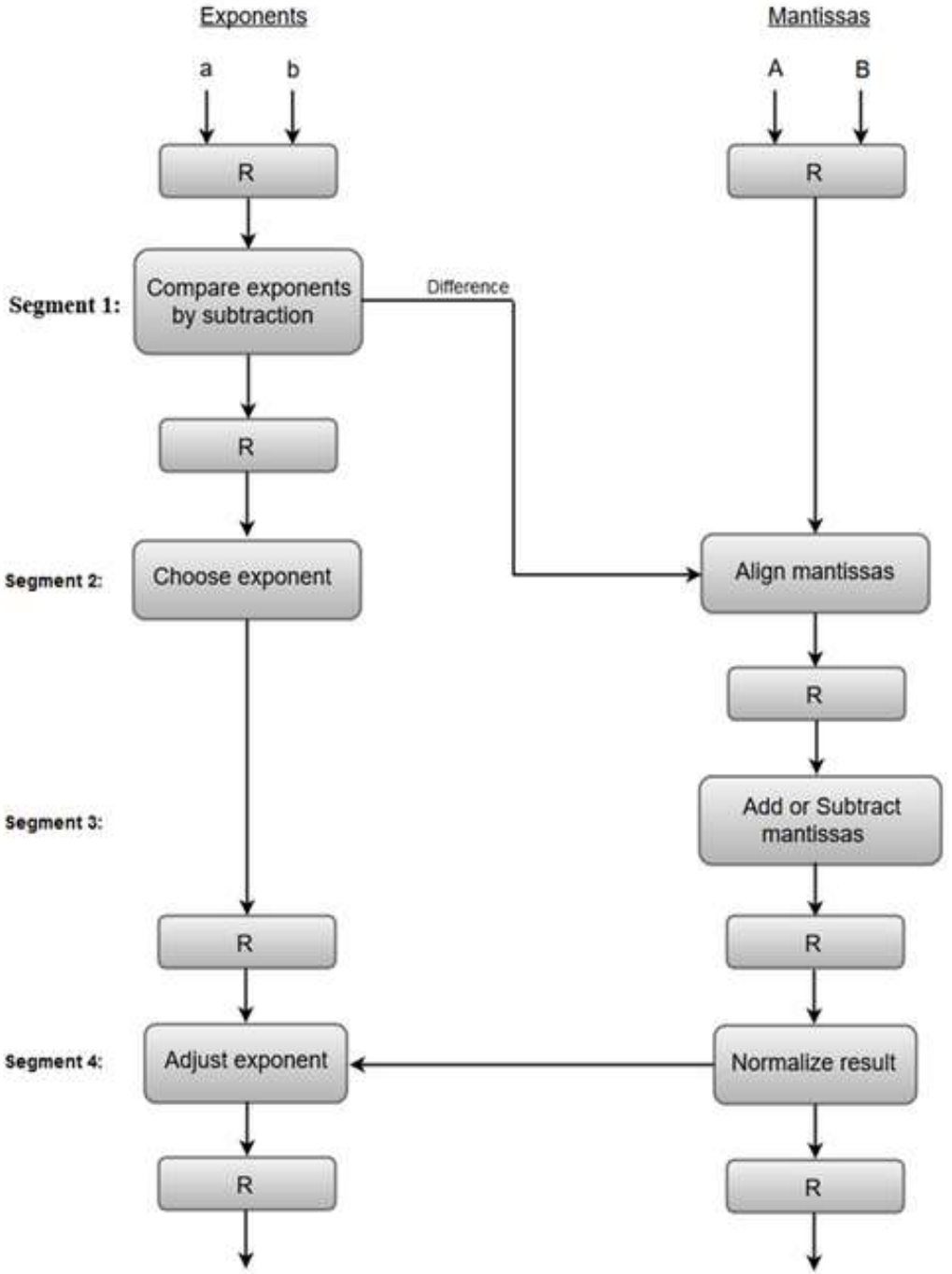
- A and B are two fractions that represent the mantissas and a and b are the exponents.

- The floating-point addition and subtraction can be performed in four segments

- The registers labeled R are placed between the segments to store intermediate results.

- The sub-operations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result.



- The exponents are compared by subtracting them to determine their difference
- The larger exponent is chosen as the exponent of the result.
- The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right.
- This produces an alignment of the two mantissas.
- The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4.
- When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one.
- If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

Example:

- $X = 0.9504 \times 10^3$
- $Y = 0.8200 \times 10^2$
- The two exponents are subtracted in the first segment to obtain 3 - 2 = 1.

- The larger exponent 3 is chosen as the exponent of the result.

- The next segment shifts the mantissa of Y to the right to obtain

  $X = 0.9504 \times 10^3$

  $Y = 0.0820 \times 10^3$

- This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

  $Z = 1.0324 \times 10^3$

- Normalize the result so that it has a fraction with a non-zero first digit.

- This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \text{ X } 10^4$$

- The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits.

- Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns.

- The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns.

- An equivalent non-pipeline floating point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns.

- In this case the pipelined adder has a speedup of 320/110 = 2. 9 over the non-pipelined adder.

ARRAY PROCESSORS

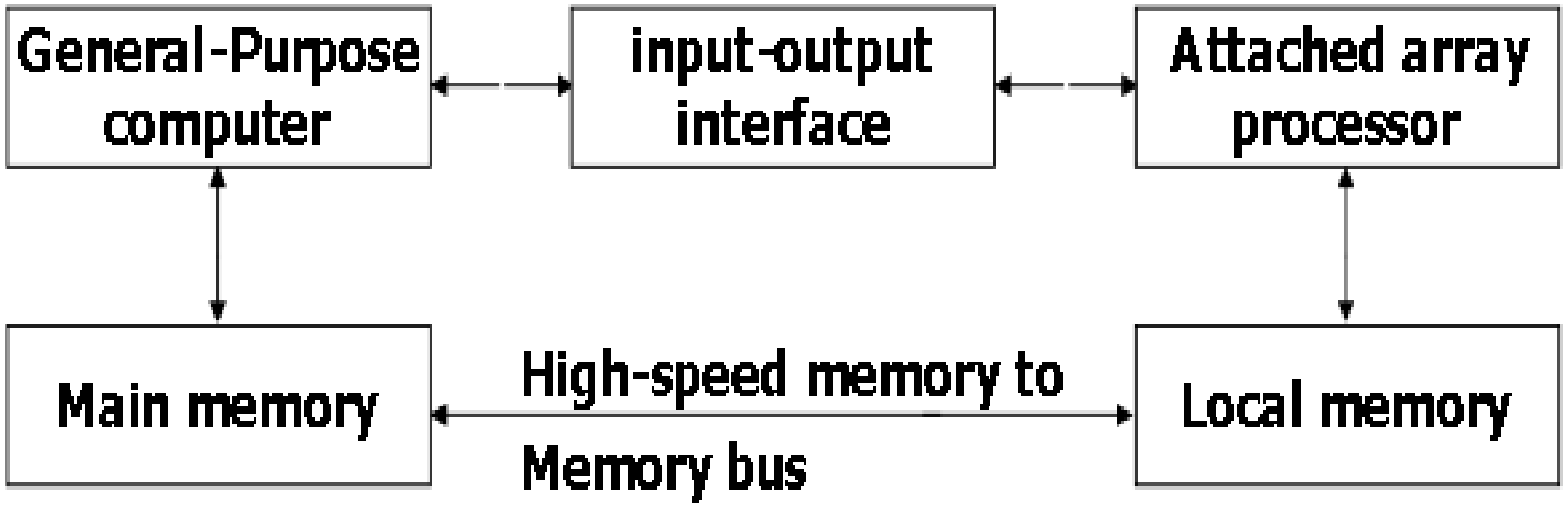
ARRAY PROCESSORS

- It is a processor that performs computations on large arrays of data.
- Two different types of processors: An attached array processor is an auxiliary processor attached to a general-purpose computer.
- It is intended to improve the performance of the host computer in specific numerical computation tasks.
- An SIMD array processor is a processor that has a single-instruction multiple-data organization.
- It manipulates vector instructions by means of multiple functional units responding to a common instruction.
- Although both types of array processors manipulate vectors, their internal organization is different.

ATTACHED ARRAY PROCESSOR

- It is designed as a peripheral for a conventional host computer, and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
- It achieves high performance by means of parallel processing with multiple functional units.
- It includes an arithmetic unit containing one or more pipelined floating point adders and multipliers.
- The array processor can be programmed by the user to accommodate a variety of complex arithmetic problems.

Interconnection of an attached array processor to a host computer



- The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer.
- The array processor is connected through an input-output controller to the computer and the computer treats it like an external interface.
- The data for the attached processor are transferred from main memory to a local memory through a high-speed bus.
- The system with the attached processor satisfies the needs for complex arithmetic applications.
- Some models that can be connected to a variety of different host computers.
- For example, when attached to a VAX 11 computer, the FSP-164/MAX from Floating-Point Systems increases the computing power of the VAX to 100 megaflops.
- The objective of the attached array processor is to provide vector manipulation capabilities to a conventional computer at a fraction of the cost of supercomputers

SIMD ARRAY PROCESSOR

- It is a computer with multiple processing units operating in parallel.
- The processing units are synchronized to perform the same operation under the control of a common control unit, thus providing a single instruction stream, multiple data stream (SIMD) organization.
- It contains a set of identical processing elements (PEs), each having a local memory M .
- Each processor element includes an ALU, a floating-point arithmetic unit, and working registers.
- The master control unit controls the operations in the processor elements.
- The main memory is used for storage of the program.

- The function of the master control unit is to decode the instructions and determine how the instruction is to be executed.
- Scalar and program control instructions are directly executed within the master control unit.
- Vector instructions are broadcast to all PEs simultaneously.
- Each PE uses operands stored in its local memory.
- Vector operands are distributed to the local memories prior to the parallel execution of the instruction.

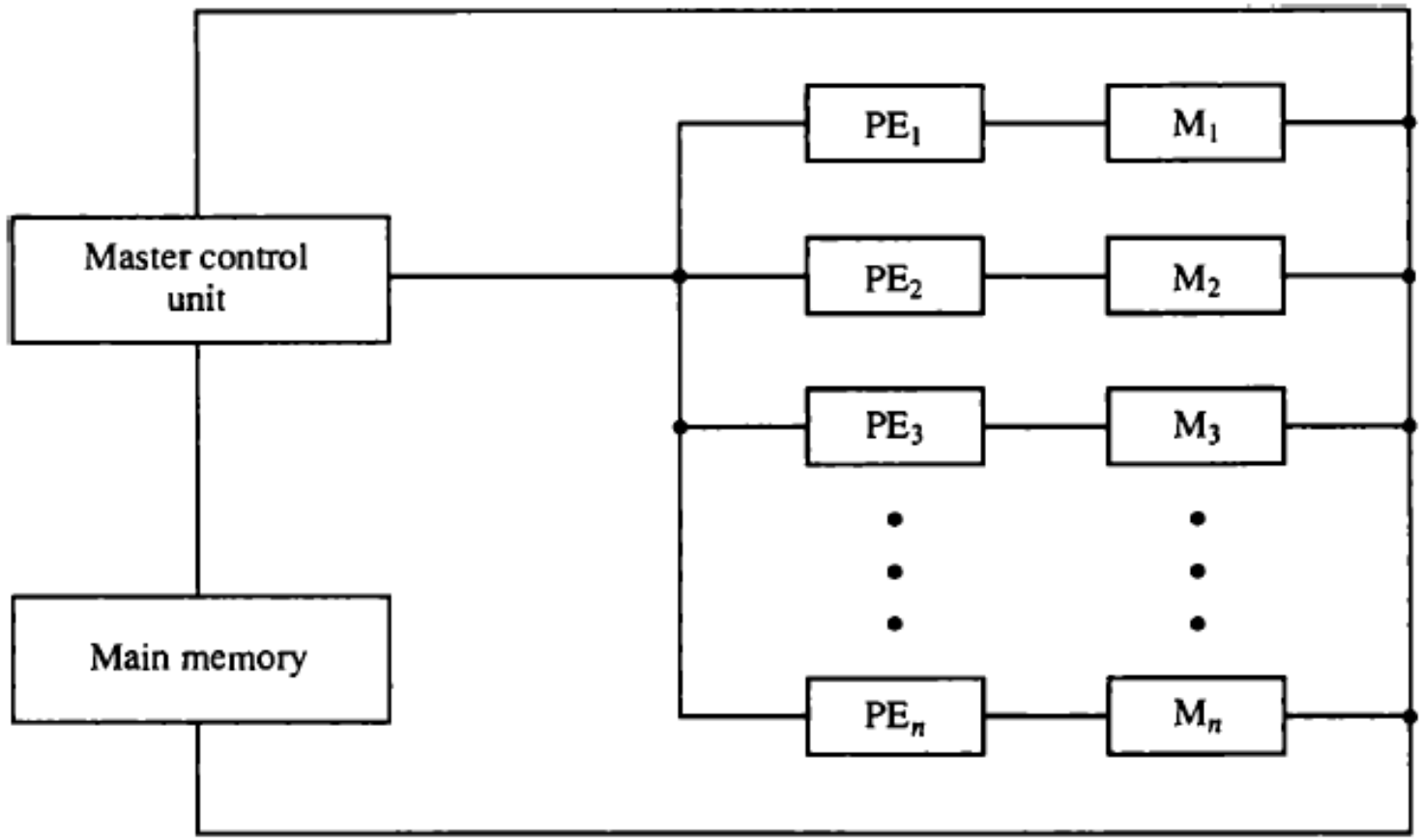


Figure 9-15 SIMD array processor organization.

