UNIT - V: PIPELINE & VECTOR PROCESSING AND MULTI PROCESSORS

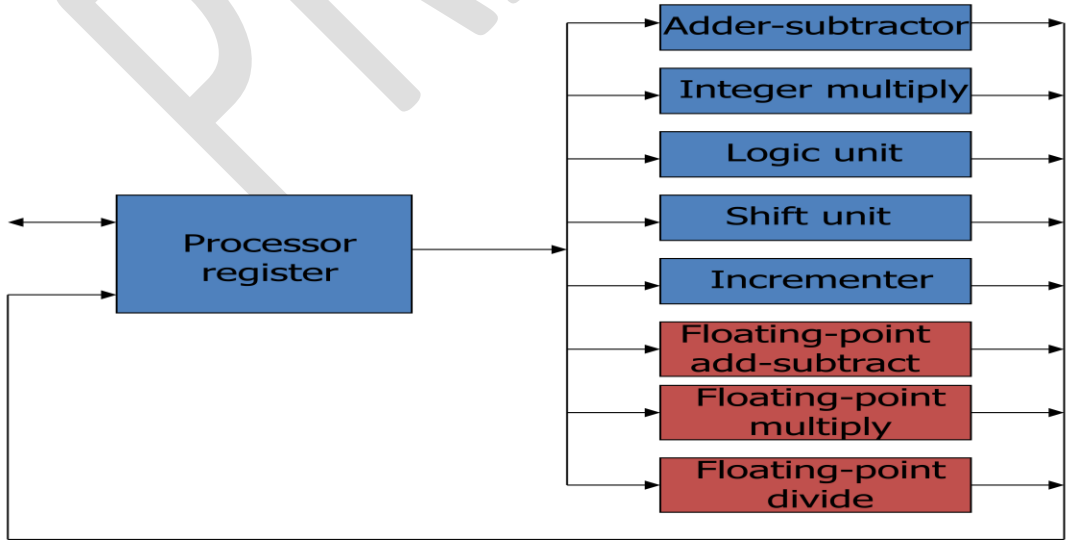
(09 periods)

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processors.

Multiprocessors: Characteristics of Multiprocessors, Interconnection Structures, Inter-processor Arbitration, Inter-Processor Communication and Synchronization.

5.1 Parallel Processing:

- *Parallel processing* is a term used to denote a large class of techniques that are used to provide simultaneous *data-processing tasks* for the purpose of increasing the computational speed of a computer system.
- The purpose of parallel processing is to *speed up the computer processing capability* and *increase its throughput,* that is, the amount of processing that can be accomplished during a given interval of time.
- The amount of hardware increases with parallel processing, and with it, the cost of the system increases.
- Parallel processing can be viewed from various levels of complexity.
    - At the lowest level, we distinguish between parallel and serial operations by the type of registers used. e.g. shift registers and registers with parallel load
    - At a higher level, it can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously.
- Fig. below shows one possible way of separating the execution unit into eight functional units operating in parallel.
    - A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.



`The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented. A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

- There are a variety of ways that parallel processing can be classified. it can be considered from the
    - Internal organization of the processors
    - Interconnection structure between processors
    - The flow of information through the system

One classification introduced by M.]. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor.

The sequence of instructions read from memory constitutes an *instruction stream.* The operations performed on the data in the processor constitute a *data stream*. Parallel processing may occur in the instruction stream, in the datastream, or in both. Flynn's classification divides computers into four major
groups as follows:

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

Single instruction stream, single data stream (SISD)

- Represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- Instructions are executed *sequentially* and the system may or may not have internal parallel processing capabilities.
- parallel processing may be achieved by means of *multiple functional units* or by *pipeline processing*.

Single instruction stream, multiple data stream (SIMD)

- Represents an organization that includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different items of data.
- The shared memory unit must contain *multiple modules* so that it can communicate with all the processors simultaneously.

Multiple instruction stream, single data stream (MISD)

- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

Multiple instruction stream, multiple data stream (MIMD)

- MIMD organization refers to a computer system capable of processing several programs at the same time. e.g. multiprocessor and multicomputer system

- We consider parallel processing under the following main topics:
    - Pipeline processing
        - Is an implementation technique where arithmetic sub operations or the phases of a computer instruction cycle overlap in execution.
    - Vector processing
        - Deals with computations involving large vectors and matrices.
    - Array processing
        - Perform computations on large arrays of data.

5.2 Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. The name "pipeline" implies aflow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.

- Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an *input register* followed by a *combinational circuit*.
    - The register holds the data.
    - The combinational circuit performs the suboperation in the particular segment.
- A clock is applied to all registers after *enough time* has elapsed to perform all segment activity.
- **Example**
- The pipeline organization will be demonstrated by means of a simple example.
    - To perform the combined multiply and add operations with a stream of numbers

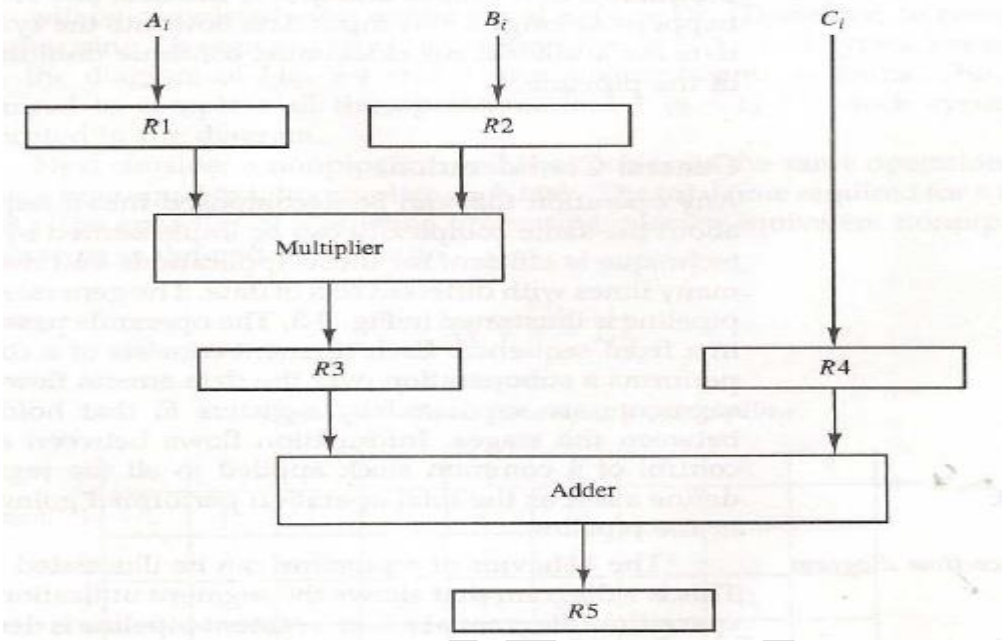
      $A_i * B_i + C_i$ for $i = 1, 2, 3, \ldots, 7$

- Each suboperation is to be implemented in a segment within a pipeline.
    - $R1 \leftarrow A_i$, $R2 \leftarrow B_i$ Input A_i and B_i
    - $R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$ Multiply and input C_i
    - $R5 \leftarrow R3 + R4$ Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A1 and 31 into R1 andR2. The second clock pulse transfers the product of R1 and R2 into R3 and C1into R4. The same clock pulse transfers A2 and B2 into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places A3 and B3into R1 and R2, transfers the product of R1 and R2 into R3, transfers C2 intoR4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

- Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2.

Figure 9-2 Example of pipeline processing.



- The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1.

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	—	—	—	—	$A_7 * B_7 + C_7$

General considerations

- Any operation that can be decomposed into a sequence of sub operations of about the *same complexity* can be implemented by a pipeline processor.
- The general structure of a four-segment pipeline is illustrated in Fig. 9-3.
- We define a *task* as the total operation performed going through all the segments in the pipeline.
- The behavior of a pipeline can be illustrated with a *space-time* diagram.
    - It shows the segment utilization as a function of time.

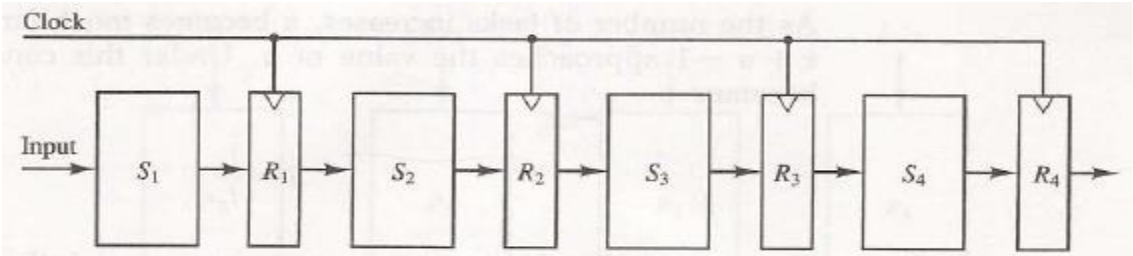


Figure 9-3 Four-segment pipeline.

- The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4.
- Where a *k*-segment pipeline with a clock cycle time t_p is used to execute *n* tasks.
  - The first task T_1 requires a time equal to kt_p to complete its operation.
  - The remaining *n*-1 tasks will be completed after a time equal to $(n\text{-}1)t_p$
  - Therefore, to complete *n* tasks using a *k*-segment pipeline requires $k+(n\text{-}1)$ 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.

