DIGITAL NOTES
ON
COMPUTER ORGANIZATION AND ARCHITECTURE

B.TECH II YEAR - I SEM
(2019-20)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

(R18A1201) COMPUTER ORGANIZATION AND ARCHITECTURE

Objectives:
The students will be exposed:
1. To how the Computer Systems work and its basic principles
2. To Instruction Level Architecture and Instruction Execution and memory system design
3. To how the I/O devices are accessed and its principles.
4. To Instruction Level Parallelism and knowledge on micro programming
5. To the concepts of advanced pipelining techniques.

UNIT - I:
Functional blocks of a computer: CPU, memory, input-output subsystems, control unit.
Computer Organization and Architecture - Von Neumann
Data representation: signed number representation, fixed and floating point Representations, Character representation. Computer arithmetic – integer addition and Subtraction, Ripple carry adder, carry look-ahead adder, etc. Multiplication – shift-and add, Booth multiplier, Carry save multiplier, etc. Division restoring and non-restoring techniques, Floating point arithmetic.

UNIT – II:
Introduction to x86 architecture.
Instruction set architecture of a CPU: Registers, instruction execution cycle, RTL Interpretation of instructions, addressing modes, instruction set.
CPU Control unit design: Hardwired and micro-programmed design approaches

UNIT – III:
Memory system design: Semiconductor memory technologies, memory organization.
Memory organization: Memory interleaving, concept of hierarchical memory organization, Cache memory, cache size vs. block size, mapping functions, Replacement algorithms, write policies.

UNIT – IV:
Peripheral devices and their characteristics: Input-output subsystems, I/O device interface, I/O transfers – program controlled, interrupt driven and DMA, privileged and non-privileged instructions, software interrupts and exceptions. Programs and processes – role of interrupts in process state transitions, I/O device interfaces – SCII, USB

UNIT – V:
Pipelining: Basic concepts of pipelining, throughput and speedup, pipeline hazards.
Parallel Processors: Introduction to parallel processors, Concurrent access to memory and cache coherency.

TEXT BOOKS:
1. **"**Computer Organization and Design: The Hardware/Software Interface", 5th Edition by David A. Patterson and John L. Hennessy, Elsevier.
2. "Computer Organization and Embedded Systems", 6th Edition by CarlHamacher,McGraw Hill Higher   Education.

REFERENCE BOOKS:
1. "Computer Architecture and Organization", 3rd Edition by John P. Hayes,WCB/McGraw-Hill
2. "Computer Organization and Architecture: Designing for Performance", 10th Editionby William Stallings, Pearson Education.
3. "Computer System Design and Architecture", 2nd Edition by Vincent P. Heuringand Harry F. Jordan, Pearson Education.

Course Outcomes :
1. Draw the functional block diagram of a single bus architecture of a computer and describe the function of the instruction execution cycle, RTL interpretation of instructions, addressing modes, instruction set.
2. Write assembly language program for specified microprocessor for computing16 bit multiplication, division and I/O device interface (ADC, Control circuit, serial port communication).
3. Write a flowchart for Concurrent access to memory and cache coherency in Parallel Processors and describe the process.
4. Given a CPU organization and instruction, design a memory module and analyze its operation by interfacing with the CPU.
5. Given a CPU organization, assess its performance, and apply design techniques to enhance performance using pipelining, parallelism and RISC methodology

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY

INDEX

UNIT – I

Functional blocks of a computer: CPU, memory, input-output subsystems, control unit. Computer Organization and Architecture - Von Neumann

Data representation: signed number representation, fixed and floating point Representations, Character representation. Computer arithmetic – integer addition and Subtraction, Ripple carry adder, carry look-ahead adder, etc. Multiplication – shift-and add, Booth multiplier, Carry save multiplier, etc. Division restoring and non-restoring techniques, Floating point arithmetic