Example:

- The vector addition C = A + B.
- The master control unit first stores the i th components a, and b, of A and B in local memory M, for i = 1, 2, 3, . . . , n.
- It then broadcasts the floating-point add instruction c, = a, + b, to all PEs, causing the addition to take place simultaneously.
- The components of c, are stored in fixed locations in each local memory. This produces the desired vector sum in one add cycle
- Each PE has a flag that is set when the PE is active and reset when the PE is inactive.
- This ensures that only those PEs that need to participate are active during the execution of the instruction.

- For example, suppose that the array processor contains a set of 64 PEs.
- 1f a vector length of less than 64 data items is to be processed, the control unit selects the proper number of PEs to be active.
- Vectors of greater length than 64 must be divided into 64-word portions by the control unit
- ILLIAC IV - no longer in operation.
- SIMD processors are highly specialized computers.
- They are suited primarily for numerical problems that can be expressed in vector or matrix form.
- However, they are not very efficient in other types of computations or in dealing with conventional data-processing programs.

INSTRUCTION PIPELINE

INSTRUCTION PIPELINE

- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.
- This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.
- One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence.
- In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

EG:

- Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline.
- The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer.
- Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory.
- The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis.
- Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment.
- The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions.

- Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase.

- The buffer acts as a queue from which control then extracts the instructions for the execution unit.

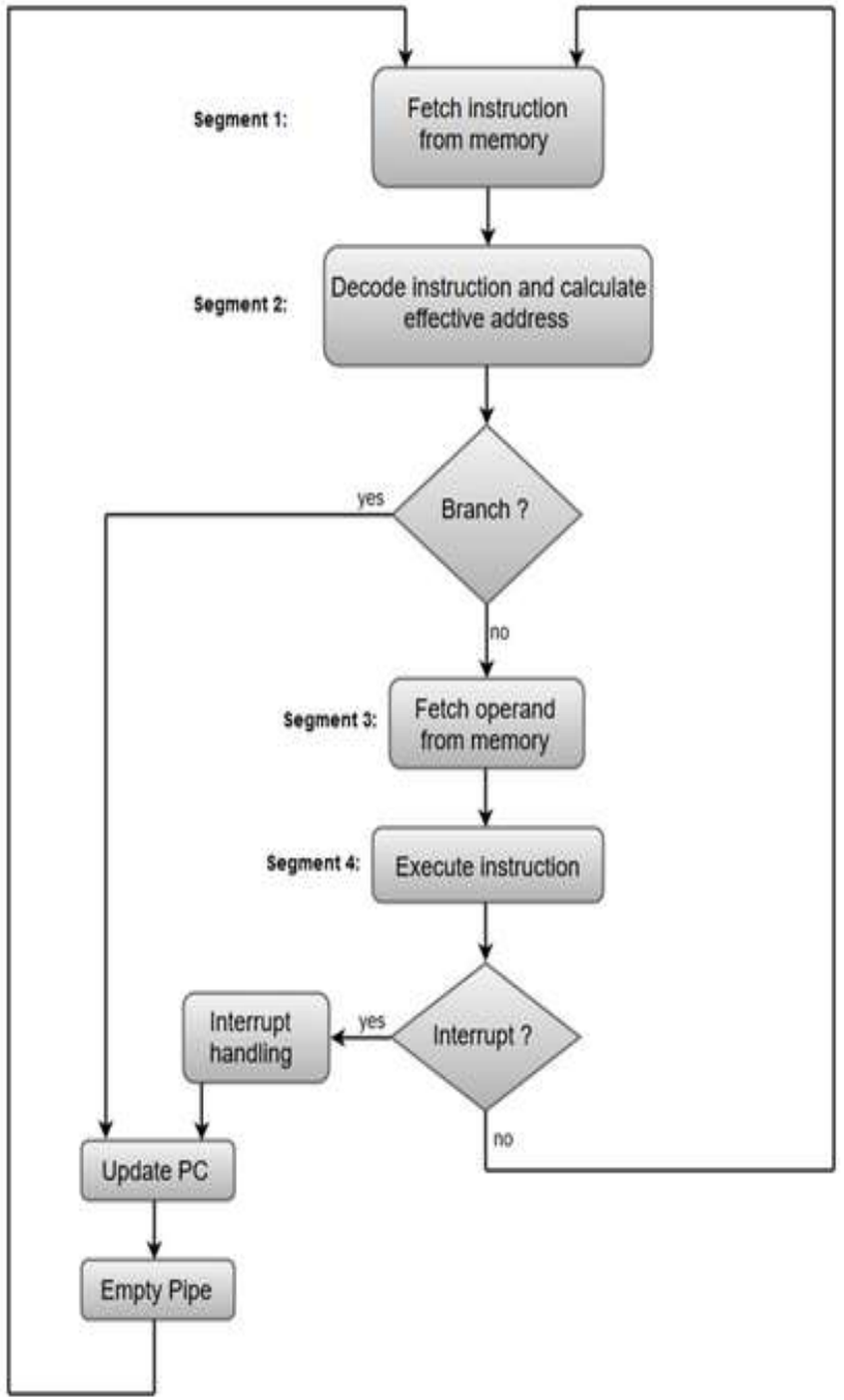
- Complex instructions require other phases

    1. Fetch the instruction from memory.
    2. Decode the instruction.
    3. Calculate the effective address.
    4. Fetch the operands from memory.
    5 . Execute the instruction.
    6. Store the result in the proper place.

Four-Segment Instruction Pipeline

- decoding of the instruction can be combined with the calculation of the effective address into one segment.

- the instruction execution and storing of the result can be combined into one segment



- While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

- The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.

- Thus up to four sub-operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.
- an instruction in the sequence may be a program control type that causes a branch out of normal sequence.

- In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted.

- The pipeline then restarts from the new address stored in the program counter.

- Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

Operation of instruction pipeline

1. Fl is the segment that fetches an instruction.

2. DA is the segment that decodes the instruction and calculates the effective address.

3. FO is the segment that fetches the operand.

4. EX is the segment that executes the instruction.

- It is assumed that the processor has separate instruction and data memories
- so that the operation in Fl and FO can proceed at the same time.

Timing of instruction pipeline

	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	--	--	FI	DA	FO	EX			
	5					--	--	--	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

- In step 4, instruction 1 is being executed in segment EX;
- the operand for instruction 2 is being fetched in segment FO;
- instruction 3 is being decoded in segment DA;
- instruction 4 is being fetched from memory in segment FL

- Assume now that instruction 3 is a branch instruction.

- As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6.

- If the branch is taken, a new instruction is fetched in step 7.

- If the branch is not taken, the instruction fetched previously in step 4 can be used.

- The pipeline then continues until a new branch instruction is encountered

- Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand.

- In that case, segment FO must wait until segment EX has finished its operation.

- There are three major difficulties that cause the instruction pipeline to deviate from its normal operation- Pipeline conflicts or pipeline hazards

1. Resource conflicts caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

3. Branch difficulties arise from branch and other instructions that change the value of PC .

Data Dependency

- Due to possible collision of data or address.
- A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations.
- A data dependency occurs when an instruction needs data that are not yet available.
- For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX.
- Therefore, the second instruction must wait for data to become available by the first instruction.
- Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
- For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register.
- Therefore, the operand access to memory must be delayed until the required address is available.

Solution 1: HARDWARE INTERLOCKS

- The most straightforward method is to insert hardware interlocks.
- An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
- Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.
- This approach maintains the program sequence by using hardware to insert the required delays.

Solution 2: OPERAND FORWARDING

- Uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.

- For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register file.

- This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

Solution 3: DELAYED LOAD

- A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program.

- The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting delayed load no-operation instructions.

HANDLING OF BRANCH INSTRUCTIONS

- Major issue in operating an instruction pipeline is the occurrence of branch instructions.
- A branch instruction can be conditional or unconditional.
- An unconditional branch always alters the sequential program flow by loading the program counter with the target address.
- In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied.
- The branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline
- Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching

Solution 1: PREFETCH TARGET INSTRUCTION

- One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch.
- Both are saved until the branch is executed.
- If the branch condition is successful, the pipeline continues from the branch target instruction.
- An extension of this procedure is to continue fetching instructions from both places until the branch decision is made.
- At that time control chooses the instruction stream of the correct program flow.

Solution 2: BRANCH TARGET BUFFER

- The use of a branch target buffer or BTB.
- The BTB is an associative memory included in the fetch segment of the pipeline.
- Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch.
- It also stores the next few instructions after the branch target instruction.
- When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction.
- If it is in the BTB, the instruction is available directly and prefetch continues from the new path.
- If the instruction is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB.
- The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

Solution 3: BRANCH PREDICTION

- A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- The pipeline then begins prefetching the instruction stream from the predicted path.
- A correct prediction eliminates the wasted time caused by branch penalties

Solution 4: DELAYED BRANCH

- Employed in most RISC processors is the delayed branch.
- In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.
- An example of delayed branch is the insertion of a no-operation instruction after a branch instruction.
- This causes the computer to fetch the target instruction during the execution of the no-operation instruction, allowing a continuous flow of the pipeline

Example: Three-Segment Instruction Pipeline (Refer text page 326)

- I: Instruction fetch
- A: ALU operation
- E: Execute instruction
- Delayed Load

1. LOAD: Rl ←M[address 1]
2. LOAD: R2←M[address 2]
3. ADD: R3←Rl + R2
4. STORE: M[address 3] ←R3

There will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment.

Clock cycles:	1	2	3	4	5	6
1. Load $R1$	I	A	E			
2. Load $R2$		I	A	E		
3. Add $R1 + R2$			I	A	E	
4. Store $R3$				I	A	E

(a) Pipeline timing with data conflict

- Compiler inserts a no-op (no-operation) instruction.
- This is a type of instruction that is fetched from memory but has no operation, thus wasting a clock cycle.
- This concept of delaying the use of the data loaded from memory is referred to as delayed load

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

Delayed Branch

- Analyzes the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps
- For example, the compiler can determine that the program dependencies allow one or more instructions preceding the branch to be moved into the delay steps after the branch.
- These instructions are then fetched from memory and executed through the pipeline while the branch instruction is being executed in other segments.
- The effect is the same as if the instructions were executed in their original order, except that the branch delay is removed.
- It is up to the compiler to find useful instructions to put after the branch instruction.

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to *X*					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in *X*								I	A	E

(a) Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to *X*			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in *X*						I	A	E

(b) Rearranging the instructions

Figure 9-10 Example of delayed branch.

MULTIPROCESSORS

CHARACTERISTICS OF MULTIPROCESSORS

- It is an interconnection of two or more CPUs with memory and input-output equipment.
- The term "processor“ in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (lOP).
- Multiprocessors are classified as multiple instruction stream, multiple data stream (MIMO) systems.
- Multicomputer systems-Computers are interconnected with each other by means of communication lines to form a computer network.
- The network consists of several autonomous computers that may or may not communicate with each other.
- A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem

- Microprocessor-take very little physical space and are very inexpensive brings about the feasibility of interconnecting a large number of microprocessors into one composite system.

- Very-large-scale integrated circuit technology(VLSI)

- Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system.

- If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor.

- The system as a whole can continue to function correctly

- Improved system performance.

- The system derives its high performance from the fact that computations can proceed in parallel in one of two ways.

    1. Multiple independent jobs can be made to operate in parallel.
    2. A single job can be partitioned into multiple parallel tasks.

- An overall function can be partitioned into a number of tasks that each processor can handle individually

- This can be achieved in one of two ways.

- 1. The user can explicitly declare that certain tasks of the program be executed in parallel. This must be done prior to loading the program by specifying the parallel executable segments.