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

- The *speedup of a pipeline processing* over an equivalent nonpipeline processing is defined by the ratio $S = nt_n/(k+n\text{-}1)t_p$.
- If *n* becomes much larger than *k*-1, the speedup becomes $S = t_n/t_p$.
- If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, i.e.,$t_n = kt_p$, the speedup reduces to $S=kt_p/t_p=k$.
- This shows that the theoretical maximum speedup that a pipeline can provide is *k*, where *k* is the number of segments in the pipeline.

- To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct *k* identical units that will be operating in parallel.
- This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel.
- Instead of operating with the *input data in sequence* as in a pipeline, the parallel circuits accept four input data items *simultaneously* and perform four tasks at the same time.

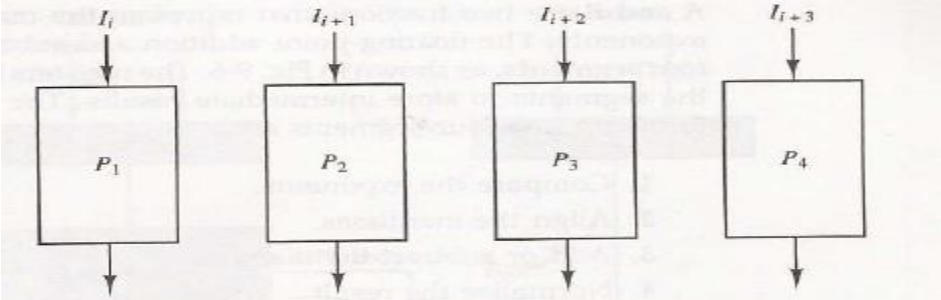


Figure 9-5 Multiple functional units in parallel.

5.3 Arithmetic Pipeline

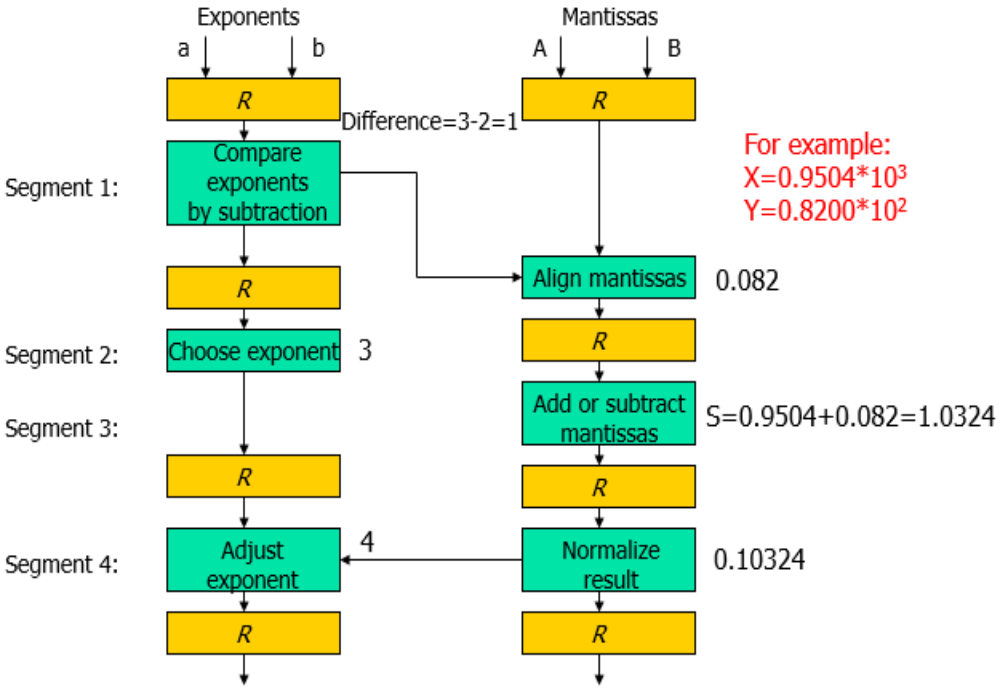
- There are various reasons why the pipeline cannot operate at its maximum theoretical rate.
  - Different segments may take different times to complete their sub operation.
  - It is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit.
- There are two areas of computer design where the pipeline organization is applicable.
  - Arithmetic pipeline
  - Instruction pipeline

Arithmetic Pipeline: Introduction

- Pipeline arithmetic units are usually found in very high speed computers
  - Floating–point operations, multiplication of fixed-point numbers, and similar computations in scientific problem
- Floating–point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5.
- An example of a pipeline unit for floating-point addition and subtraction is showed in the following:
  - The inputs to the floating-point adder pipeline are two normalized floating-point binary number

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

- A and B are two fractions that represent the mantissas, a and b are the exponents.
- The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6.
- The suboperations that are performed in the four segments are:
  - *Compare the exponents*
    - The larger exponent is chosen as the exponent of the result.
  - *Align the mantissas*



- The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right.

- *Add or subtract the mantissas*
- *Normalize the result*
  - 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.

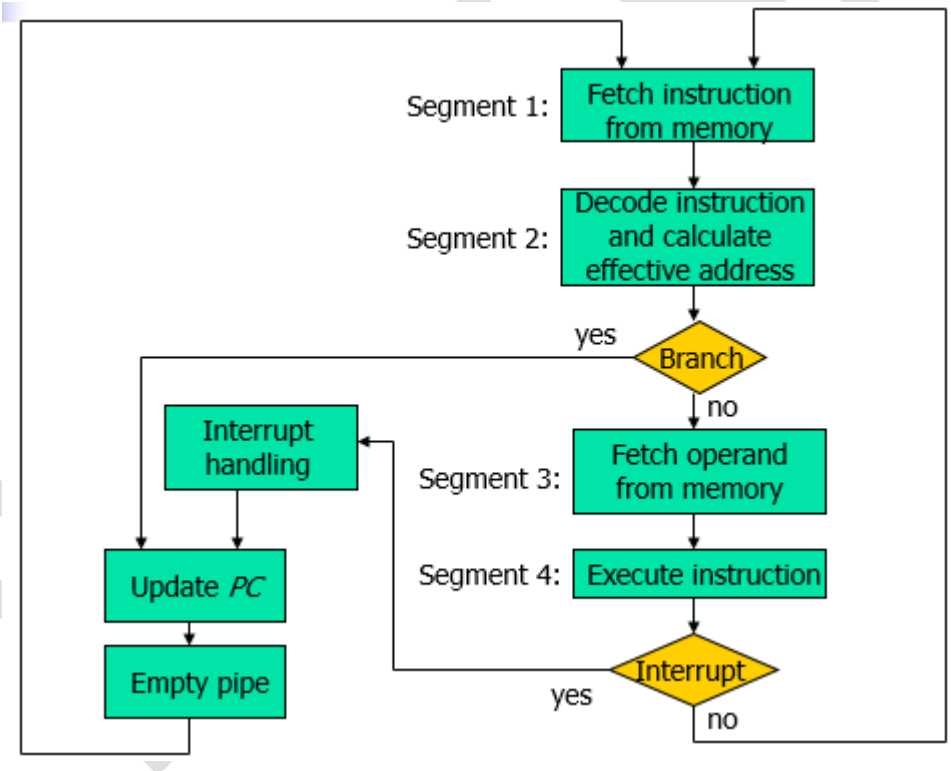
4.4 Instruction Pipeline

- Introduction:
- Pipeline processing can occur not only in the *data stream* but in the *instruction* as well.
- Consider a computer with an instruction fetch unit (FIFO)and an instruction execution unit(PC) designed to provide a *two-segment* pipeline.
- Computers with complex instructions require other phases in addition to above phases to process an instruction completely.
- In the most general case, the computer needs to process each instruction with the following sequence of steps.
  - Fetch the instruction from memory.
  - Decode the instruction.
  - Calculate the effective address.
  - Fetch the operands from memory.
  - Execute the instruction.
  - Store the result in the proper place.
- There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate.

- Different segments may take different times to operate on the incoming information.
- Some segments are skipped for certain operations.
- Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

Example: four-segment instruction pipeline:

- Assume that:
  - The 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.
- Fig 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline.
  - Thus up to four suboperations 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 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.
  - Similarly, an interrupt request will cause the pipeline to empty and start again from a new address value.
- Fig. 9-8 shows the operation of the instruction pipeline.

The time in thehorizontal axis is divided into steps of equal duration. The four segments arerepresented in the diagram with an abbreviated symbol.

1. F1 is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates theeffective address.
3. Fo is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

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

FI: the segment that fetches an instruction
DA: the segment that decodes the instruction and calculate the effective address
FO: the segment that fetches the operand
EX: the segment that executes the instruction

It is assumed that the processor has separate instruction and data memoriesso that the operation in F1 and PC can proceed at the same time. In the absenceof a branch instruction, each segment operates on different instructions. Thus,in step 4, instruction 1 is being executed in segment EX; the operand forinstruction 2 is being fetched in segment FO; instruction 3 is being decoded insegment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as thisinstruction is decoded in segment DA in step 4, the transfer from F1 to DA ofthe other instructions is halted until the branch instruction is executed in step6. If the branch is taken, a new instruction is fetched in step 7. If the branchis not taken, the instruction fetched previously in step 4 can be used. Thepipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to storethe result of the operation in the data memory while the FO segment needsto fetch an operand. In that case, segment FO must wait until segment EX hasfinished its operation.