Functional Units

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1.
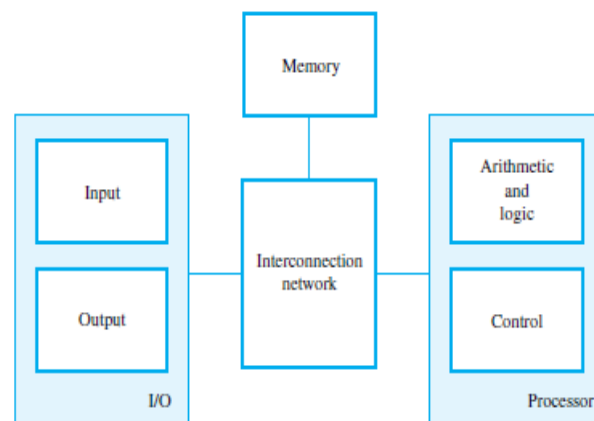

Figure 1.1   Basic functional units of a computer.

The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions. The arithmetic and logic circuits, in conjunction with the main control circuits, is the processor. Input and output equipment is often collectively referred to as the input-output (I/O) unit.

A program is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. Data are numbers and characters that are used as operands by the instructions. Data are also stored in the memory. The instructions and data handled by a computer must be encoded in a suitable format. Each instruction, number, or character is encoded as a string of binary digits called bits, each having one of two possible values, 0 or 1, represented by the two stable states.

Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.

Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays.

Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing.

Similarly, cameras can be used to capture video input.

Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually.

Instead, they are handled in groups of fixed size called words. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits.

To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. Instructions and data can be written into or read from the memory under the control of the processor. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units

Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data, located in the main memory, the data are fetched and copies are also placed in the cache. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use.

Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondarystorage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. The devices available are including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit

Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.

When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

Output unit function is to send processed results to the outside world. A familiar example of such a device is a printer. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor. Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases.

Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred.

Control circuits are responsible for generating the timing signals that govern the transfers. They determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:
• The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
• Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
• Processed information leaves the computer through an output unit.
• All activities in the computer are directed by the control unit.

Von Neumann architecture

In the 1940s, a mathematician called John Von Neumann described the basic arrangement (or architecture) of a computer. Most computers today follow the concept that he described although there are other types of architecture. A Von Neumann-based computer is a computer that:
Uses a single processor.
Uses one memory for both instructions and data. A von Neumann computer cannot distinguish between data and instructions in a memory location! It 'knows' only because of the location of a particular bit pattern in RAM.
Executes programs by doing one instruction after the next in a serial manner using a fetch-decode-execute cycle.

Number Representation and Arithmetic Operations
1.4.1 Integers
Consider an n-bit vector : B = bn−1 . . . b1b0   where bi = 0 or 1 for $0 \leq i \leq n − 1$.

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

We need to represent both positive and negative numbers. Three systems are used for representing such numbers:
• Sign-and-magnitude
• 1's-complement
• 2's-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

B   Values represented

In *1's-complement* representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for −3 is obtained by complementing each bit in the vector 0011 to yield 1100.

In the *2's-complement* system, forming the 2's-complement of an *n*-bit number is done by subtracting the number from 2*n*. Hence, the 2's-complement of a number is obtained by adding 1 to the 1's-complement of that number.

There are distinct representations for +0 and −0 in both the sign-and magnitude and 1's-complement systems, but the 2's-complement system has only one representation for 0.

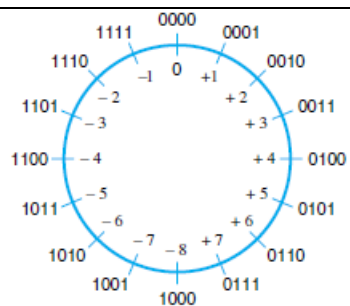**Addition of Unsigned Integers**
The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the *sum* is 0 and the *carry-out* is 1. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the *carry-in* to the next bit pair to the left.

```
    0          1          0          1
  + 0        + 0        + 1        + 1
  ---        ---        ---        ---
    0          1          1         1 0
                                     ↑
                                 Carry-out
```

Addition and Subtraction of Signed Integers
The 2's-complement system is the most efficient method for performing addition and subtraction operations.
Unsigned integers mod *N* is a circle with the values 0 through *N* − 1. The decimal values 0 through 15 are represented by their 4-bit binary values 0000 through 1111.