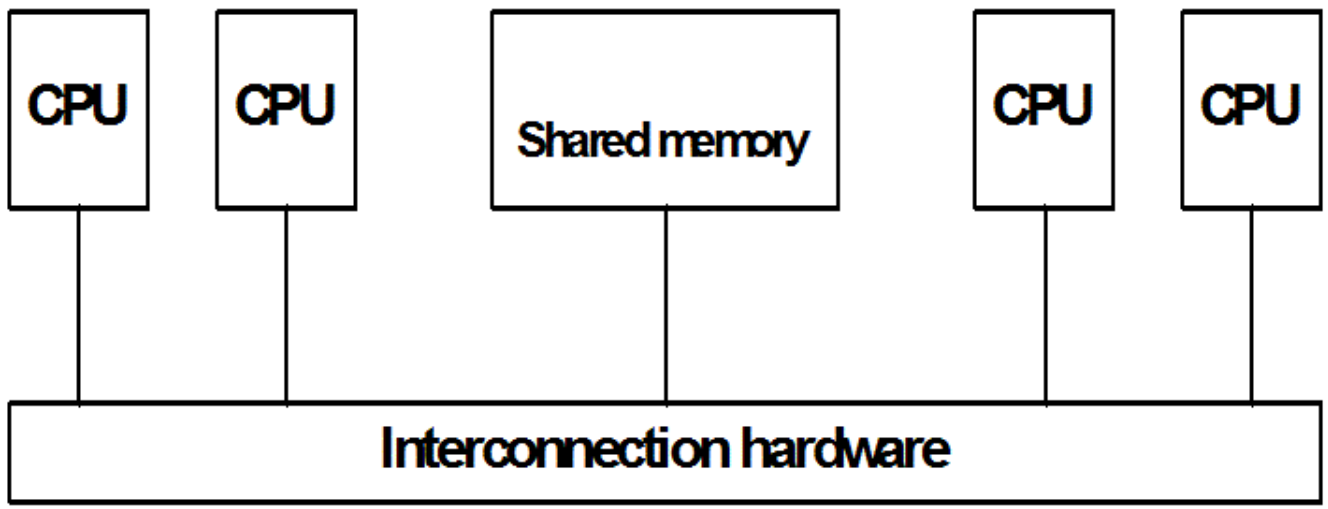
- 2. More efficient way is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program. The compiler checks for data dependency in the program.

CLASSIFICATION OF MULTIPROCESSORS

- Classified by the way their memory is organized.
- A multiprocessor system with common shared memory is classified as a shared-memory or tightly coupled multiprocessor.
- This does not preclude each processor from having its own local memory
- In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory
- tightly coupled systems can tolerate a higher degree of interaction between tasks.

Tightly Coupled Systems

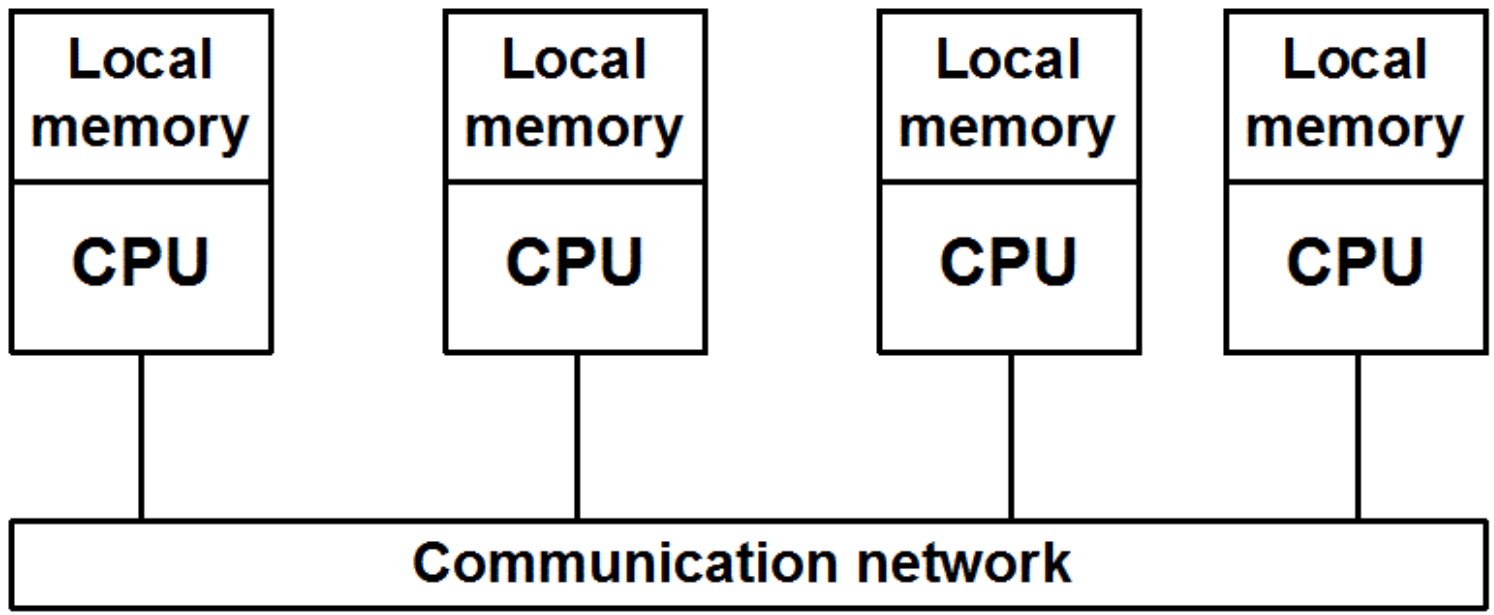
- Systems with a single system wide memory
- Parallel Processing System , SMMP (shared memory multiprocessor systems)



- Distributed-memory or loosely coupled system.
- Each processor element in a loosely coupled system has its own private local memory.
- The processors are tied together by a switching scheme designed to route information from one processor to another through a message- passing scheme.
- The processors relay program and data to other processors in packets.
- A packet consists of an address, the data content, and some error detection code.
- The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used.
- Loosely coupled systems are most efficient when the interaction between tasks is minimal

Loosely Coupled System

- Distributed Memory Systems (DMS)
- Communication via Message Passing



INTERCONNECTION STRUCTURES (refer text page 493)

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

Time-Shared Common Bus

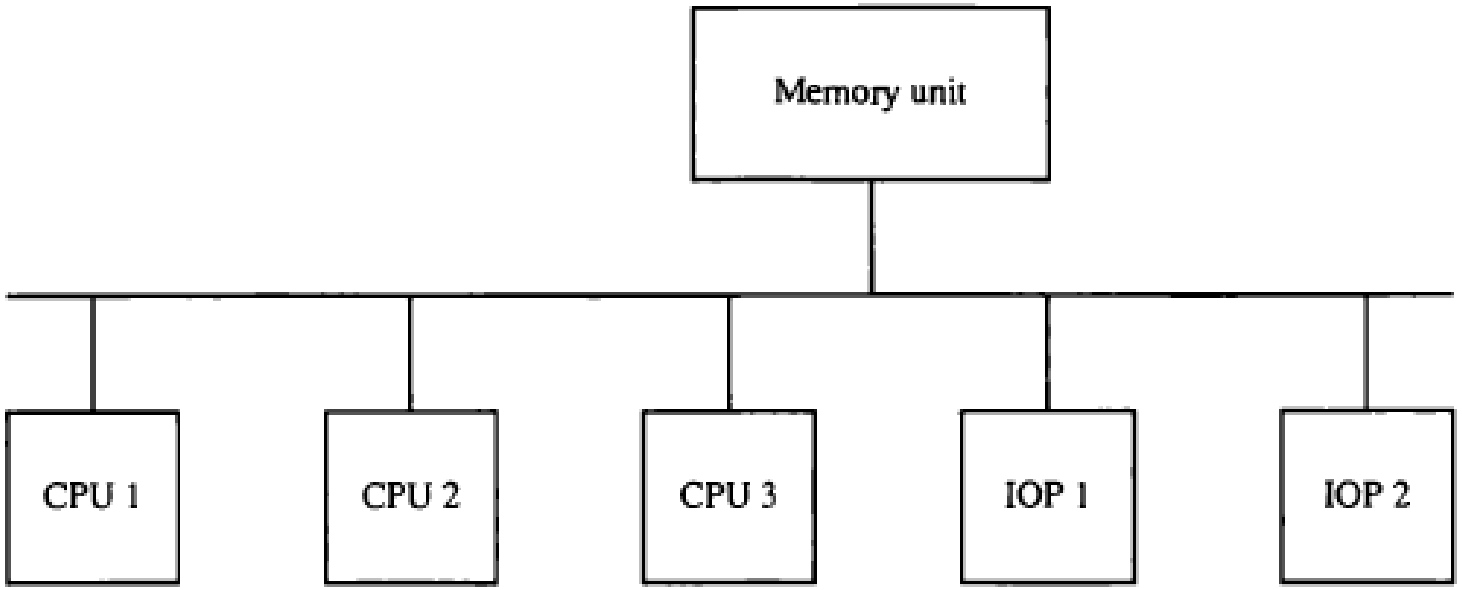


Figure 13-1 Time-shared common bus organization.

- Only one processor can communicate with the memory or another processor at any given time.
- Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer.
- A command is issued to inform the destination unit what operation is to be performed .
- The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated.
- The system may exhibit transfer conflicts since one common bus is shared by all processors.
- These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

- Dual bus structure

- Here we have a number of local buses each connected to its own local memory and to one or more processors.

- Each local bus may be connected to a CPU, an lOP, or any combination of processors.

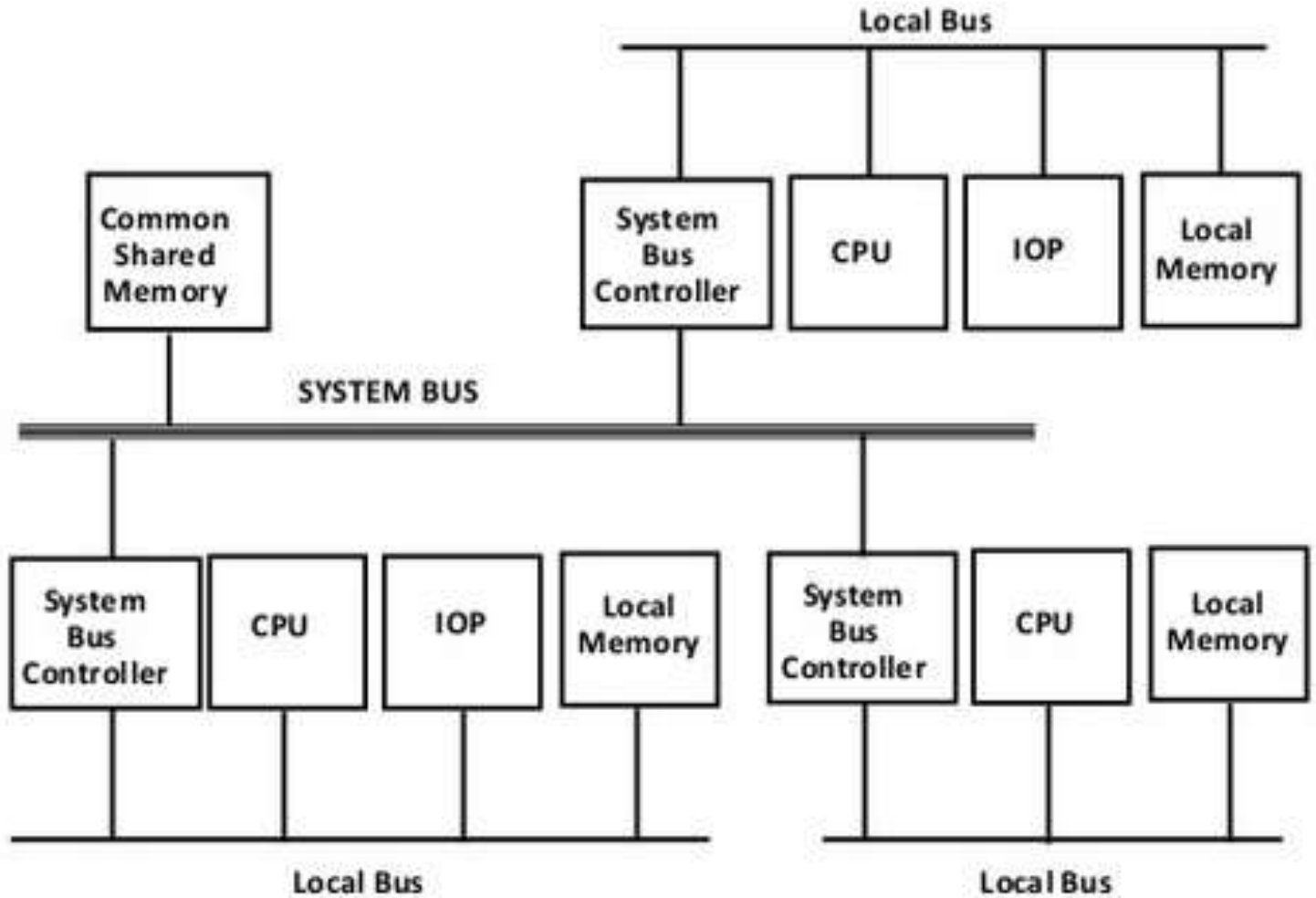
- A system bus controller links each local bus to a common system bus .