- In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.
    - *Resource conflicts* caused by access to memory by two segments at the same time.
        - Can be resolved by using separate instruction and data memories
    - *Data dependency conflicts* arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
    - *Branch difficulties* arise from branch and other instructions that change the value of PC.

Data dependency:

- A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address.
    - A data dependency occurs when an instruction needs data that are not yet available.

        - An address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
    - Pipelined computers deal with such conflicts between data dependencies in a variety of ways.
    - *Hardware interlocks*: an interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
        - This approach maintains the program sequence by using hardware to insert the required delays.
    - *Operand forwarding*: uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.
        - This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.
    - *Delayed load*: 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 no-operation instructions.

Handling of branch instructions

- One of the major problems in operating an instruction pipeline is the occurrence of branch instructions.
    - 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.
- Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching.

- *Prefetch target instruction*: To prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed.
- *Branch target buffer(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.
- *Loop buffer*: This is a small very high speed register file maintained by the instruction fetch segment of the pipeline.
- *Branch prediction*: A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- *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.
    - A procedure employed in most RISC processors.
    - e.g. no-operation instruction

-

5.5 RISC Pipeline

Among the characteristics attributed to RISC is its ability to use an efficient instruction pipeline. The simplicity of the instruction set can be utilized toimplement an instruction pipeline using a small number of suboperations,with each being executed in one clock cycle. Because of the fixed-lengthinstruction format, the decoding of the operation can occur at the same timeas the register selection. All data manipulation instructions have register-to-register operations. Since all operands are in registers, there is no need forcalculating an effective address or fetching of operands from memory. Therefore, the instruction pipeline can be implemented with two or three segments.One segment fetches the instruction from program memory, and the othersegment executes the instruction in the ALU. A third segment may be used tostore the result of the ALU operation in a destination register.

- The data transfer instructions in RISC are limited to load and store instructions.
    - These instructions use register indirect addressing. They usually need three or four stages in the pipeline.
    - To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories.
    - Cache memory: operate at the same speed as the CPU clock
- One of the major advantages of RISC is its ability to execute instructions at the rate of one per clock cycle.
    - In effect, it is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.
    - RISC can achieve pipeline segments, requiring just one clock cycle.
- *Compiler* supported that translates the high-level language program into machine language program.
    - Instead of designing hardware to handle the difficulties associated with data conflicts and branch penalties.
    - RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems.

Example: Three-Segment Instruction Pipeline

- A typical set of instructions for a RISC processor are discussed earlier.
- Thee are three types of instructions:
    - The data manipulation instructions: operate on data in processor registers
    - The data transfer instructions:
    - The program control instructions:

Now consider the hardware operation for such a computer.

- The *control section* fetches the instruction from program memory into an instruction register.
    - The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected.

- The processor unit consists of a number of registers and an arithmetic logic unit (ALU).
- A data memory is used to load or store the data from a selected register in the register file.
- The instruction cycle can be divided into three suboperations and implemented in three segments:
    - I: Instruction fetch
        - Fetches the instruction from program memory
    - A: ALU operation
        - The instruction is decoded and an ALU operation is performed.
        - It performs an operation for a data manipulation instruction.
        - It evaluates the effective address for a load or store instruction.
        - It calculates the branch address for a program control instruction.
    - E: Execute instruction
        - Directs the output of the ALU to one of three destinations, depending on the decoded instruction.
        - It transfers the result of the ALU operation into a destination register in the register file.
        - It transfers the effective address to a data memory for loading or storing.
        - It transfers the branch address to the program counter.

Delayed Load

- Consider the operation of the following four instructions:
    1. LOAD: R1 ← M[address 1]
    2. LOAD: R2 ← M[address 2]
    3. ADD: R3 ← R1 +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.
- This can be seen from the timing of the pipeline shown in Fig. 9-9(a).

The E segment in clock cycle 4 is in a process of placing the memorydata into R2. The A segment in clock cycle 4 is using the data from R2, but thevalue in R2 will not be the correct value since it has not yet been transferredfrom memory. It is up to the compiler to make sure that the instructionfollowing the load instruction uses the data fetched from memory. If thecompiler cannot find a useful instruction to put after the load, it inserts a no-op(no-operation) instruction. This is a type of instruction that is fetched frommemory but has no operation, thus wasting a clock cycle. This concept ofdelaying the use of the data loaded from memory is referred to as *delayed load*.

Figure 9-9(b) shows the same program with a no-op instruction insertedafter the load to R2 instruction. The data is loaded into R2 in clock cycle 4. Theadd instruction uses the value of R2 in step 5. Thus the no-op instruction isused to advance one clock cycle in order to compensate for the data conflict inthe pipeline. (Note that no operation is performed in segment A during clockcycle 4 or segment E during clock cycle 5.) The advantage of the delayed loadapproach is that the data dependency is taken care of by the compiler

ratherthan the hardware. This results in a simpler hardware segment since thesegment does not have to check if the content of the register being accessedis currently valid or not.

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

9 (a) Pipeline timing with data conflict

	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

9 (b) Pipeline timing with delayed load

Delayed Branch

- The method used in most RISC processors is to rely on the *compiler to redefine the branches* so that they take effect at the proper time in the pipeline. This method is referred to as *delayed branch*.
- The compiler is designed to analyze the instructions *before and after the branch* and *rearrange the program sequence* by inserting useful instructions in the delay steps.
- It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert *no-op* instructions.

An Example of Delayed Branch

- The program for this example consists of five instructions.
  - Load from memory to R1
  - Increment R2
  - Add R3 to R4
  - Subtract R5 from R6
  - Branch to address X
- In Fig. 9-10(a) the compiler inserts *two no-op instructions* after the branch.
- The branch address X is transferred to PC in clock cycle 7.
- The program in Fig. 9-10(b) is rearranged by placing the add and subtract instructions *after the branch instruction*.
- PC is updated to the value of X in clock cycle 5.

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

10 (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

10 (b) Rearranging the instructions

In Fig. 9-10(a) the compiler inserts two no-op instructions after the branch.The branch address X is transferred to PC in clock cycle 7. The fetching of theinstruction at X is delayed by two clock cycles by the no-op instructions. Theinstruction at X starts the fetch phase at clock cycle 8 after the program counterPC has been updated.

The program in Fig. 9-10(b) is rearranged by placing theadd and subtractinstructions after the branch instruction instead of before as in the originalprogram. Inspection of the pipeline timing shows that PC is updated to thevalue of X in clock cycle 5, but the add and subtract instructions are fetchedfrom memory and executed in the proper sequence. In other words, if the loadinstruction is at address 101 and X is equal to 350, the branch instruction isfetched from address 103. The add instruction is fetched from address 104 andexecuted in clock cycle 6. The subtract instruction is fetched from address 105and executed in clock cycle 7. Since the value of X is transferred to PC with clockcycle 5 in the E segment, the instruction fetched from memory at clock cycle6 is from address 350, which is the instruction at the branch address.

5.6 Vector Processing

- In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.
- Computers with vector processing capabilities are in demand in specialized applications. e.g.

- Long-range weather forecasting
- Petroleum explorations
- Seismic data analysis
- Medical diagnosis
- Artificial intelligence and expert systems
- Image processing
- Mapping the human genome

- To achieve the required level of high performance it is necessary to utilize the *fastest and most reliable hardware* and apply innovative procedures from *vector and parallel processing techniques*.

Vector Operations

- Many scientific problems require arithmetic operations on large arrays of numbers.
- A vector is an ordered set of a one-dimensional array of data items.
- A vector V of length n is represented as a row vector by V=[v1,v2,…,Vn].
- To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

```
   DO 20 I = 1, 100
20      C(I) = B(I) + A(I)
```

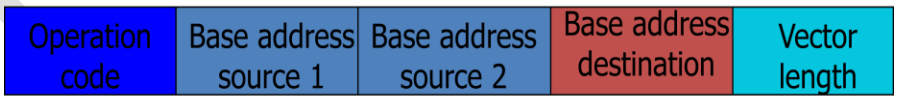
- This is implemented in machine language by the following sequence of operations.

```
Initialize I=0
20 Read A(I)
   Read B(I)
   Store C(I) = A(I)+B(I)
   Increment I = I + 1
   If I  100 go to 20
   Continue
```

- A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop.

```
C(1:100) = A(1:100) + B(1:100)
```

- A possible instruction format for a vector instruction is shown in Fig. 9-11.
  - This assumes that the vector operands reside in *memory*.



- It is also possible to design the processor with a large number of *registers* and store all operands in registers prior to the addition operation.
  - The base address and length in the vector instruction specify a group of CPU registers.