(b) Mod 16 system for 2's-complement numbers

The operation $(7 + 5)$ mod 16 yields the value 12. To perform this operation graphically, locate 7 (0111) on the outside of the circle and then move 5 units in the clockwise direction to arrive at the answer 12 (1100).

Similarly, $(9 + 14)$ mod 16 = 7; this is modeled on the circle by locating 9 (1001) and moving 14 units in the clockwise direction past the zero position to arrive at the answer 7 (0111).

Apply the mod 16 addition technique to the example of adding $+7$ to -3. The 2's-complement representation for these numbers is 0111 and 1101, respectively.

To *add* two numbers, add their *n*-bit representations, ignoring the carry-out bit from the most significant bit (MSB) position. The sum will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

To *subtract* two numbers X and Y, that is, to perform $X - Y$, form the 2's-complement of Y, then add it to X using the *add* rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

$$
\begin{array}{ll}
0010 & (+2) \\
+\ 0011 & (+3) \\
\hline
0101 & (+5)
\end{array}
\qquad
(b)
\begin{array}{ll}
0100 & (+4) \\
+\ 1010 & (-6) \\
\hline
1110 & (-2)
\end{array}
$$

$$
\begin{array}{ll}
1011 & (-5) \\
+\ 1110 & (-2) \\
\hline
1001 & (-7)
\end{array}
\qquad
(d)
\begin{array}{ll}
0111 & (+7) \\
+\ 1101 & (-3) \\
\hline
0100 & (+4)
\end{array}
$$

Floating-Point Numbers

If we use a full word in a 32-bit word length computer to represent a signed integer in 2's-complement representation, the range of values that can be represented is -2^{31} to $+2^{31} - 1$.

Since the position of the binary point in a floating-point number varies, it must be indicated explicitly in the representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, 3.7291×10^{-27}, -1.0341×10^{2}, -7.3000×10^{-14}. these numbers have been given to 5 *significant digits* of precision.

A binary floating-point number can be represented by:
· a sign for the number
· some significant bits
· a signed scale factor exponent for an implied base of 2
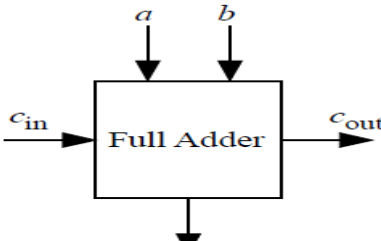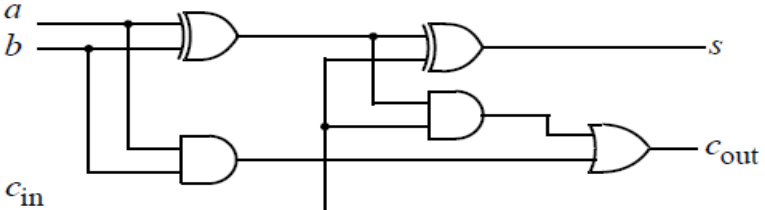
Character Representation

Bit positions 3210	Bit positions 654							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l		
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

The most common encoding scheme for characters is ASCII (American Standard Code for Information Interchange). Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes. It is convenient to use an 8-bit *byte* to represent and store a character.

The code occupies the low-order seven bits. The high-order bit is usually set to 0. This facilitates sorting operations on alphabetic and numeric data.

The low-order four bits of the ASCII codes for the decimal digits 0 to 9 are the first ten values of the binary number system.

This 4-bit encoding is referred to as the *binary-coded decimal* (BCD) code.

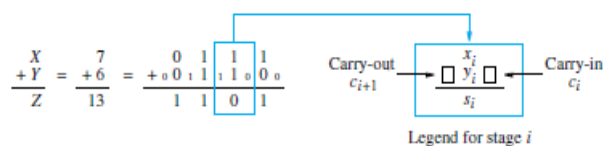|	
	A one-bit full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs(a,b, and cin) and two outputs(s, and cout)___ as illustrated in Figure 1.
Table 1: Full adder truth table. 	a	b	c_{in}	c_{out}	s
---	---	---	---	---
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1		The truth table of 1-bit full adder is given in the table
	The gate implementation of 1-bit full adder is given in the figure

Figure 9.1 Logic specification for a stage of binary addition.