- The VO devices connected to the local lOP, as well as the local memory, are available to the local processor.

- The memory connected to the common system bus is shared by all processors.
- If an lOP is connected directly to the system bus, the IO devices attached to it may be made available to all processors.

- Only one processor can communicate with the shared memory and other common resources through the system bus at any given time.

- The other processors are kept busy communicating with their local memory and the IO devices. Part of the local memory may be designed as a cache memory attached to the CPU


Local Bus
Common Shared Memory
System Bus Controller
CPU
IOP
Local Memory
SYSTEM BUS
System Bus Controller
CPU
IOP
Local Memory
System Bus Controller
CPU
Local Memory
Local Bus
Local Bus

Multiport Memory

- separate buses between each memory module and each CPU.
- Each processor bus is connected to each memory module.
- A processor bus consists of the address, data, and control lines required to communicate with memory.
- The memory module is said to have four ports and each port accommodates one of the buses.
- The module must have internal control logic to determine which port will have access to memory at any given time.
- Memory access conflicts are resolved by assigning fixed priorities to each memory port.
- The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module.
- Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.

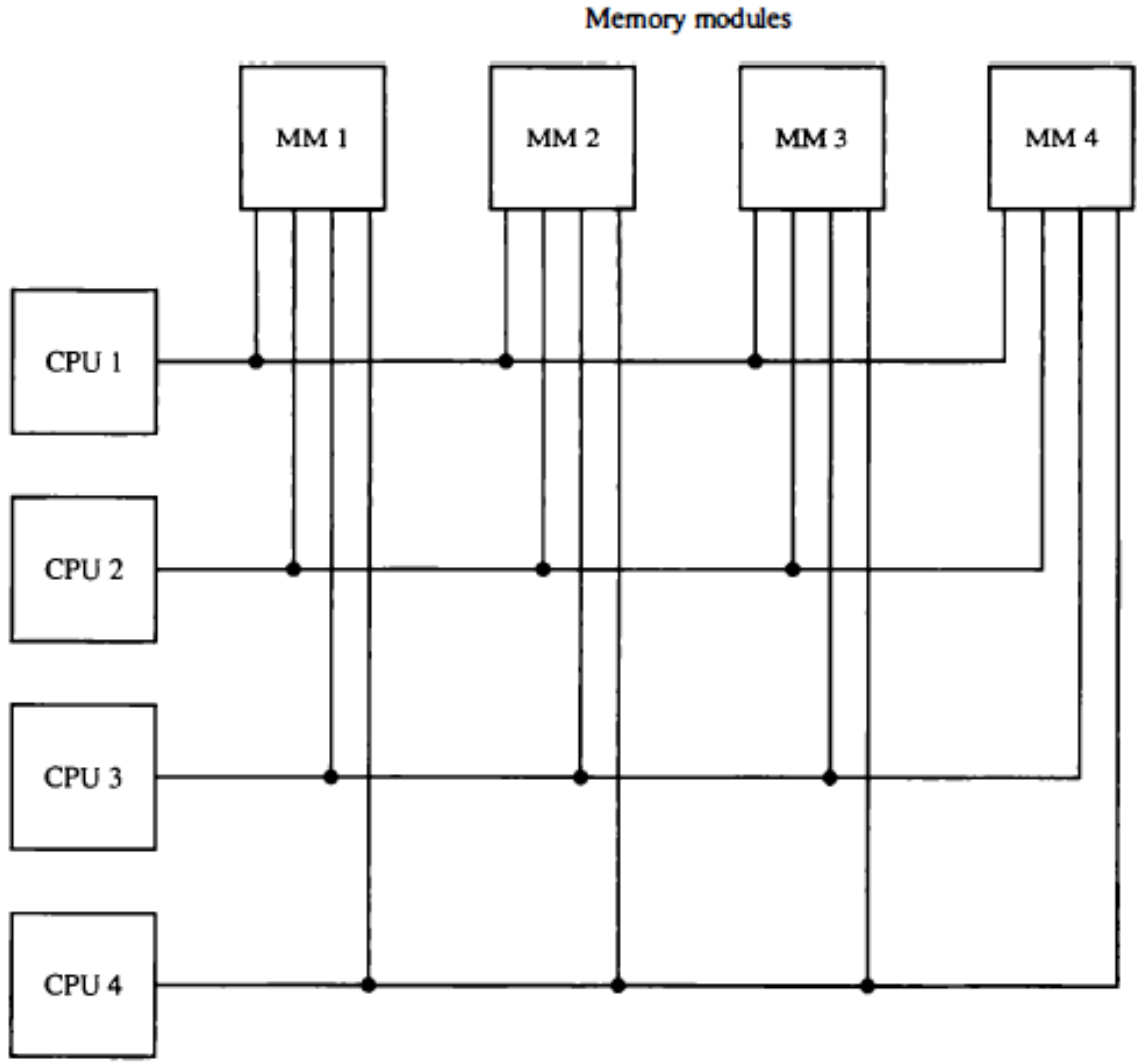


Figure 13-3 Multiport memory organization.

- The advantage of the multi port memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory.

- The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors.

- appropriate for systems with a small number of processors

Crossbar Switch

- consists of a number of crosspoints that are placed at intersections between processor buses and memory module paths.
- The small square in each crosspoint is a switch that determines the path from a processor to a memory module.
- Each switch point has control logic to set up the transfer path between a processor and memory.
- It examines the address that is placed in the bus to determine whether its particular module is being addressed.
- It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

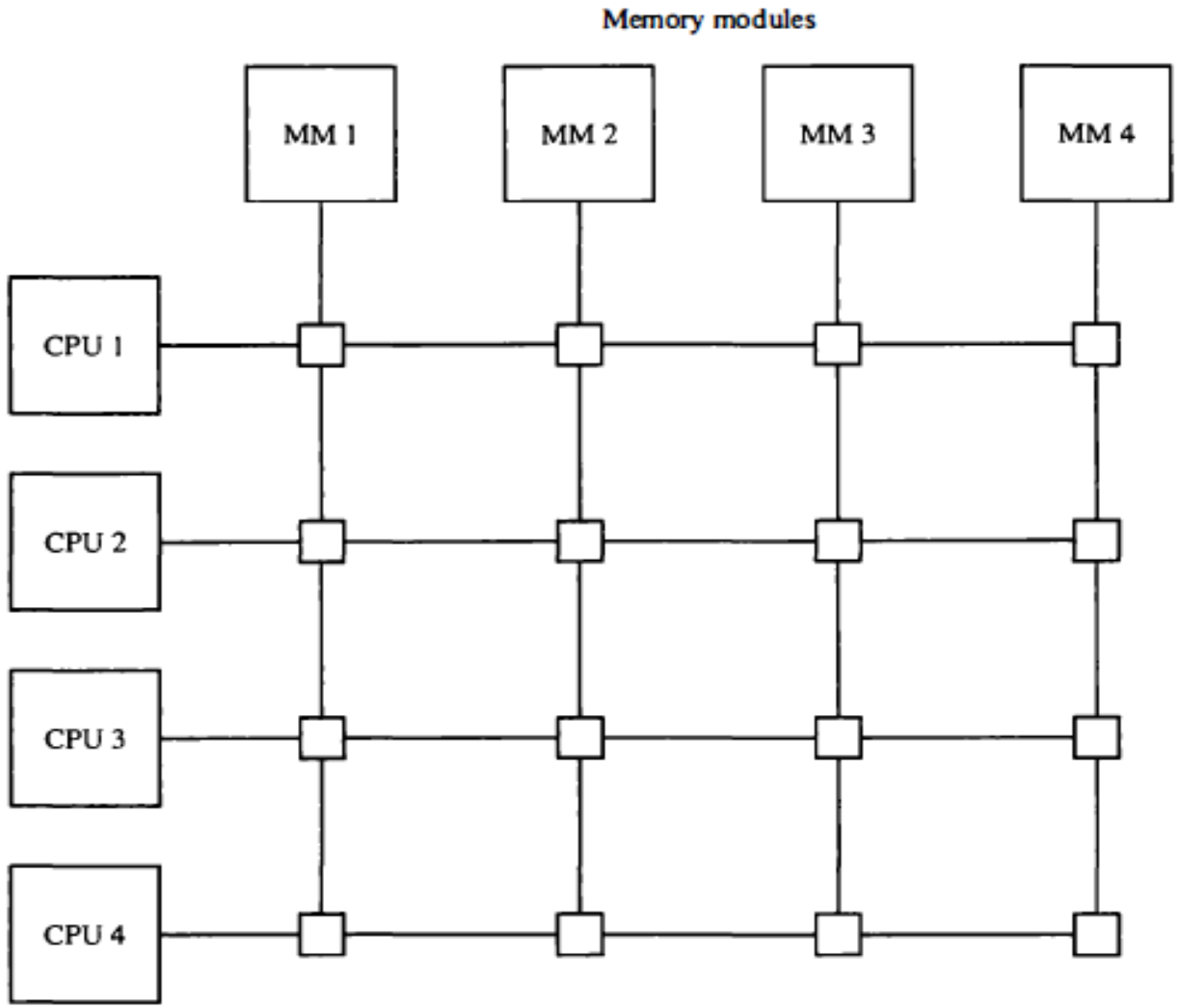
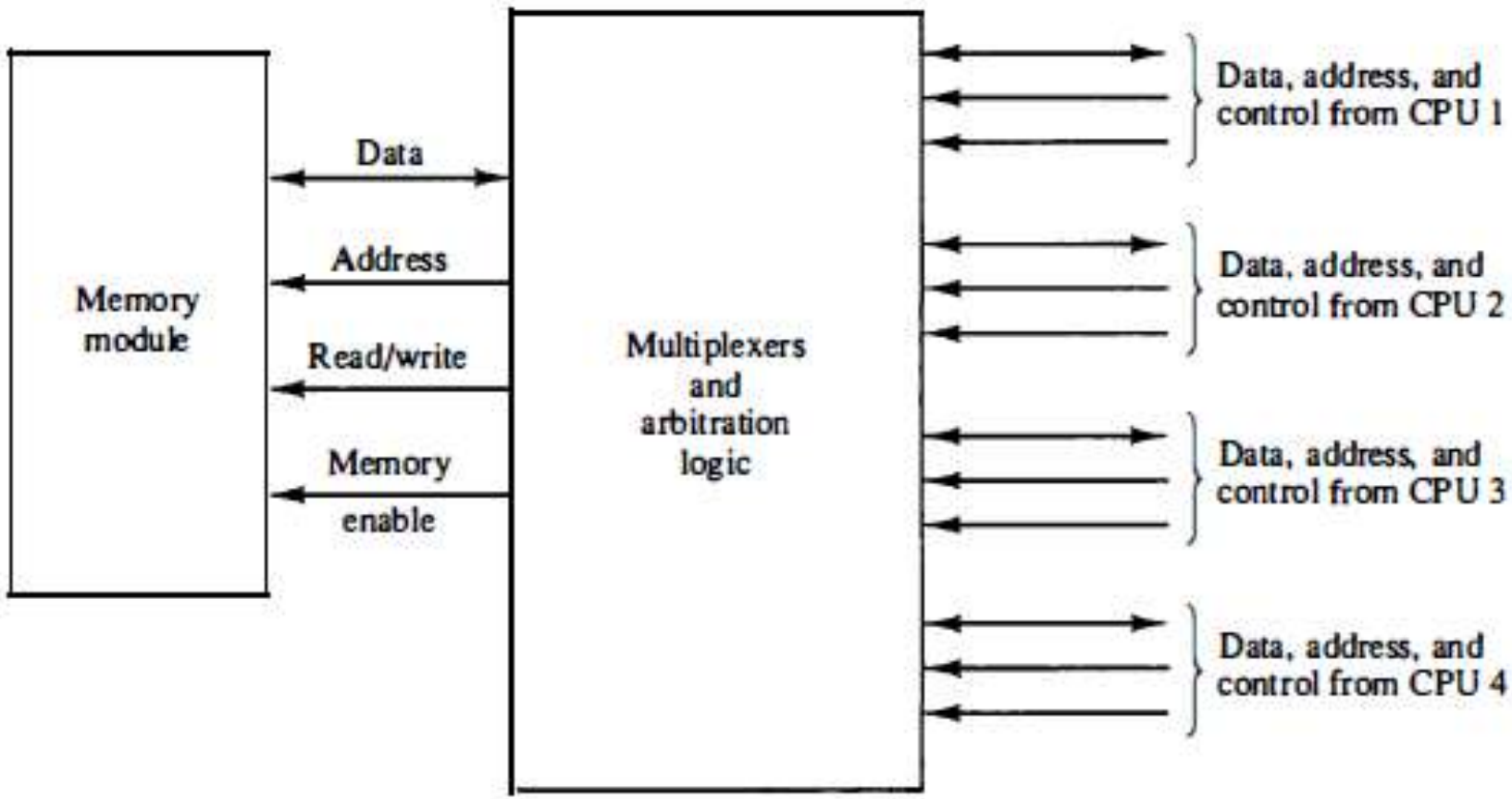


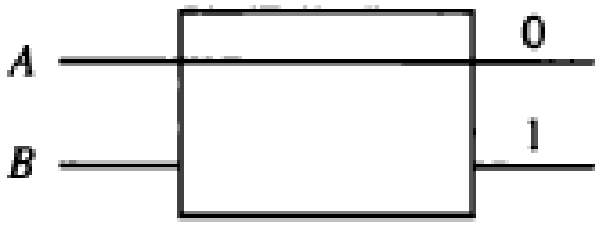
Figure 13-4 Crossbar switch.