Matrix Multiplication

- The multiplication of two n x n matrices consists of n^2 inner products or n^3 multiply-add operations.
- Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

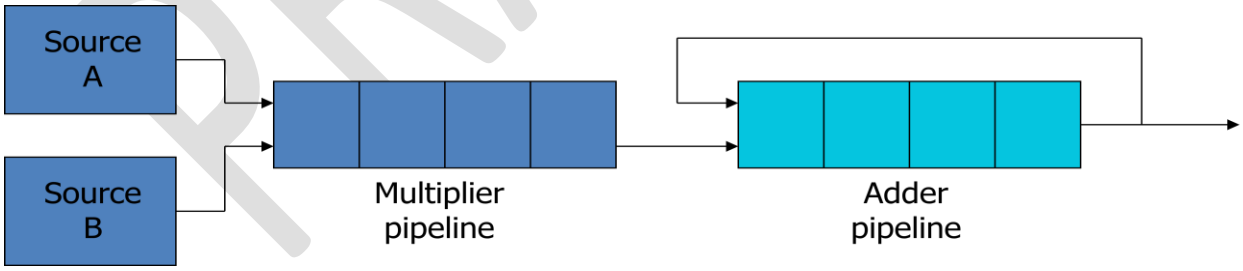
The product matrix C is a 3×3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting $i = 1, j = 1$, to obtain

  - $c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$
  - This requires three multiplication and (after initializing c_{11} to 0) three additions.
- In general, the inner product consists of the sum of *k* product terms of the form $C = A_1B_1+A_2B_2+A_3B_3+\ldots+A_kB_k$.
  - In a typical application k may be equal to 100 or even 1000.
- The inner product calculation on a pipeline vector processor is shown in Fig. 9-12.

$$\begin{aligned} C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots \\ & + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots \\ & + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots \\ & + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots \end{aligned}$$

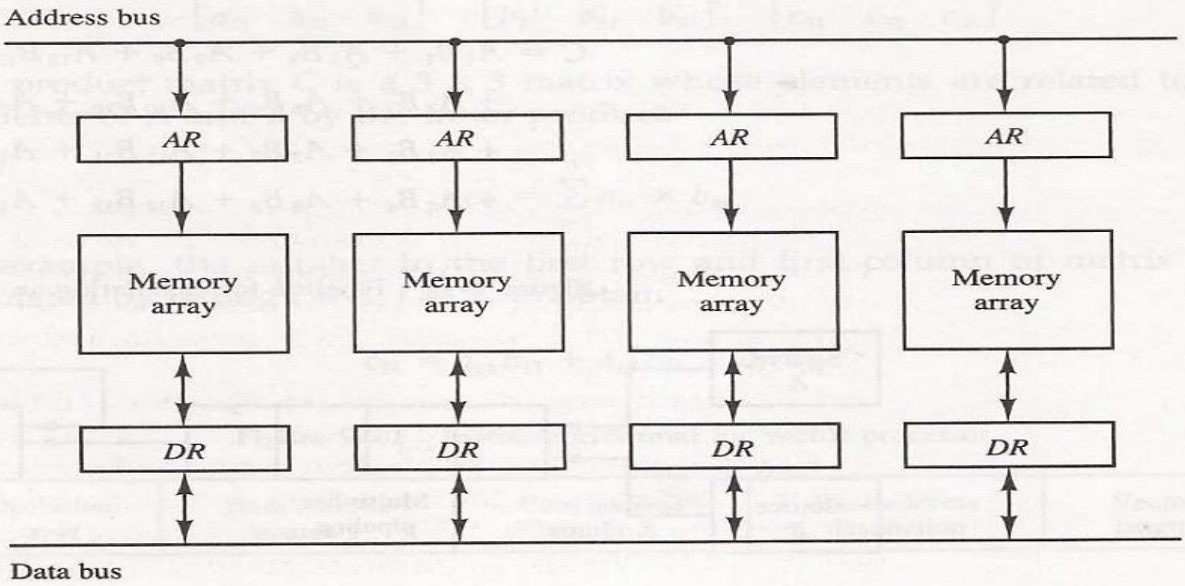


Memory Interleaving

- *Pipeline* and *vector processors* often require simultaneous access to memory from two or more sources.
  - An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.
  - An arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time.

- Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses.
    - A memory module is a memory array together with its own address and data registers.
- Fig. 9-13 shows a memory unit with four modules.

Figure 9-13 Multiple module memory organization.



- The advantage of a modular memory is that it allows the use of a technique called *interleaving*.
- In an interleaved memory, different sets of addresses are assigned to different memory modules.
- By staggering the memory access, the effective memory cycle time can be *reduced by a factor close to the number of modules*.

Supercomputers

- A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a *supercomputer*.
    - To speed up the operation, the components are *packed tightly* together to minimize the distance that the electronic signals have to travel.
- This is augmented by instructions that process vectors and combinations of scalars and vectors.
- A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing.
    - It is equipped with *multiple functional units* and each unit has its own *pipeline* configuration.
- It is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.
- They are limited in their use to a number of scientific applications, such *as numerical weather forecasting, seismic wave analysis, and space research*.
- A measure used to evaluate computers in their ability to perform a given number of floating-point operations per second is referred to as *flops*.
- A typical supercomputer has a basic cycle time of 4 to 20 ns.
- The examples of supercomputer:

- Cray-1: it uses vector processing with 12 distinct functional units in parallel; a large number of registers (over 150); multiprocessor configuration (Cray X-MP and Cray Y-MP)
- Fujitsu VP-200: 83 vector instructions and 195 scalar instructions; 300 megaflops

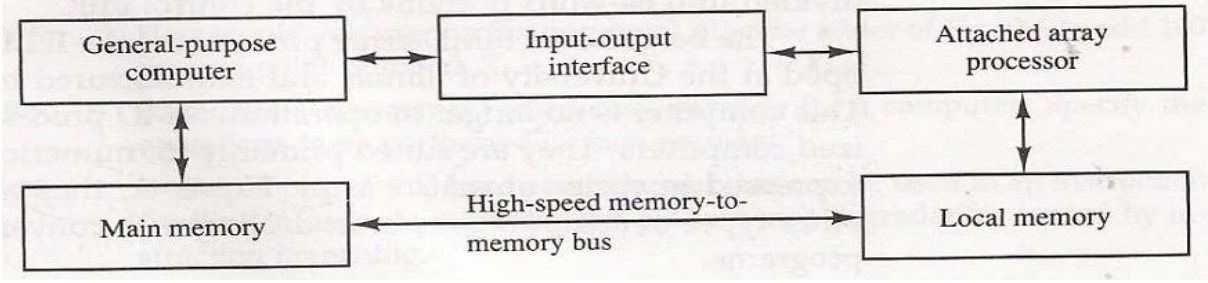
5.7 Array Processors : Introduction

- An array processor is a processor that performs computations on large arrays of data.
- The term is used to refer to two different types of processors.
  - Attached array processor:
    - Is an auxiliary processor.
    - It is intended to improve the performance of the host computer in specific numerical computation tasks.
  - SIMD array processor:
    - Has a single-instruction multiple-data organization.
    - It manipulates vector instructions by means of multiple functional units responding to a common instruction.

Attached Array Processor

- Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
  - Parallel processing with multiple functional units
- Fig. 9-14 shows the 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 general-purpose computer without the attached processor serves the users that need conventional data processing. The system with the attached processor satisfies the needs for complex arithmetic applications.

Figure 9-14 Attached array processor with host computer.



- 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 100megaflops.
- The objective of the attached array processor is to provide *vector manipulation capabilities* to a conventional computer at a fraction of the cost of supercomputer.

SIMD Array Processor

- An SIMD array processor is a computer with multiple processing units operating in parallel.
- A general block diagram of an array processor is shown in Fig. 9-15.
    - It contains a set of identical processing elements (PEs), each having a local memory *M*.
    - Each PE includes an ALU, a floating-point arithmetic unit, and working registers.
    - Vector instructions are broadcast to all PEs simultaneously.
- Masking schemes are used to control the status of each PE during the execution of vector instructions.
    - Each PE has a flag that is set when the PE is active and reset when the PE is inactive.

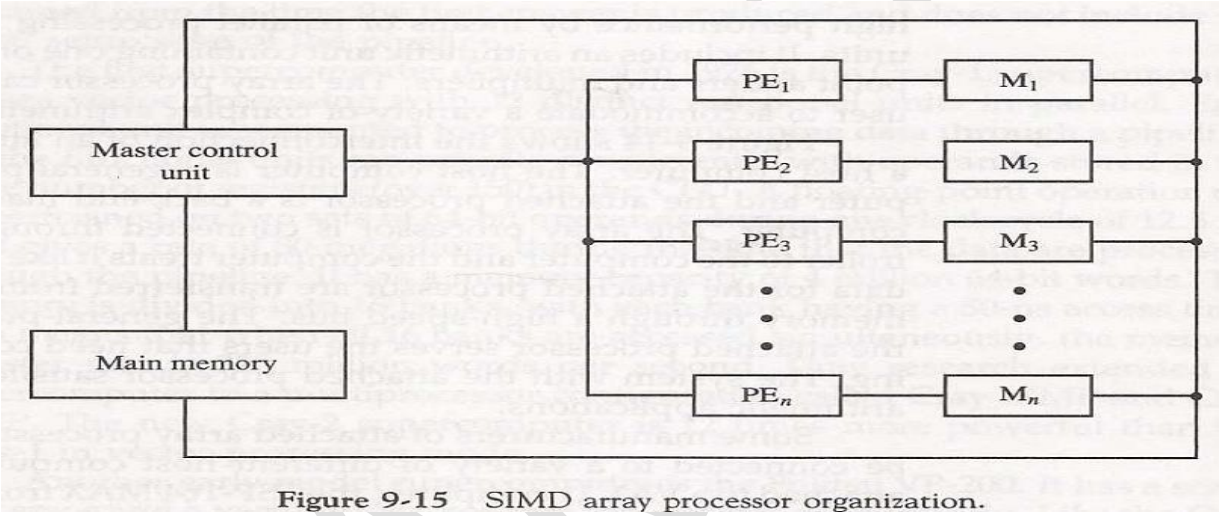


Figure 9-15 SIMD array processor organization.

- For example, the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp.
    - Are highly specialized computers.
    - They are suited primarily for numerical problems that can be expressed in vector or matrix form.