Ripple carry adder

A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascaded. with the carry output from each full adder connected to the carry input of the next full adder in the chain. Figure 3 shows the interconnection of four full adder (FA) circuits to provide a 4-bit ripple carry adder. Notice from Figure 3 that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits a0 and b0 _ in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits s0 and s3

Ripple carry adder delays

In the ripple carry adder, the output is known after the carry generated by the previous stage is produced. Thus, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage. As a result, the final sum and carry bits will be valid after a considerable delay.
Table 2 shows the delays for several CMOS gates assuming all gates are equally loaded for simplicity. All delays are normalized relative to the delay of a simple inverter. The table also shows the corresponding gate areas normalized to a simple minimum-area inverter. Note from the table that multiple-input gates have to use a different circuit technique compared to simple 2-input gates.

For an n-bit ripple carry adder the sum and carry bits of the most significant bit (MSB) are obtained after a normalized delay of

$$\text{Sum } s_{n-1} \text{ delay } = 4n+2 \qquad\qquad (1)$$

$$\text{Carry } c_n \text{ delay } = 4n+3 \qquad\qquad (2)$$

For a 32-bit processor, the carry chain normalized delay would be 131. The ripple carry adder can get very slow when many bits need to be added. In fact, the carry chain propagation delay is the determining factor in most microprocessor speeds.

Carry lookahead adder (CLA)
The carry lookahead adder (CLA) solves the carry delay problem by calculating the carry signals in advance, based on the input signals. It is based on the fact that a carry signal will be generated in two cases:
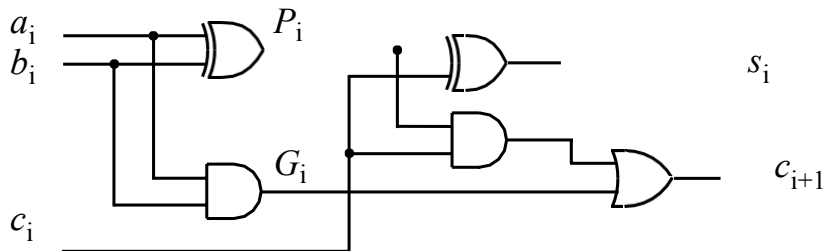(1) when both bits a_i and b_i are 1, or
(2) when one of the two bits is 1 and the carry-in is 1 .
Thus, one can write,

$$c_{i+1} = a_i.b_i + (a_i \oplus b_i).c_i \qquad (3)$$
$$s_i = (a_i \oplus b_i) \oplus c_i \qquad (4)$$

The above two equations can be written in terms of two new signals Pi and Gi , which are shown in Figure 4

Where Pi and Gi are called the carry generate and carry propagate terms, respectively. Notice that the generate and propagate terms only depend on the input bits and thus will be valid after one and two gate delay, respectively. If one uses the above expression to calculate the carry signals, one does not need to wait for the carry to ripple through all the previous stages to find its proper value. Let's apply this to a -bit adder to make it clear.
Notice that the carry-out bit, , of the last stage will be available after four delays: two gate de- lays to calculate the propagate signals and two delays as a result of the gates required to implement Equation 13.

$$c_{i+1} = G_i + P_i.c_i \qquad (5)$$
$$s_i = P_i \oplus c_i \qquad (6)$$

$$G_i = a_i.b_i \qquad (7)$$
$$P_i = a_i \oplus b_i \qquad (8)$$
$$\qquad (9)$$

Putting $i = 0, 1, 2, 3$ in Equation 5, we get

$$c_1 = G_0 + P_0.c_0 \qquad (10)$$
$$c_2 = G_1 + P_1.G_0 + P_1.P_0.c_0 \qquad (11)$$
$$c_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \qquad (12)$$
$$c_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0 \qquad (13)$$