Figure 13-5 Block diagram of crossbar switch.

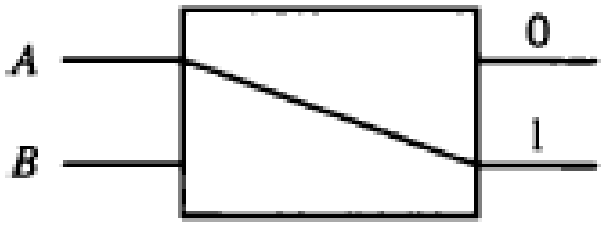


- The circuit consists of multiplexers that select the data, address, and control from one CPU for communication with the memory module.
- Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory.
- The multiplex are controlled with the binary code that is generated by a priority encoder within the arbitration logic.
- A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module.
- However, the hardware required to implement the switch can become quite large and complex.

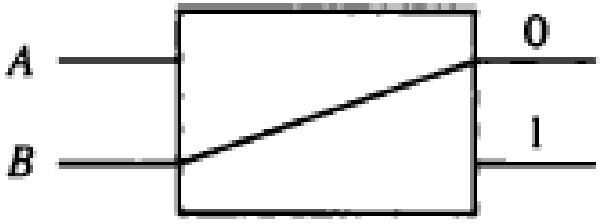
Figure 13-6 Operation of a 2 × 2 interchange switch.



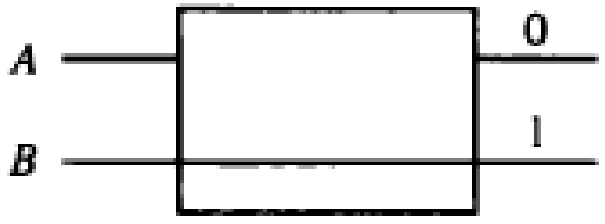
A connected to 0



A connected to 1



B connected to 0



B connected to 1

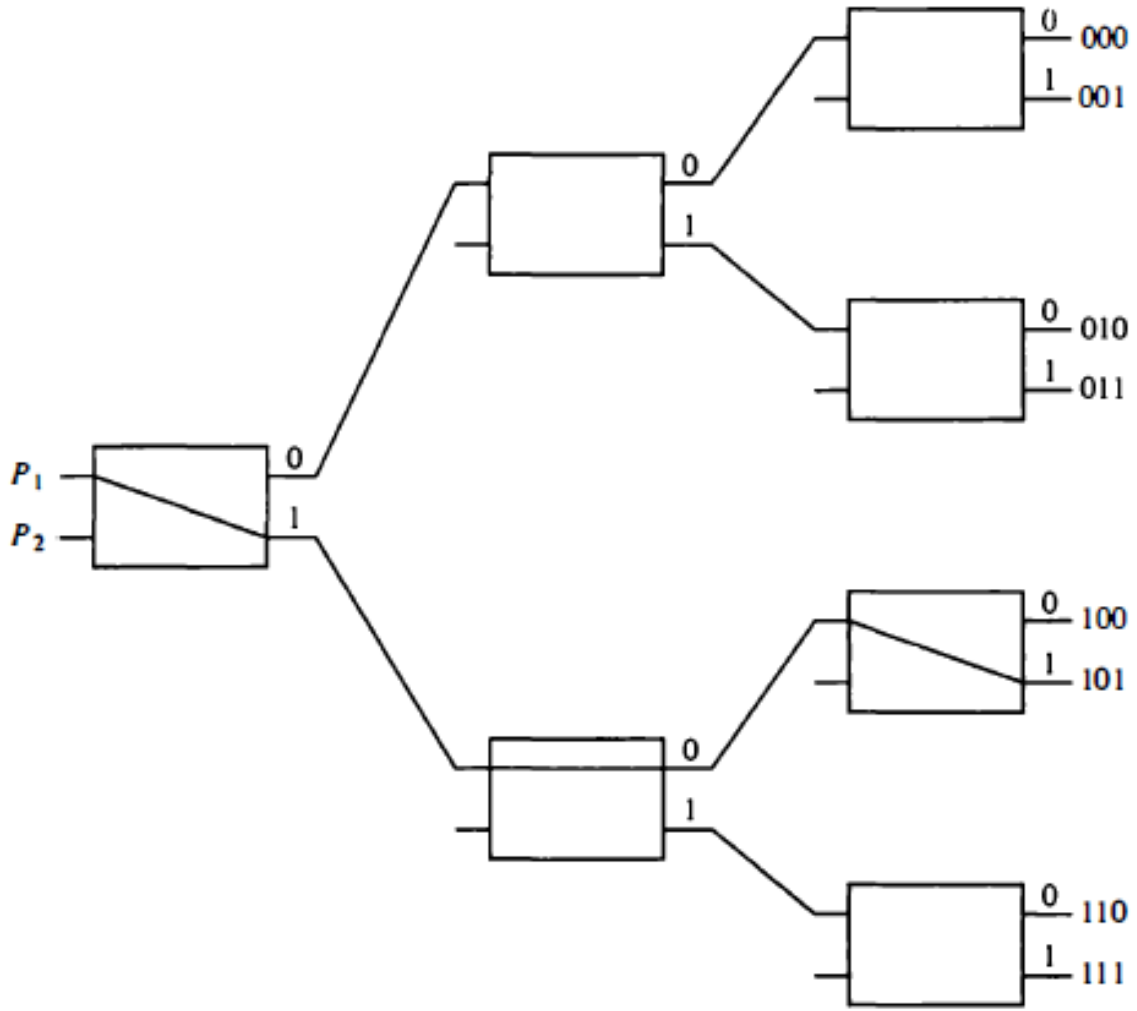


Figure 13-7 Binary tree with 2 × 2 switches.

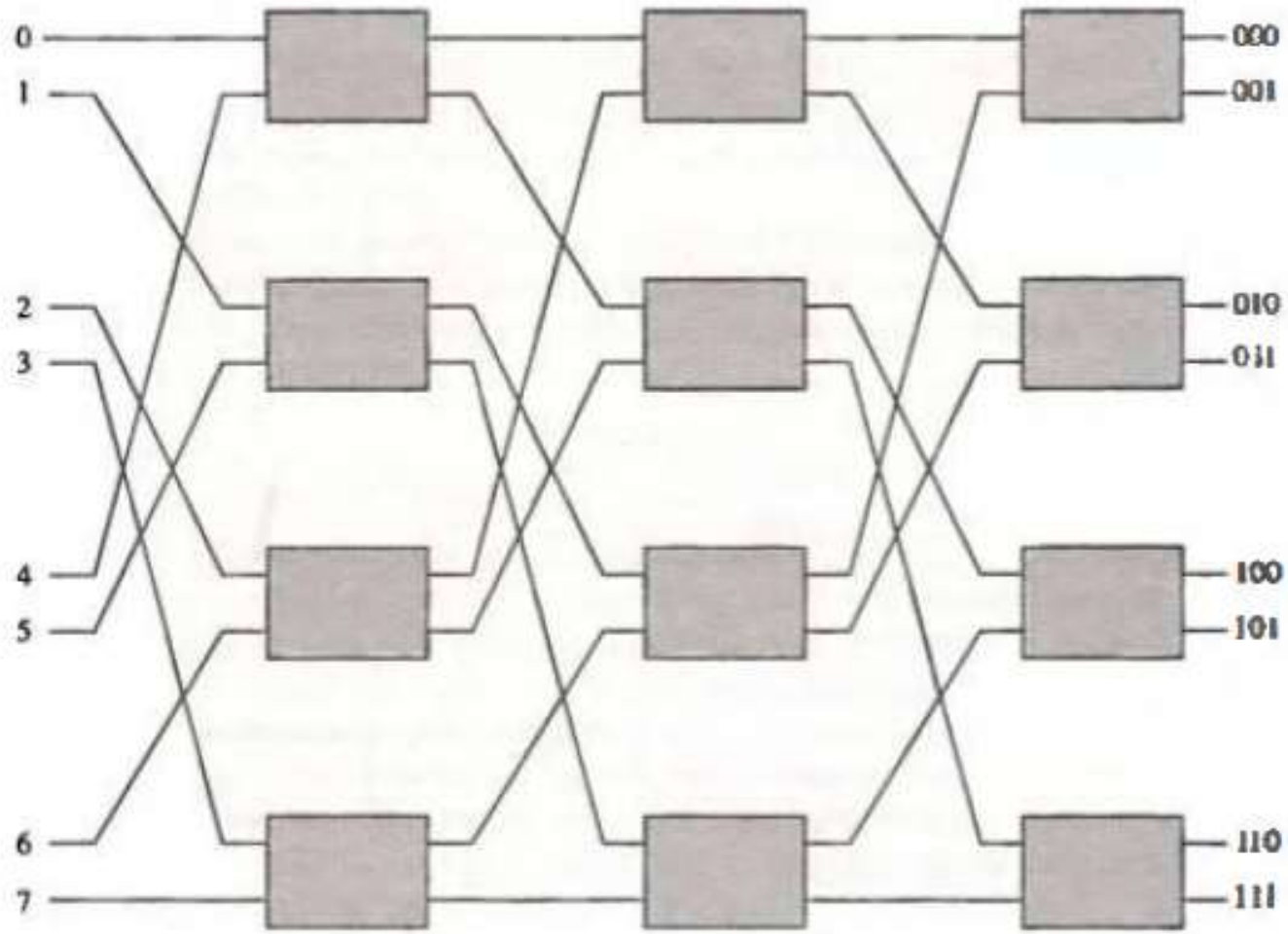
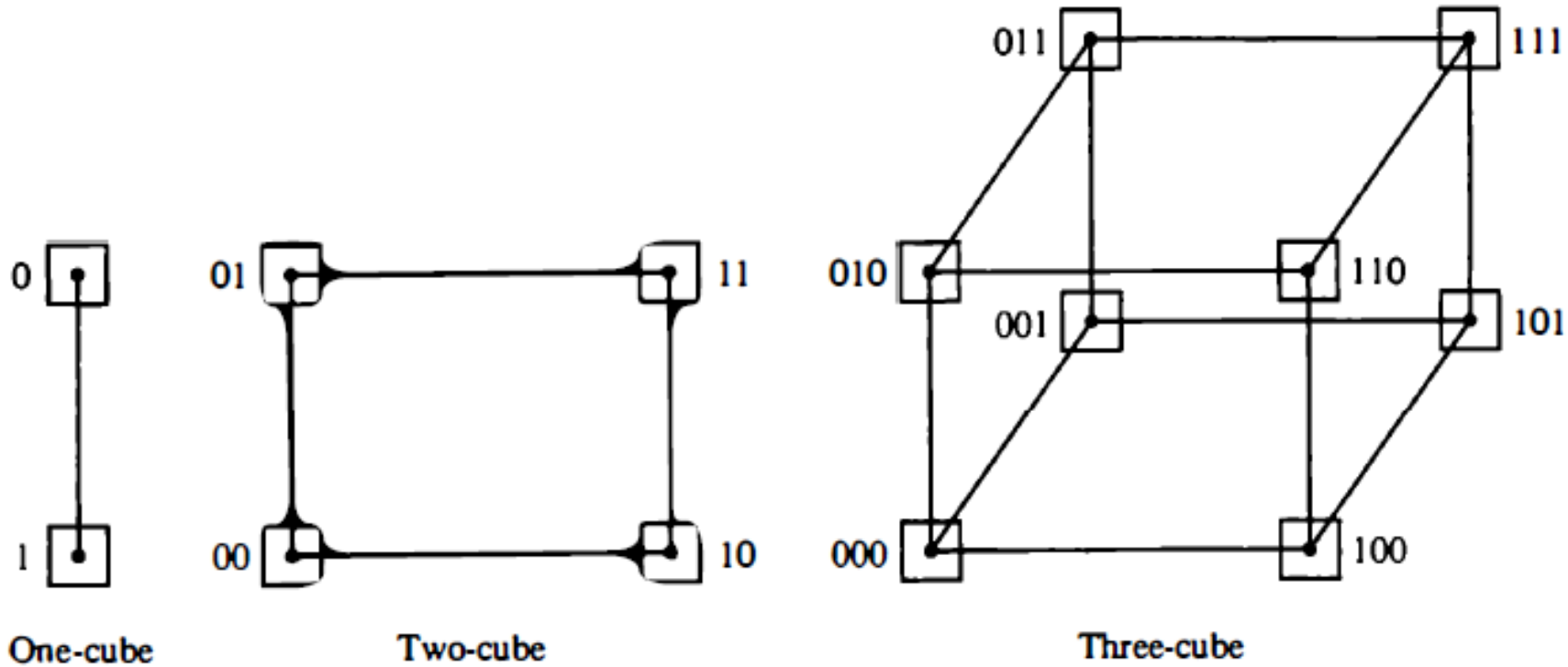
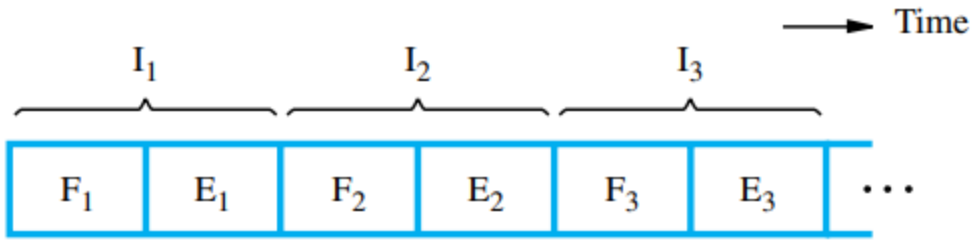