MULTIPROCESSORS

Characteristics of Multiprocessors, Interconnection Structures, Interprocessor Arbitration, Interprocessor Communication and Synchronization, Cache Coherence, Shared Memory Multiprocessors.

5.8 Characteristics of Multiprocessors

A multiprocessor system 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 (IOP). However, a system with a single CPU and one or more IOPs is usually not included in the definition of a multiprocessor system unless the IOP has computational facilities comparable to a CPU. As it is most commonly defined, a multiprocessor system implies the existence of multiple CPUs, although usually there will be one or more IOPs as well. Multiprocessors are classified as multiple instruction stream, multiple data MIMD stream (MIMD) systems.

There are some similarities between multiprocessor and multicomputer systems since both support concurrent operations. However, there exists an important distinction between a system with multiple computers and a system with multiple processors. 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.

Although some large-scale computers include two or more CPUs in their microprocessor overall system, it is the emergence of the microprocessor that has been the major motivation for multiprocessor systems.

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 with perhaps some loss in efficiency.

The benefit derived from a multiprocessor organization is an 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.

Multiprocessing can improve performance by decomposing a program into parallel executable tasks. This can be achieved in one of two ways. The user can explicitly declare that certain tasks of the program be executed in parallel.

The other, 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. If a program depends on data generated in another part, the part yielding the needed data must be executed first. However, two parts of a program that do not use data generated by each can run concurrently.

Multiprocessors are classified by the way their memory is organized. A multiprocessor system with common shared memory is classified as a **shared- tightly coupled memory** or **tightly coupled multiprocessor**. This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a

cache memory with each CPU. 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.

An alternative model of microprocessor is the **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. Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.

5.9 Interconnection Structures

The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

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

Time-Shared Common Bus

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig. 13-1.

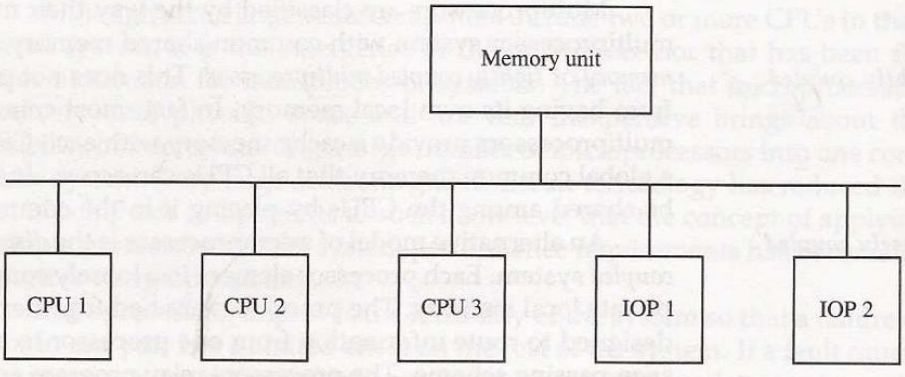


Figure 13-1 Time-shared common bus organization.

Only one processor can communicate with the memory or another processor at any given time. Transfer operations are conducted by the processor that is in control of the bus at the 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.

A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. 13-2. 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 IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, 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 IOP is connected directly to the system bus, the I/O devices attached to it may be made available to all processors. Only one processor can communicate shared memory 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 I/O devices. Part of the local memory may be designed as a cache memory attached to the CPU. In this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

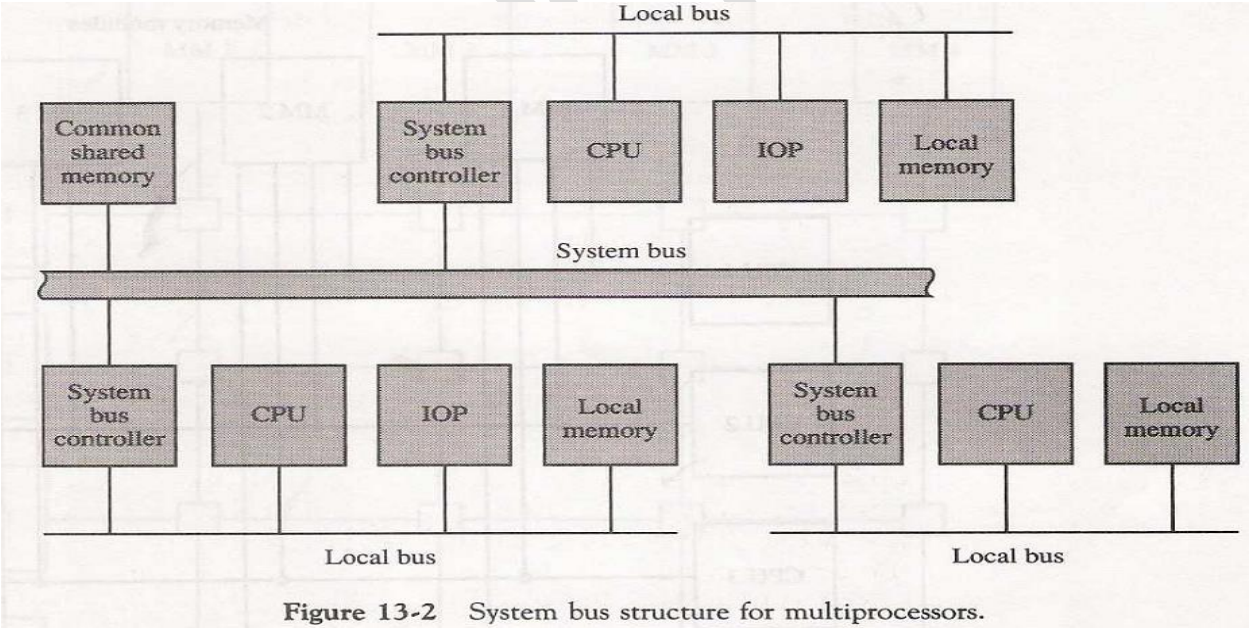


Figure 13-2 System bus structure for multiprocessors.

Multiport Memory

A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 13-3 for four CPUs and four memory modules (MMs). 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.

The advantage of the multiport 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. As a consequence, this interconnection structure is usually appropriate for systems with a small number of processors.

Crossbar Switch

The crossbar switch organization consists of a number of crosspoints that are placed at intersections between processor buses and memory module paths. Figure 13-4 shows a crossbar switch interconnection between four CPUs and four memory modules. 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.

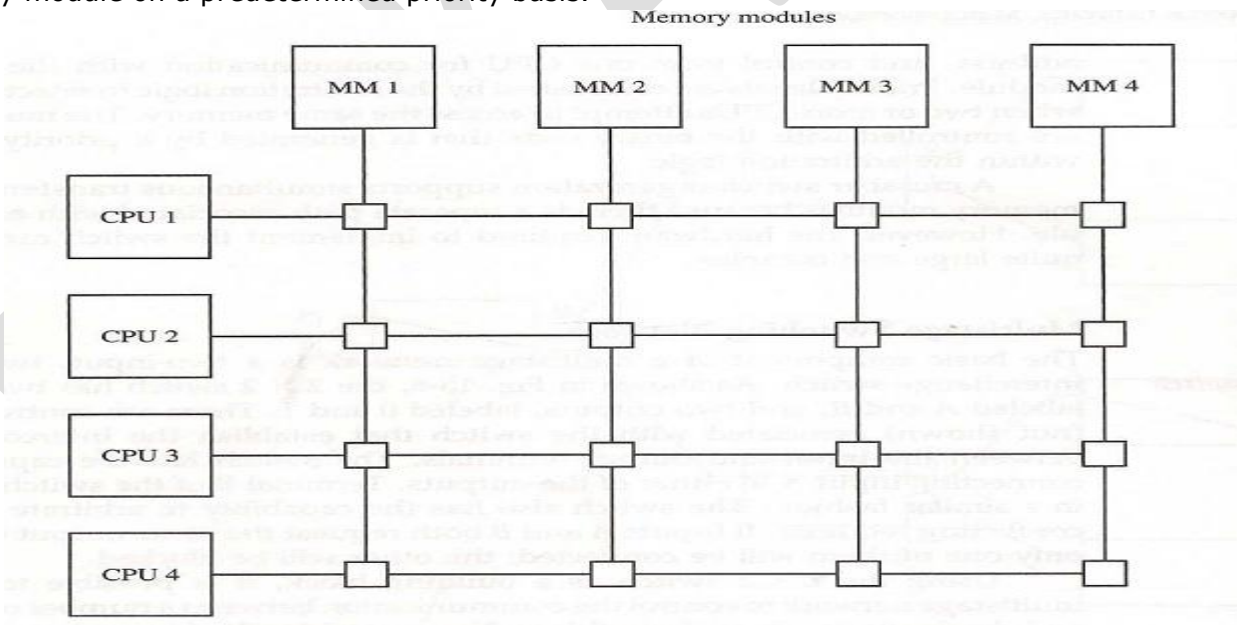
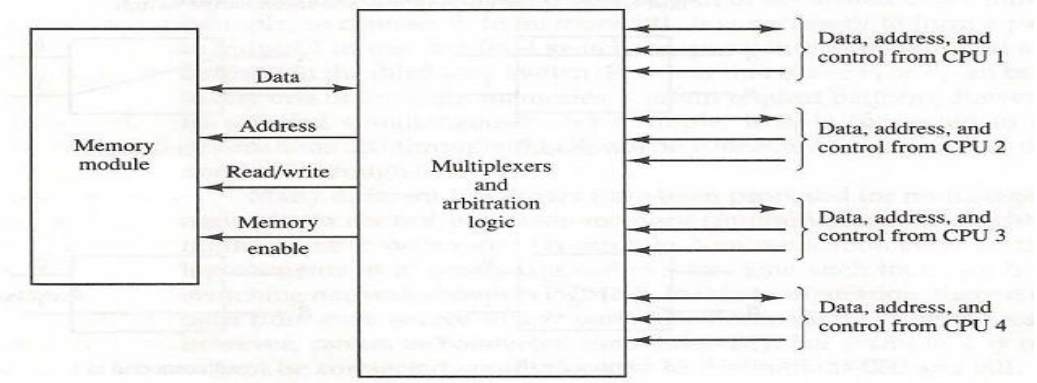


Figure 13-4 Crossbar switch.