Figure 5 shows that a 4-bit CLA is built using gates to generate the and signals and a logic block to generate the carry out signals according to Equations 10–13.
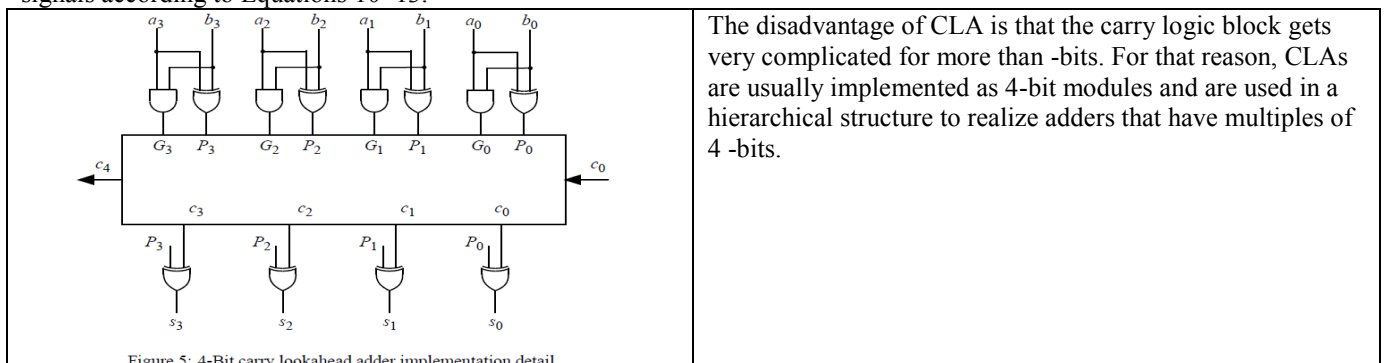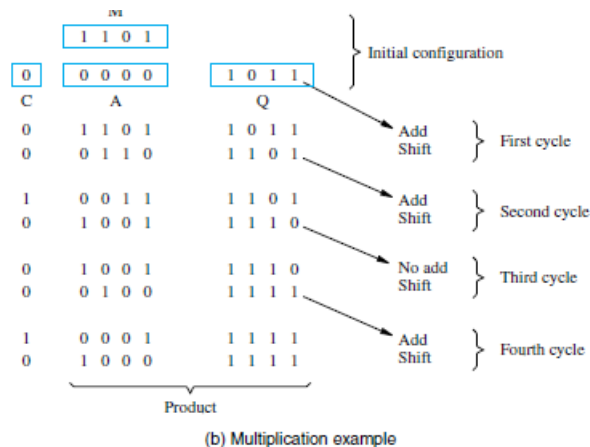

Figure 5: 4-Bit carry lookahead adder implementation detail.

The disadvantage of CLA is that the carry logic block gets very complicated for more than -bits. For that reason, CLAs are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 -bits.

Shift – Add Multiplier

Multiplication is often defined as repeated additions. Thus, to calculate 11 × 23, you would start with 0 and add 11 to it 23 times.

(b) Multiplication example

In this, the 4 bit multiplier is stored in Q register, the 4 bit multiplicand is stored in register B and the register A is initially cleared to zero. The multiplication process starts with checking of the least significant bit of B whether it is 0 or 1.

If the B0 = 1, the number in the multiplicand (B) is added with the least significant bits of the A register and all bits of C, A and Q registers are shifted to the right one bit.

If the bit B0 = 0, the combined C and Q registers are shifted to the right by one bit without performing any addition. This process is repeated for n times for n bit numbers. This method of binary multiplication is called as parallel multiplier.
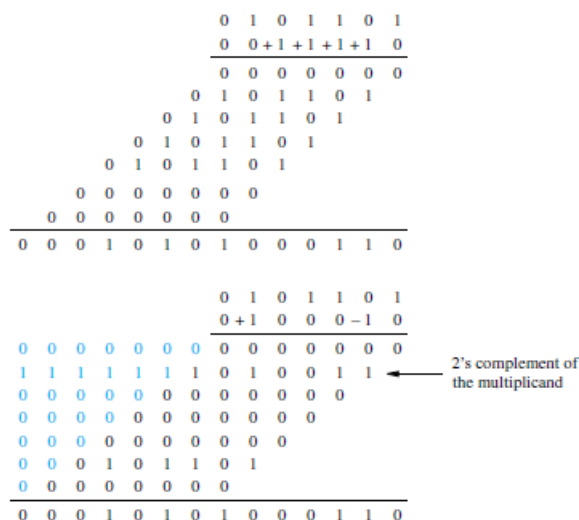
Consider the above figure in which the multiplier and multiplicand values are given as 1011 and 1101 which are loaded into the Q and A registers respectively.