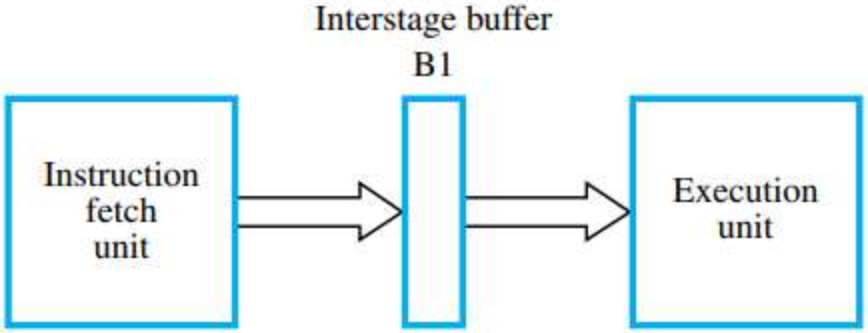
Figure 13-8 8 × 8 omega switching network.

Figure 13-9 Hypercube structures for $n = 1,2,3$.

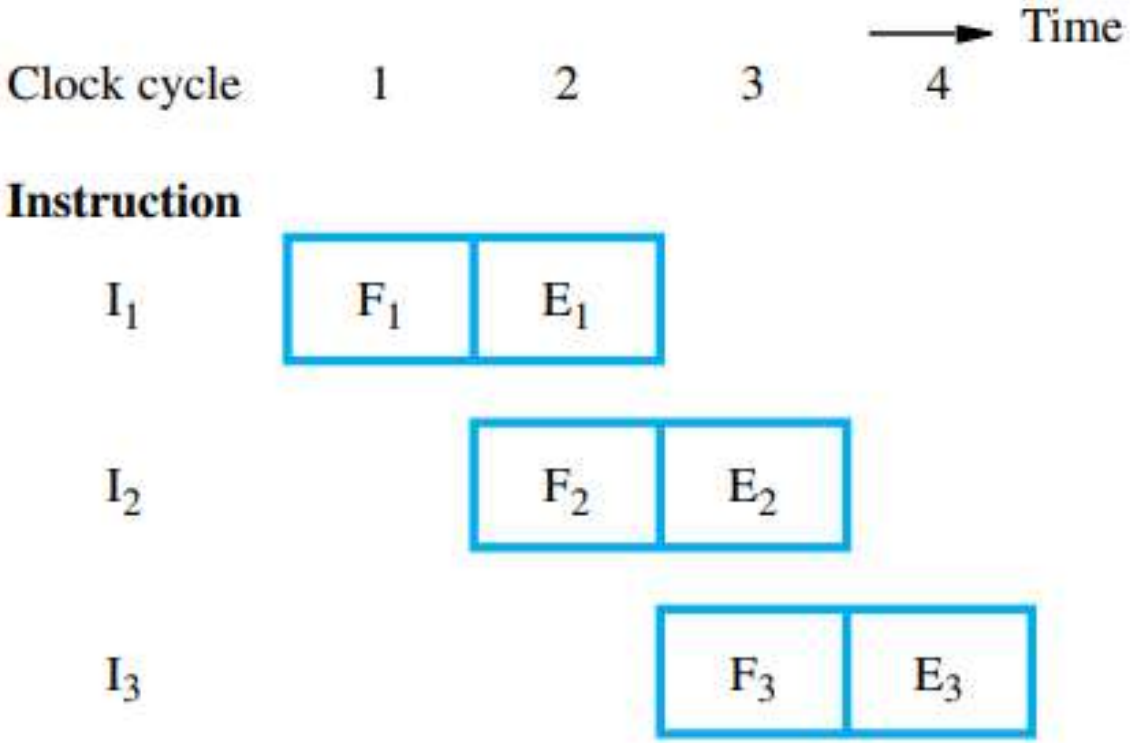




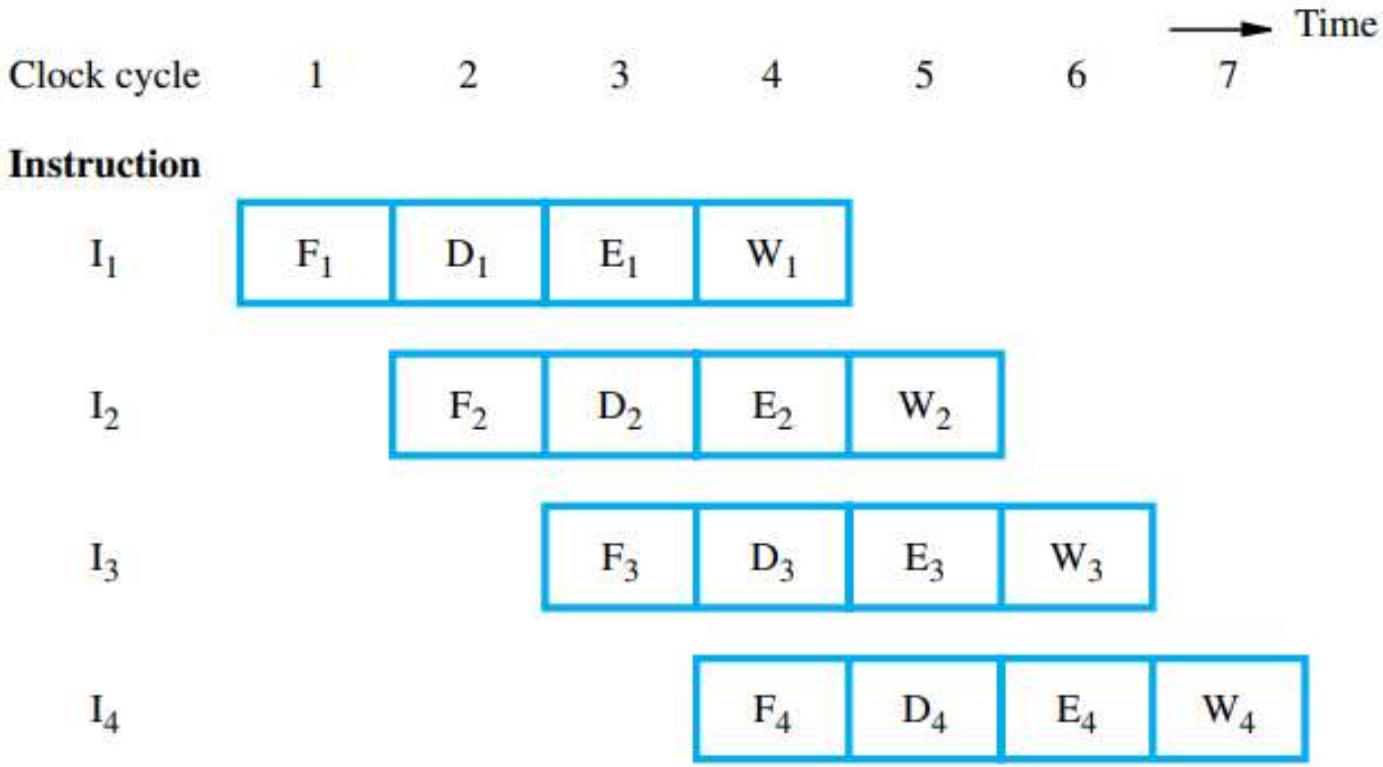
(a) Sequential execution



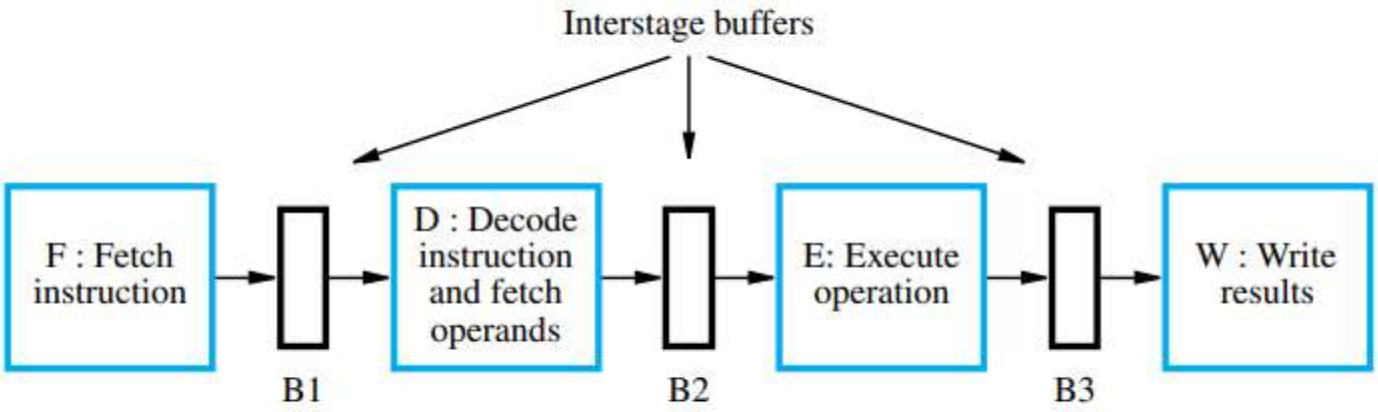
(b) Hardware organization



(c) Pipelined execution



(a) Instruction execution divided into four steps



(b) Hardware organization

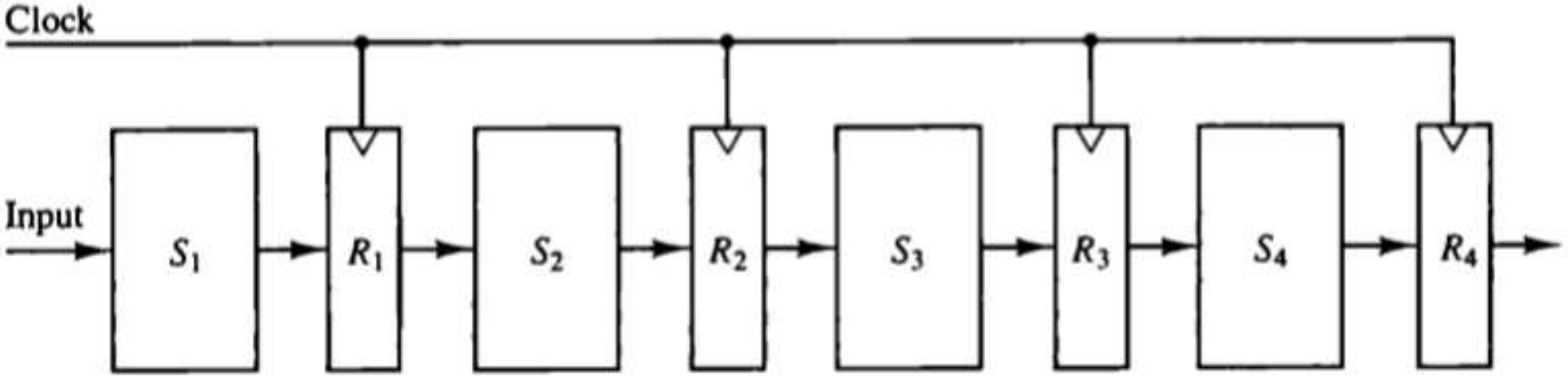


Figure 9-3 Four-segment pipeline.