Figure 13-5 shows the functional design of a crossbar switch connected to one memory module. 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.

Figure 13-5 Block diagram of crossbar switch.

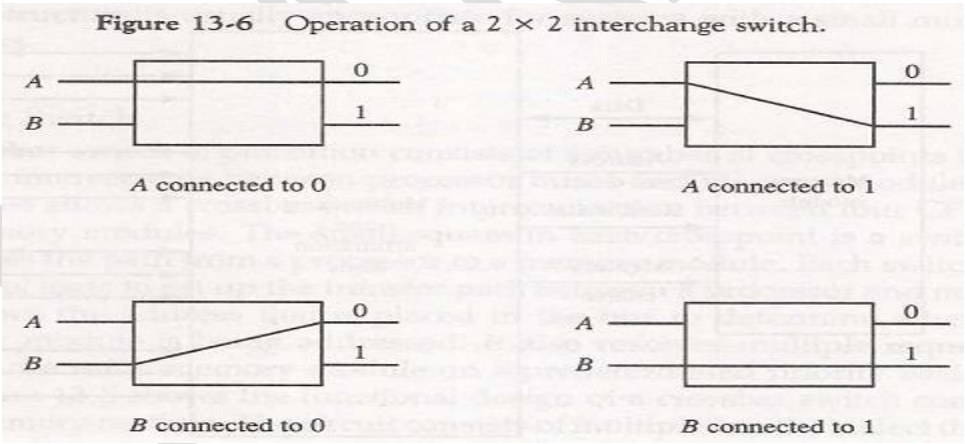


A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module. However, the hardware required to implement the switch can become quite large and complex.

Multistage Switching Network

The basic component of a multistage network is a two-input, two-output interchange switch interchange switch. As shown in Fig. 13-6, the 2 X 2 switch has two input labeled A and B, and two outputs, labeled 0 and 1. There are control signals (not shown) associated with the switch that establish the interconnection between the input and output terminals. The switch has the capability connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs A and B both request the same output terminal only one of them will be connected; the other will be blocked.

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



Using the 2 X 2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations. To see how this is done, consider the binary tree shown Fig. 13-7. The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination number. The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and the third bit specifies the output of the switch in the third level. For example, to connect P1 to memory 101, it is necessary to form a path from P1 to output 1 in the first-level switch,

output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either P1 or P; can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if P1 is connected to one of the destinations 000 through 011, P2 can be connected to only one of the destinations 100 through 111.

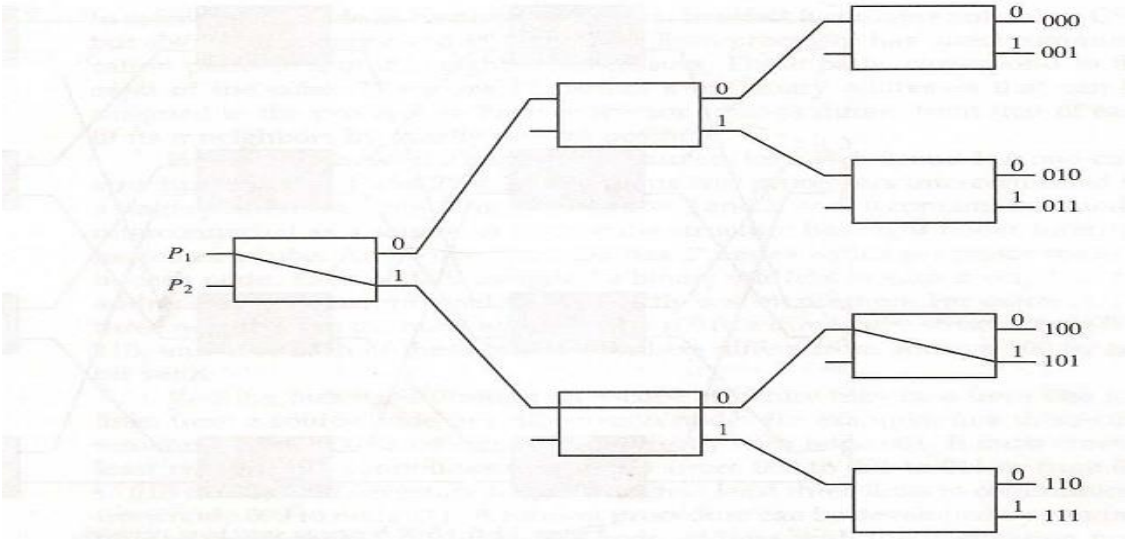


Figure 13-7 Binary tree with 2 × 2 switches.

Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the **omega network switching network** shown in Fig. 13-8. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

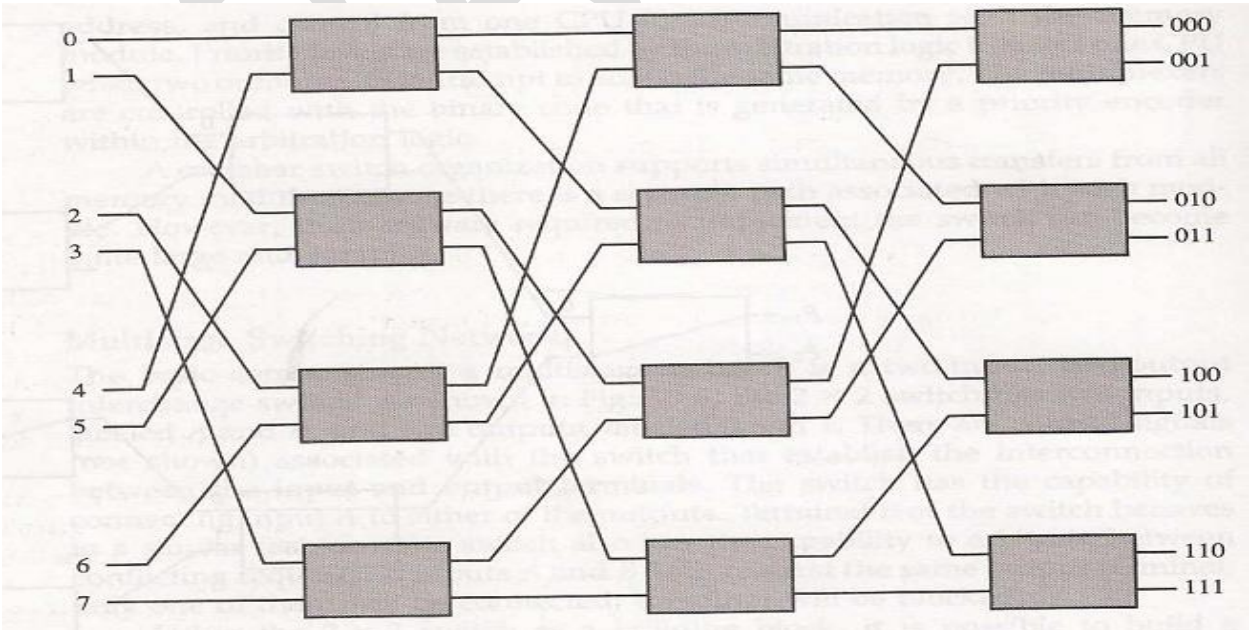


Figure 13-8 8 × 8 omega switching network.

A particular request is initiated in the switching network by the source, which sends a 3-bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the 2 X 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit. When the request arrives on either input of the 2 x 2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module. The first pass through the network sets up the path. Succeeding passes are used to transfer the address into memory and then transfer the data in either direction, depending on whether the request is a read or a write. In a loosely coupled multiprocessor system, both the source and destination are processing elements. After the path is established, the source processor transfers a message to the destination processor.

Hypercube Interconnection

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N = 2^n$ processors interconnected in an n-dimensional binary cube. Each processor forms a node of the cube. Although it is customary to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication paths to n other neighbor processors. These paths correspond to the edges of the cube. There are 2^n distinct n-bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.