Initially the register C is zero and hence the A register is zero, which stores the carry in addition.
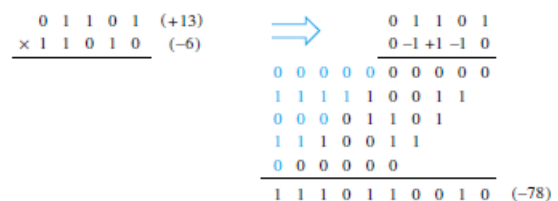
Since the B0 =1, then the number in the B is added to the bits of A and produce the addition result as 1101, and the Q and A register are shifted their values one bit right so the new values during the first cycle are 0110 and 1101 respectively.

This process has to be repeated four times to perform the 4 bit multiplication. The final multiplication result will be available in the A and Q registers as 10001111
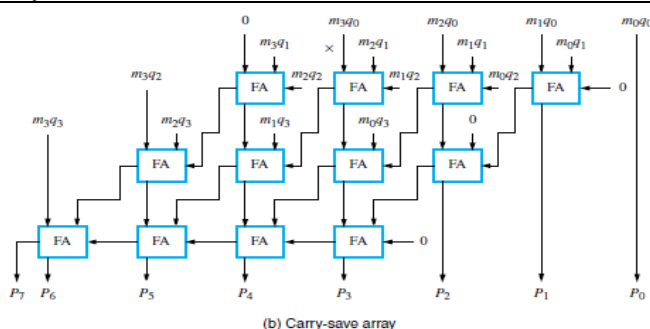
Booth Multiplier

The Booth algorithm generates a 2n-bit product and treats both positive and negative 2'scomplement n-bit operands uniformly. In general, in the Booth algorithm, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from   1 to 0, as the multiplier is scanned from right to left.

Carry-Save Addition

(b) Carry-save array

Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) can be used to speed up the process.

This structure is in the form of the array in which the first row consists of just the AND gates that produce the four inputs m_3q_0, m_2q_0, m_1q_0, and m_0q_0.

 Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.
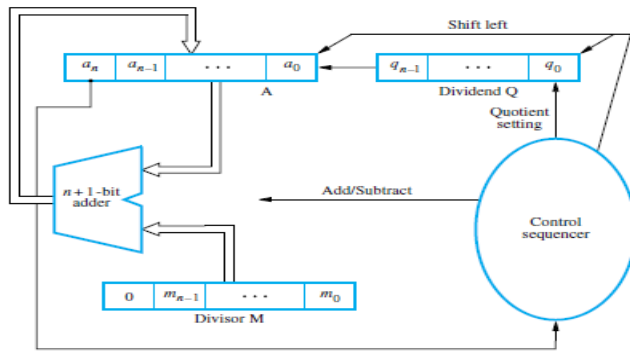
Integer Division
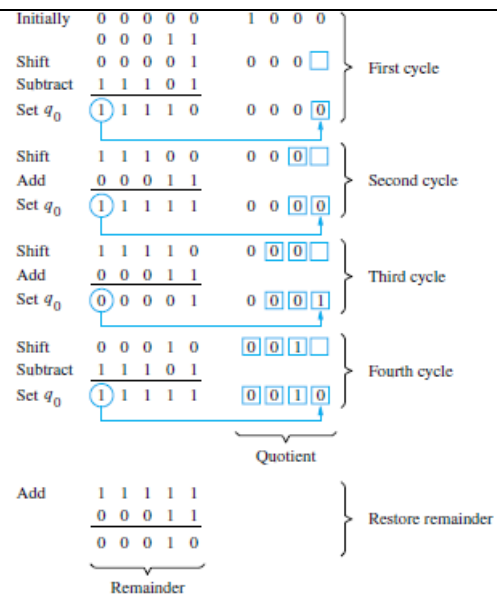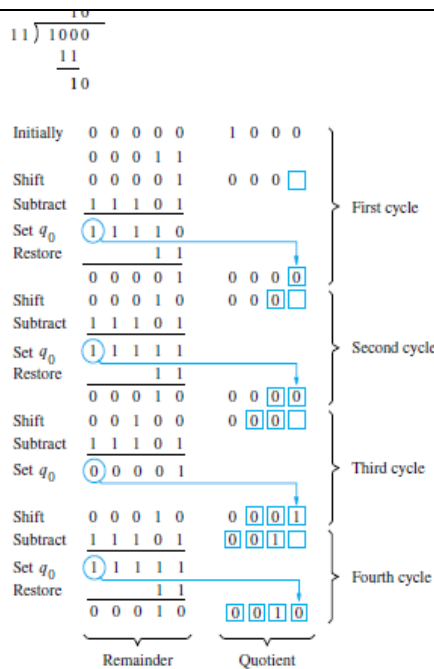