- a k-segment pipeline with a clock cycle time t, is used to execute n tasks. The first task T1 requires a time equal to kt, to complete its operation since there are k segments in the pipe.

- The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1)tp.

- Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles.

- four segments and six tasks. The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles,

- consider a nonpipeline unit that performs the same operation and takes a time equal to t_n. to complete each task. The total time required for n tasks is nt_n.

- The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

- As the number of tasks increases, n becomes much larger than k - 1, and k + n - 1 approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non pipeline circuits, we will have $t_p = kt_p$.

Including this assumption, the speedup reduces to

$$S = \frac{kt_p}{t_p} = k$$

maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

- Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

- Ai * Bi + C, for i = 1, 2, 3, . .. , 7

- Each suboperation is to be implemented in a segment within a pipeline.

- Each segment has one or two registers and a combinational circuit

- R 1 through R5 are registers that receive new data with every clock pulse.

- The multiplier and adder are combinational circuits.
-
- The suboperations performed

- R1<--Ai, R2<--Bi R3<--R1*R2, R4<--Ci
- R5<--R3 + R4

Figure 9-2 Example of pipeline processing.

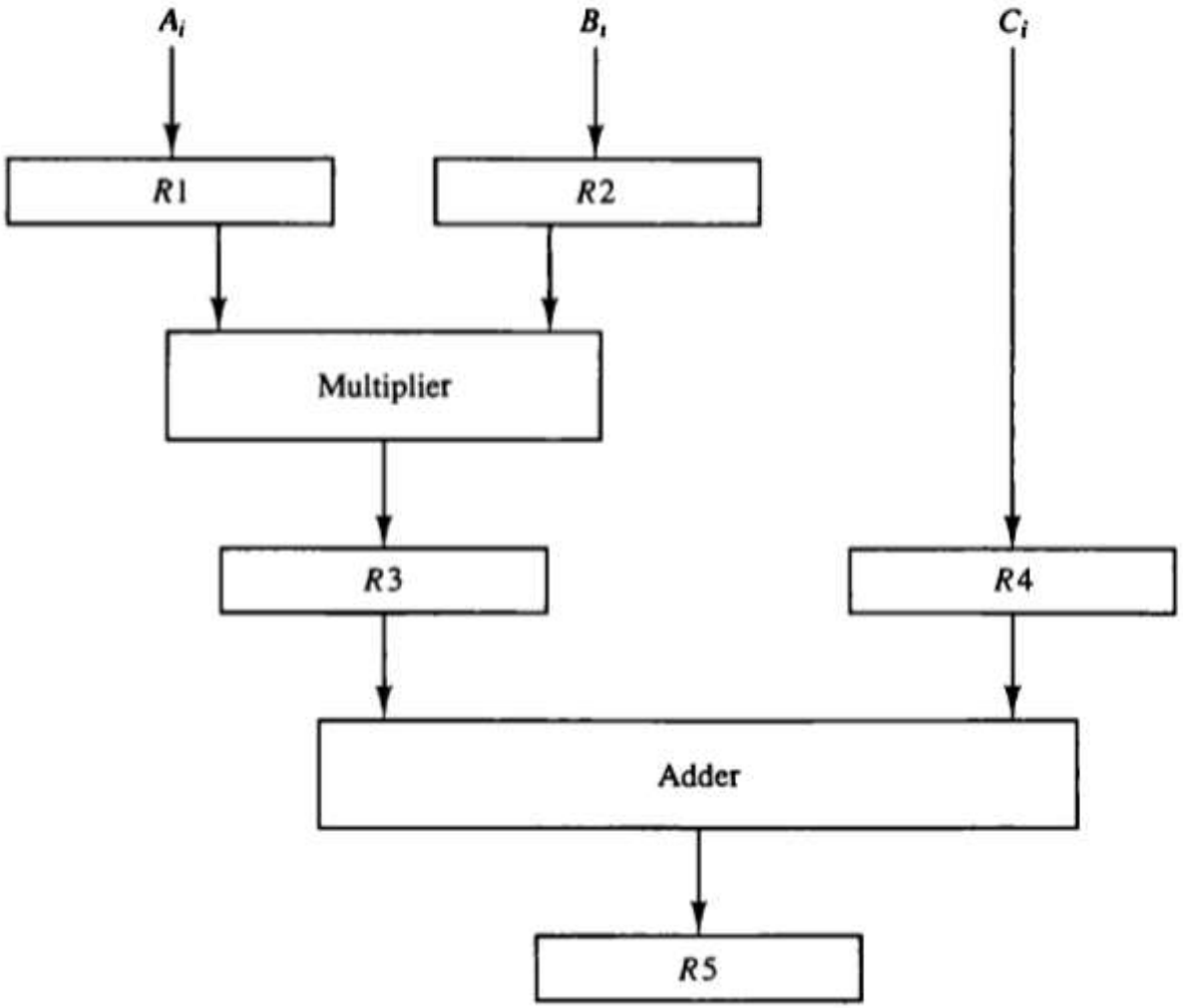


TABLE 9-1 Content of Registers in Pipeline Example

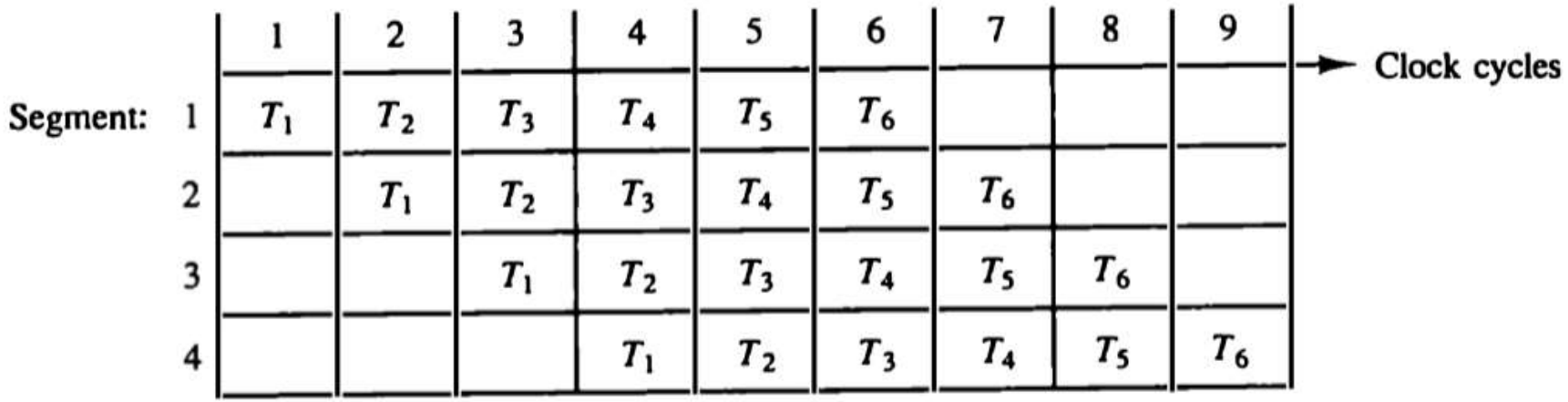
Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

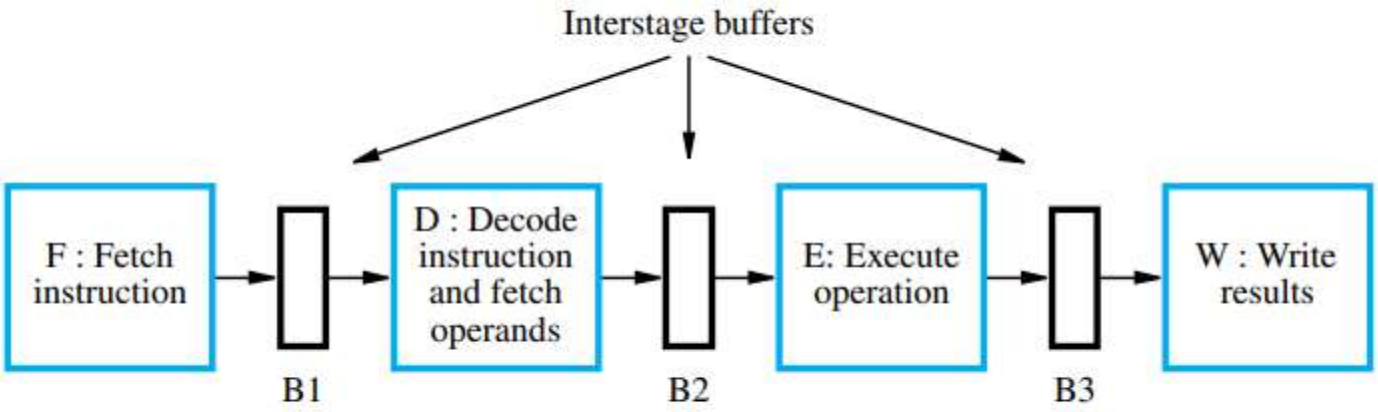
space-time diagram

- shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline

- The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.

Figure 9-4 Space-time diagram for pipeline.