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

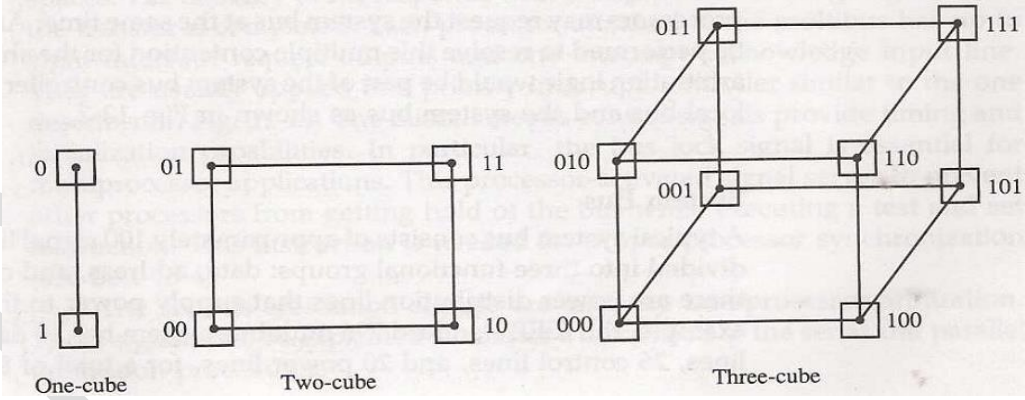


Figure 13-9 shows the hypercube structure for n = 1, 2, and 3. A one-cube structure has n = 1 and $2^n = 2$. It contains two processors interconnected by a single path. A two-cube structure has n = 2 and $2^n = 4$. It contains four nodes interconnected as a square. A three-cube structure has eight nodes interconnected as a cube. An n-cube structure has 2^n nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three-cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value. Routing messages through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three-cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with

011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111. A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three-cube structure, a message at 010 going to 001 produces an exclusive-OR of the two addresses equal to 011. The message can be sent along the second axis to 000 and then through the third axis to 001.

5.10 Interprocessor Arbitration

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU. A memory bus consists of lines for transferring data, address, and read/write information. An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multisystem bus processor system, such as CPUs, IOPs, and memory, is called a system bus.

The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must then be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus as shown in Fig. 13-2.

System Bus

A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control. In addition, there are power distribution lines that supply power to the components.

The data lines provide a path for the transfer of data between processors and common memory. The number of data lines is usually a multiple of 8, with 16 and 32 being most common. The address lines are used to identify a memory address or any other source or destination, such as input or output ports. The number of address lines determines the maximum possible memory capacity in the system. For example, an address of 24 lines can access up to 224 (16 mega) words of memory. The data and address lines are terminated with three-state buffers. The address buffers are unidirectional from processor to memory. The data lines are bidirectional, allowing the transfer of data in either direction.

Data transfers over the system bus may be synchronous or asynchronous. In a synchronous bus, each data item is transferred during a time slice known in advance to both source and destination units. Synchronization is achieved by driving both units from a common clock source. An alternative procedure is to have separate clocks of approximately the same frequency in each unit. Synchronization signals are transmitted periodically in order to keep all clocks in the system in step with each other.

In an asynchronous bus, each data item being transferred is accompanied by handshaking control signals to indicate when the data are transferred from the source and received by the destination.

The control lines provide signals for controlling the information transfer between units. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include transfer signals such as memory read and write, acknowledge of a transfer, interrupt requests, bus control signals such as bus request and bus grant, and signals for arbitration procedures.

Table 13-1 lists the 86 lines that are available in the IEEE standard 796 multibus. It includes 16 data lines and 24 address lines.

TABLE 13-1 IEEE Standard 796 Multibus Signals

	Signal name
Data and address	
Data lines (16 lines)	DATA0–DATA15
Address lines (24 lines)	ADRS0–ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK
Interrupt control	
Interrupt request (8 lines)	INT0–INT7
Interrupt acknowledge	INTA
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1–INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

Reprinted with permission of the IEEE.

The six bus arbitration signals are used for interprocessor arbitration. These signals will be explained later after a discussion of the serial and parallel arbitration procedures.

Serial Arbitration Procedure

Arbitration procedures service all processor requests on the basis of established priorities. A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the units requesting control of the system bus. The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits. The processors connected to the system bus are assigned priority according to their position along the priority control line. The device closest to the priority line is assigned the highest priority. When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.

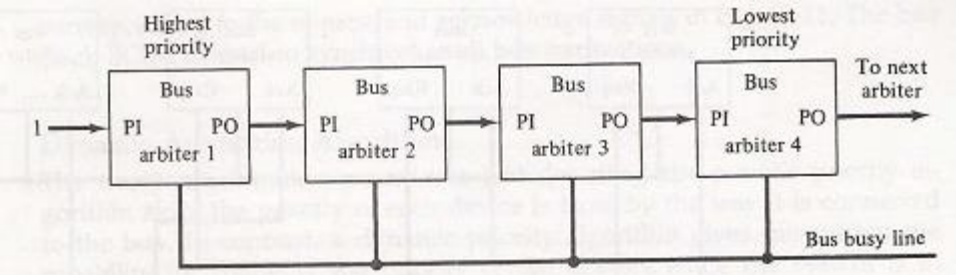


Figure 13-10 Serial (daisy-chain) arbitration.

Figure 13-10 shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (Po) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter. The PI of the highest-priority unit is maintained at logic 1 value. The highest-priority unit in the system will always receive access to the system bus when it requests it. The Po output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next unit in the chain. It the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1, it sets its PO output to 0. Lower-priority arbiters receive a 0 in PI and generate a 0 in Po. Thus the processor whose arbiter has a PI = 1 and Po = 0 is the one that is given control of the system bus.

A processor may be in the middle of a bus operation when a higher-priority processor requests the bus. The lower-priority processor must complete its bus operation before it relinquishes control of the bus. The bus busy line shown in Fig. 13-10 provides a mechanism for an orderly transfer of control. The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection. When an arbiter receives control of the bus (because its PI = 1 and Po = 0) it examines the busy line. If the line is inactive, it means that no other processor is using the bus. The arbiter activates the busy line and its processor takes control of the bus. However, if the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation. When the bus busy line returns to its inactive state, the higher-priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

Parallel Arbitration Logic

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in Fig. 13-11. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. Each arbiter enables the request line when its processor is requesting access to the system bus. The processor takes control of the bus if its acknowledge input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.

Figure 13-11 shows the request lines from four arbiters going into a 4 X 2 priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus. The 2-bit code from the encoder output drives a 2 X 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

Dynamic Arbitration Algorithms

The two bus arbitration procedures just described use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus. In contrast, a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation. We now discuss a few arbitration procedures that use dynamic priority algorithms.

Time slice

The time slice algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.

Polling

In a bus system that uses polling, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.

LRU

The least recently used (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to the LRU algorithm. With this procedure, no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

FIFO

In the first-come, first-serve scheme, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on a first-in, first-out (FIFO) basis.

Rotating Daisy-Chain

The rotating daisy-chain procedure is a dynamic extension of the daisy-chain algorithm. In this scheme there is no central bus controller, and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop. This is similar to the connections shown in Fig. 13-10 except that the PO output of arbiter 4 is connected to the PI input of arbiter 1. Whichever device has access to the bus serves as a bus controller for the following arbitration. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

Differences between TCS & LCS

- Tightly Coupled Systems are a group of machines that are largely dependent on each other. They are often used when latency is an important factor in the application. For example, a web cluster is often a tightly coupled cluster as the web

servers/application servers require quick and consistent access to a shared storage system (network file system or database).

- Loosely Coupled Systems are a group of machines (or groups of groups) which can operate independent of each other. Communications between nodes (or sub clusters) is often done via a queuing system.

5.10 Interprocessor Communication and synchronization

The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input-output channels. In a shared memory multiprocessor system, the most common procedure is to set aside a portion of memory that is accessible to all processors. The primary use of the common memory is to act as a message center similar to a mailbox, where each processor can leave messages for other processors and pick up messages intended for it.

The sending processor structures a request, a message, or a procedure, and places it in the memory mailbox. Status bits residing in common memory are generally used to indicate the condition of the mailbox, whether it has meaningful information, and for which processor it is intended. The receiving processor can check the mailbox periodically to determine if there are valid messages for it. The response time of this procedure can be time consuming since a processor will recognize a request only when polling messages. A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal. This can be accomplished through a software-initiated interprocessor interrupt by means of an instruction in the program of one processor which when executed produces an external interrupt condition in a second processor. This alerts the interrupted processor of the fact that a new message was inserted by the interrupting processor.

In addition to shared memory, a multiprocessor system may have other shared resources. For example, a magnetic disk storage unit connected to an IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk. A communication path between two CPUs can be established through a link between two IOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an I/O device so that messages can be transferred through the I/O path.

To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. This task is given to the operating system. There are three organizations that have been used in the design of operating system for multiprocessors: master-slave configuration, separate operating system, and distributed operating system.

In a master-slave mode, one processor, designated the master, always executes the operating system functions. The remaining processors, denoted as slaves, do not perform operating system functions. If a slave processor needs an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.

In the separate operating system organization, each processor can execute the operating system routines it needs. This organization is more suitable for loosely coupled systems where every processor may have its own copy of the entire operating system.

In the distributed operating system organization, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time. This type of organization is also referred to as a floating operating system since the routines float from one processor to another and the execution of the routines may be assigned to different processors at different times.

In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communication between processors is by means of message passing through 1/0 channels. The communication is initiated by one processor calling a procedure that resides in the memory of the processor with which it wishes to communicate. When the sending processor and receiving processor name each other as a source and destination, a channel of communication is established. A message is then sent with a header and various data objects used to communicate between nodes. There may be a number of possible paths available to send the message between any two nodes. The operating system in each node contains routing information indicating the alternative paths that can be used to send a message to other nodes. The communication efficiency of the interprocessor network depends on the communication routing protocol, processor speed, data link speed, and the topology of the network.

Interprocessor Synchronization

The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes. Communication refers to the exchange of data between different processes. For example, parameters passed to a procedure in a different processor constitute interprocessor communication. Synchronization refers to the special case where the data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.

Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources. Low-level primitives are implemented directly by the hardware. A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is through the use of a binary semaphore.

Mutual Exclusion with a Semaphore

A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources. This is necessary to protect data from being changed simultaneously by two or more processors. This mechanism has been termed mutual exclusion. Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when critical section it is in a critical section. A critical section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource.

A binary variable called a semaphore is often used to indicate whether or not a processor is executing a critical section. A semaphore is a software- controlled flag that is stored in a memory location that all processors can access. When the semaphore is equal to 1, it means that a processor is executing a critical program, so that the shared memory is

not available to other processors. When the semaphore is equal to 0, the shared memory is available to any requesting processor. Processors that share the same memory segment agree by convention not to use the memory segment unless the semaphore is equal to 0, indicating that memory is available. They also agree to set the semaphore to 1 when they are executing a critical section and to clear it to 0 when they are finished.

Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation. If it is not, two or more processors may test the semaphore simultaneously and then each set it, allowing them to enter a critical section at the same time. This action would allow simultaneous execution of critical section, which can result in erroneous initialization of control parameters and a loss of essential information.

A semaphore can be initialized by means of a test and set instruction in hardware lock conjunction with a hardware lock mechanism. A hardware lock is a processor-generated signal that serves to prevent other processors from using the system bus as long as the signal is active. The test-and-set instruction tests and sets a semaphore and activates the lock mechanism during the time that the instruction is being executed. This prevents other processors from changing the semaphore between the time that the processor is testing it and the time that it is setting it. Assume that the semaphore is a bit in the least significant position of a memory word whose address is symbolized by SEM. Let the mnemonic TSL designate the "test and set while locked" operation. The instruction

TSL SEM

will be executed in two memory cycles (the first to read and the second to write) without interference as follows:

R ← M[SEM] Test semaphore

M[SEM] ← 1 Set semaphore

The semaphore is tested by transferring its value to a processor register R and then it is set to 1. The value in R determines what to do next. If the processor finds that R = 1, it knows that the semaphore was originally set. (The fact that it is set again does not change the semaphore value.) That means that another processor is executing a critical section, so the processor that checked the semaphore does not access the shared memory. If R = 0, it means that the common memory (or the shared resource that the semaphore represents) is available. The semaphore is set to 1 to prevent other processors from accessing memory. The processor can now execute the critical section. The last instruction in the program must clear location SEM to zero to release the shared resource to other processors.

5. Cache Coherence

The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory. In the *write-through* policy, both cache and main memory are updated with every write operation. In the *write-back* policy, only the cache is updated and the location is marked so that it can be copied later into main memory.

In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory. To ensure the ability of the system to execute

memory operations correctly, the multiple copies must be kept identical. This requirement imposes a cache coherence problem. A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Without a proper solution to the cache coherence problem, caching cannot be used in bus-oriented multiprocessors with two or more processors.

Conditions for Incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data. Read-only data can safely be replicated without cache coherence enforcement mechanisms. To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12. Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3. As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory.

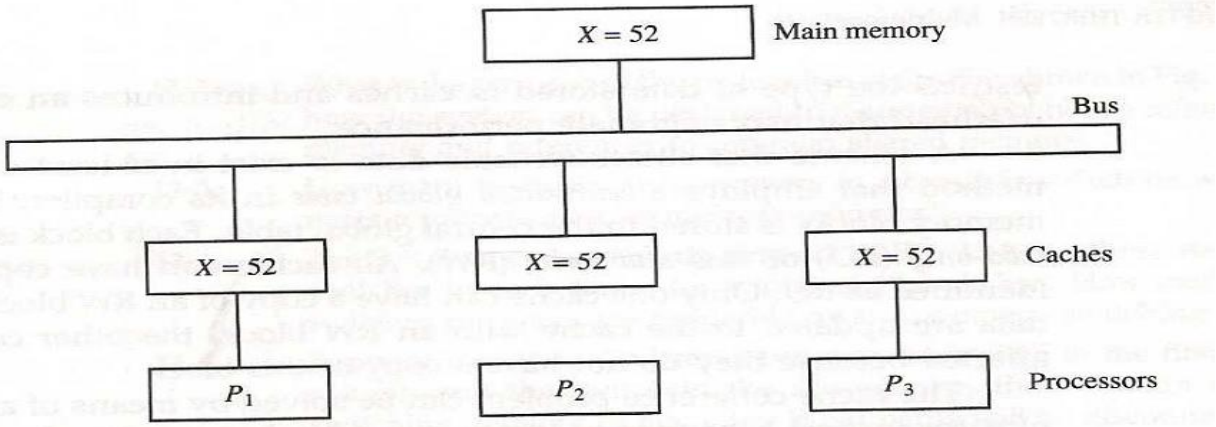
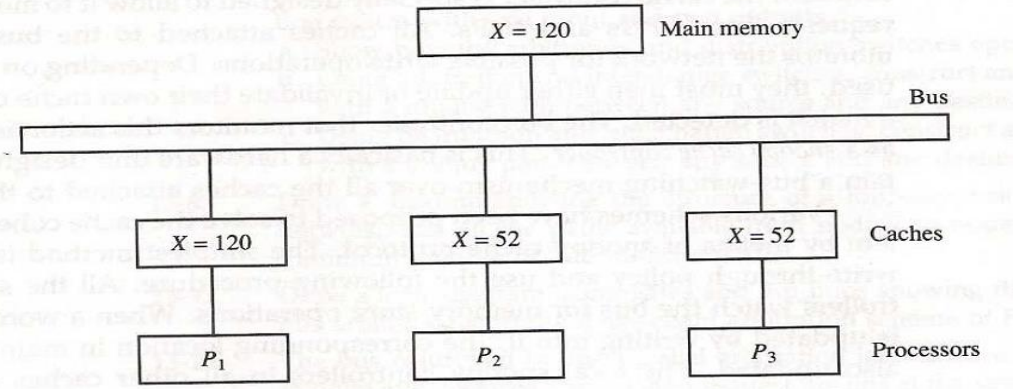


Figure 13-12 Cache configuration after a load on X.

If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache. This is shown in Fig. 13-13. A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy.

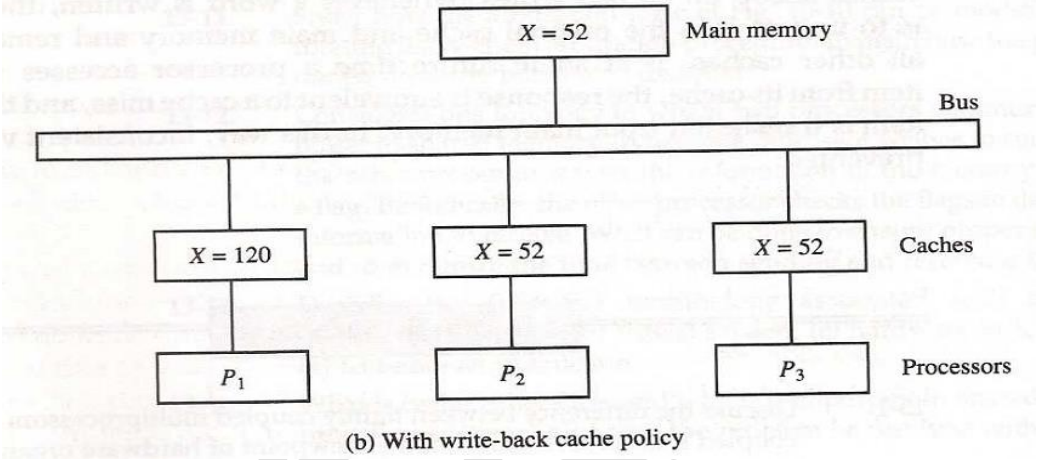
Figure 13-13 Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy

A *write-through policy* maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value. In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy. I/O-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.



(b) With write-back cache policy

Solutions to the Cache Coherence Problem

Various schemes have been proposed to solve the cache coherence problem in shared memory multiprocessors. We discuss some of these schemes briefly here.

A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. This method violates the principle of closeness of CPU to cache and increases the average memory access time. In effect, this scheme solves the problem by avoiding it.

For performance considerations it is desirable to attach a private cache to each processor. One scheme that has been used allows only nonshared and read-only data to be stored in caches. Such items are called cachable. Shared writable data are noncachable. The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches. The noncachable data remain in main memory. This method restricts the type of data stored in caches and introduces an extra software overhead that may degradate performance.

A scheme that allows writable data to exist in at least one cache is a method that employs a centralized global table in its compiler. The status of memory blocks is stored in the central global table. Each block is identified as read-only (RO) or read and write (RW). All caches can have copies of blocks identified as RO. Only one cache can have a copy of an RW block. Thus if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes. The two methods mentioned previously use software-based procedures that require the ability to tag information in order to disable caching of shared writable data. Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency. In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor the network for possible write operations. Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected. The bus controller that monitors this action is referred to snoopy cache as a snoopy cache controller. This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

Various schemes have been proposed to solve the cache coherence problem by means of snoopy cache protocol. The simplest method is to adopt a write-through policy and use the following procedure. All the snoopy controllers watch the bus for memory store operations. When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated. The local snoopy controllers in all other caches check their memory to determine if they have a copy of the word that has been overwritten. If a copy exists in a remote cache, that location is marked invalid. Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches. If at some future time a processor accesses the invalid item from its cache, the response is equivalent to a cache miss, and the updated item is transferred from main memory. In this way, inconsistent versions are prevented.