(b) Hardware organization

hazards

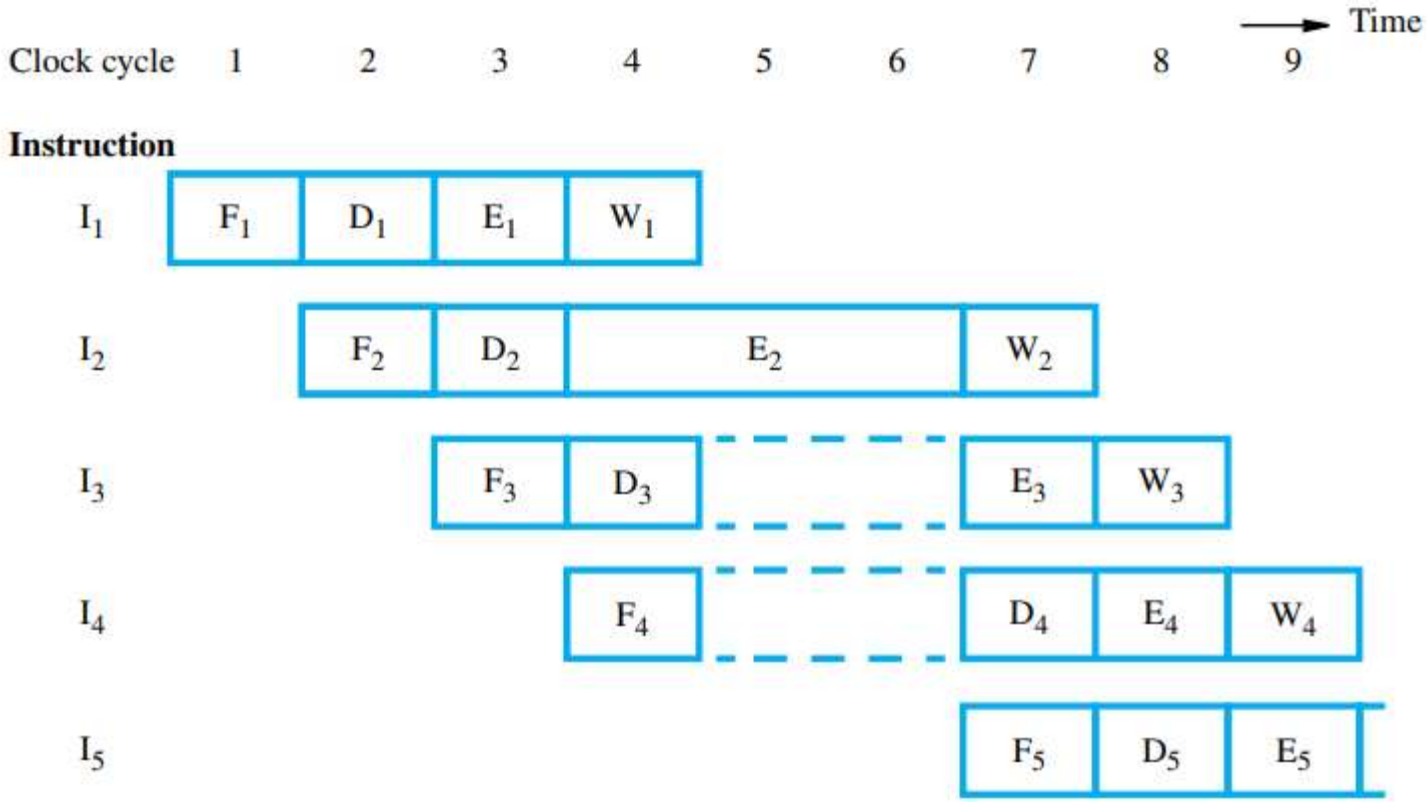


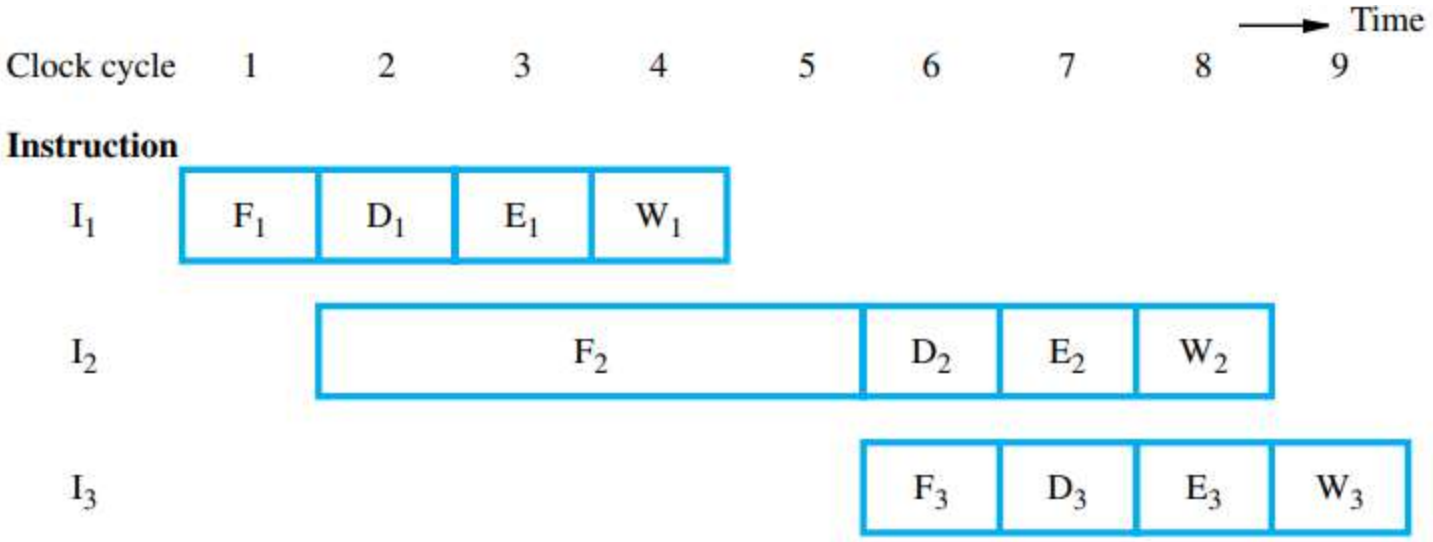
Figure 8.3 Effect of an execution operation taking more than one clock cycle.

- For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted.

- Pipelined operation in figure is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7.

- Any condition that causes the pipeline to stall is called a hazard

- The pipeline may also be stalled because of a delay in the availability of an instruction - instruction hazard - cache miss



(a) Instruction execution steps in successive clock cycles

Clock cycle → Time	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

Figure 8.4 Pipeline stall caused by a cache miss in F2.

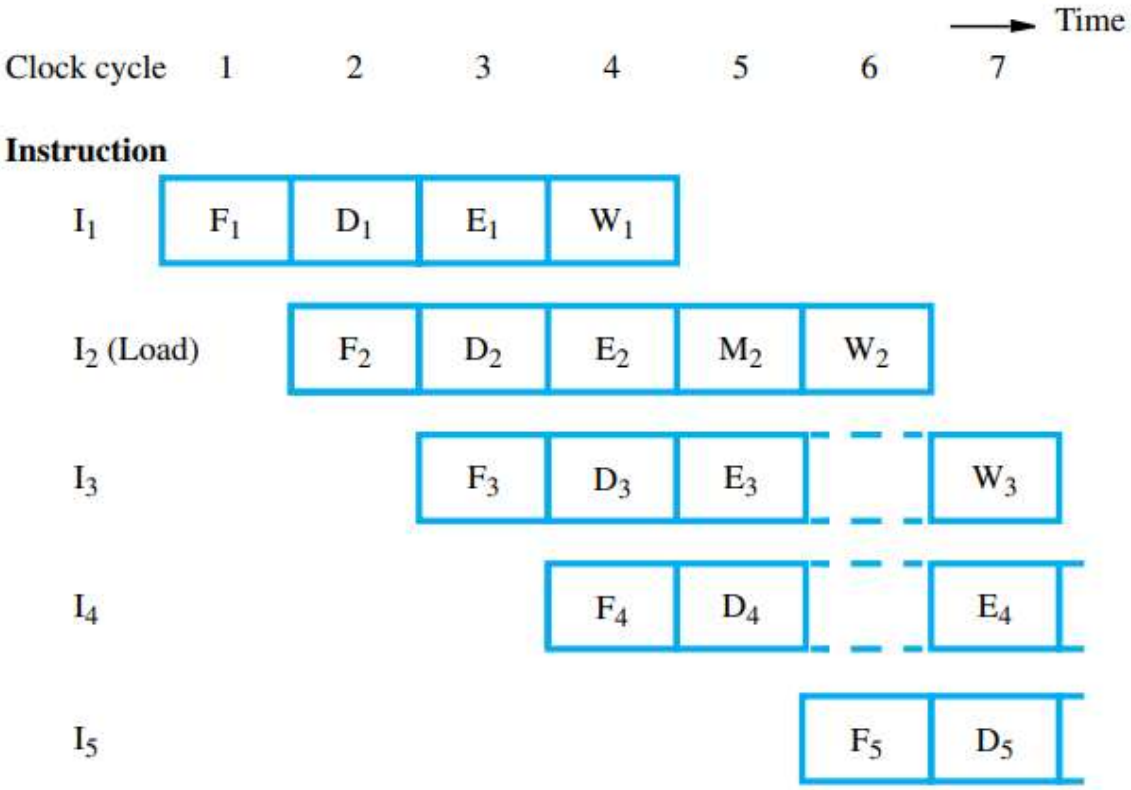


Figure 8.5 Effect of a Load instruction on pipeline timing.

- Structural hazard - This is the situation when two instructions require the use of a given hardware resource at the same time – memory

- One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched.

- Load X(R1),R2

- The memory address, X+[R1], is computed in step E2 in cycle 4, then memory access takes place in cycle 5.

- The operand read from memory is written into register R2 in cycle 6.

- This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5).

- It causes the pipeline to stall for one cycle, because both instructions I2 and I3 require access to the register file in cycle 6

DATA HAZARDS

- a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason
- A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline.
- As a result some operation has to be delayed, and the pipeline stalls.
- Consider a program that contains two instructions, I1 followed by I2.
- When this program is executed in a pipeline, the execution of I2 can begin before the execution of I1 is completed.

- Assume that A = 5, and consider the following two operations:

- $A \leftarrow 3 + A$ $B \leftarrow 4 \times A$ When these operations are performed in the order given, the result is B = 32.

- But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result.

- If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction.

- On the other hand, the two operations A ← 5 × C B ← 20 + C can be performed concurrently, because these operations are independent.

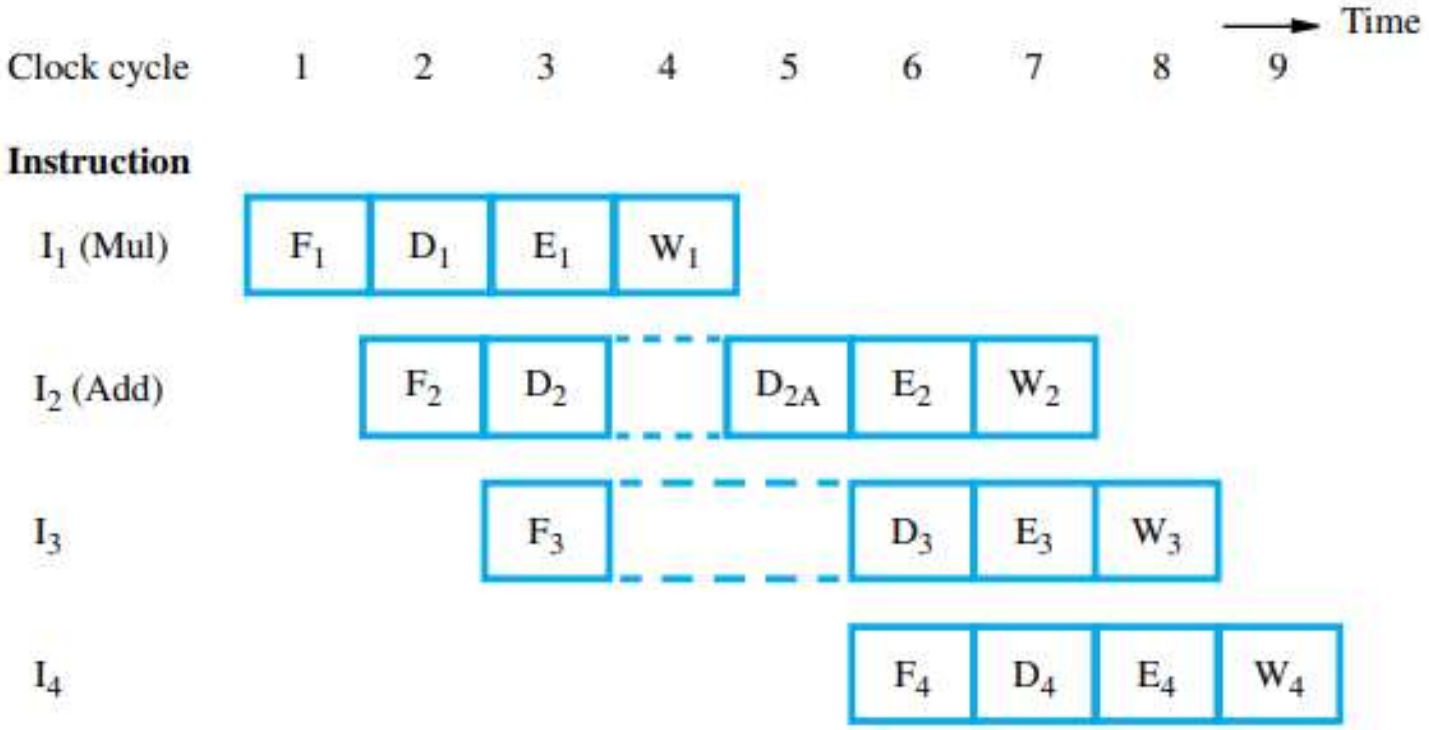
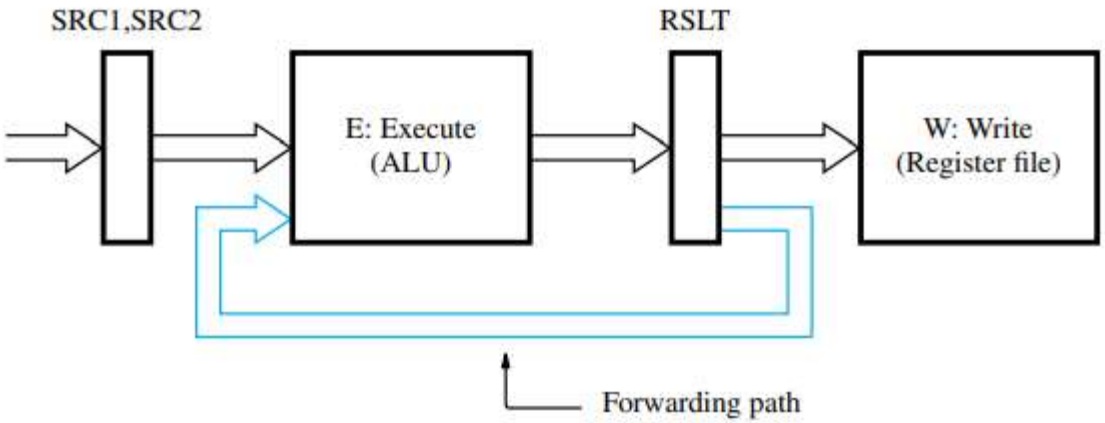


Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

OPERAND FORWARDING

- these data are available at the output of the ALU once the Execute stage completes step E1.

- Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I1 to be forwarded directly for use in step E2.



(b) Position of the source and result registers in the processor pipeline

INSTRUCTION / CYCLE	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

- In cycle 4, instructions I_1 and I_4 are trying to access same resource (Memory) which introduces a resource conflict.
- To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available.

- This wait will introduce stalls in the pipeline