Figure 9.23 Circuit arrangement for binary division.

Restoring Division

Figure 9.25 A non-restoring division example.

Non restoring division

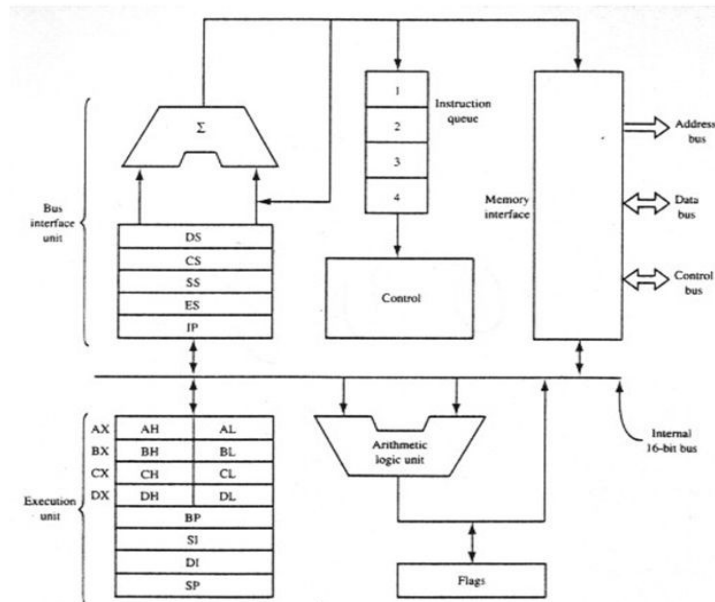| Restoring Division	Non restoring division
An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A.	If A is positive, we shift left and subtract M, that is, we perform $2A - M$. If A is negative, we restore it by performing $A + M$, and then we shift it left and subtract M.
The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.	This is equivalent to performing $2A + M$. The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed. following algorithm for *non-restoring division*.
Do the following three steps n times:	**Stage 1:** Do the following two steps n times:
1. Shift A and Q left one bit position.	1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Subtract M from A, and place the answer back in A.	2. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set q_0 to 1.	**Stage 2:** If the sign of A is 1, add M to A. Stage 2 is needed to leave the proper positive remainder in A after the n cycles of Stage 1.

Introduction to x86 architecture.
Instruction set architecture of a CPU: Registers, instruction execution cycle, RTL Interpretation of instructions, addressing modes, instruction set.
CPU Control unit design: Hardwired and micro-programmed design approaches

X86 Architecture

REGISTERS The processor provides 16 registers for use in general system and application programming. These registers can be grouped as follows:
• **General-purpose data registers**. These eight registers are available for storing operands and pointers.
• **Segment registers**. These registers hold up to six segment selectors.
 • **Status and control registers**. These registers report and allow modification of the state of the processor and of the program being executed.

General-Purpose Data Registers
The 32-bit general-purpose data registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:
 • Operands for logical and arithmetic operations
• Operands for address calculations
• Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Segment Registers
The 6 Segment Registers are:
- Stack Segment (SS). Pointer to the stack.
- Code Segment (CS). Pointer to the code.
- Data Segment (DS). Pointer to the data.

- Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').
- F Segment (FS). Pointer to more extra data ('F' comes after 'E').
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

Most applications on most modern operating systems (FreeBSD, Linux or Microsoft Windows) use a memory model that points nearly all segment registers to the same place and uses paging instead, effectively disabling their use. Typically the use of FS or GS is an exception to this rule, instead being used to point at thread-specific data.

x86 Processor Registers and Fetch-Execute Cycle

There are 8 registers that can be specified in assembly-language instructions: eax, ebx, ecx, edx, esi, edi, ebp, and esp. Register esp points to the "top" word currently in use on the stack (which grows down).

Register ebp is typically used as a pointer to a location in the stack frame of the currently executing function.

Register ecx can be used in binary arithmetic operations to hold the second operand.

There are two registers that are used implicitly in x86 programs and cannot be referenced by name in an assembly language program.

These are eip, the "instruction pointer" or "program counter"; and eflags, which contains bits indicating the result of arithmetic and compare instructions.

The basic operation of the processor is to repeatedly fetch and execute instructions.

```
while (running) {
    fetch instruction beginning at address in eip;
    eip <- eip + length of instruction;
    execute fetched instruction;
}
```

Execution continues sequentially unless execution of an instruction causes a jump, which is done by storing the target address in eip (this is how conditional and unconditional jumps, and function call and return are implemented).

Addressing modes

The addressing mode indicates how the operand is presented.

Register Addressing

Operand address R is in the address field.

```
mov ax, bx  ; moves contents of register bx into ax
```

Immediate

Aactual value is in the field.

```
mov ax, 1   ; moves value of 1 into register ax
```

Or:

```
mov ax, 010Ch ; moves value of 0x010C into register ax
```

Direct memory addressing

Operand address is in the address field.

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov ax, [my_var] ; copy my_var content in ax (ax=0xabcd)
```

Direct offset addressing

Uses arithmetics to modify address.

```
byte_tbl db 12,15,16,22,..... ; Table of bytes
mov al,[byte_tbl+2]
mov al,byte_tbl[2] ; same as the former
```

Register Indirect

Field points to a register that contains the operand address.

```
mov ax,[di]
```

The registers used for indirect addressing are BX, BP, SI, DI

Base-index

```
mov ax,[bx + di]
```

For example, if we are talking about an array, BX contains the address of the beginning of the array, and DI contains the index into the array.

Base-index with displacement

```
mov ax,[bx + di + 10]
```

CPU Operation Modes
Real Mode
Real Mode is a holdover from the original Intel 8086. The Intel 8086 accessed memory using 20-bit addresses. But, as the processor itself was 16-bit, Intel invented an addressing scheme that provided a way of mapping a 20-bit addressing space into 16-bit words. Today's x86 processors start in the so-called Real Mode, which is an operating mode that mimics the behavior of the 8086, with some very tiny differences, for backwards compatibility.
Protected Mode
Flat Memory Model
If programming in a modern operating system (such as Linux, Windows), you are basically

programming in flat 32-bit mode. Any register can be used in addressing, and it is generally more efficient to use a full 32-bit register instead of a 16-bit register part. Additionally, segment registers are generally unused in flat mode, and it is generally a bad idea to touch them.

Multi-Segmented Memory Model

Using a 32-bit register to address memory, the program can access (almost) all of the memory in a modern computer. For earlier processors (with only 16-bit registers) the segmented memory model was used. The 'CS', 'DS', and 'ES' registers are used to point to the different chunks of memory. For a small program (small model) the CS=DS=ES. For larger memory models, these 'segments' can point to different locations.

Register Transfer Language And Micro Operations:

Register Transfer language:

- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called microoperations
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best by specifying
  - The set of registers it contains and their functions
  - The sequence of microoperations performed on the binary information stored
  - The control that initiates the sequence of microoperations

Use symbols, rather than words, to specify the sequence of microoperations
The symbolic notation used is called a register transfer language
A programming language is a procedure for writing symbols to specify a given computational process
Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

Register Transfer

Designate computer registers by capital letters to denote its function.
The register that holds an address for the memory unit is called MAR.
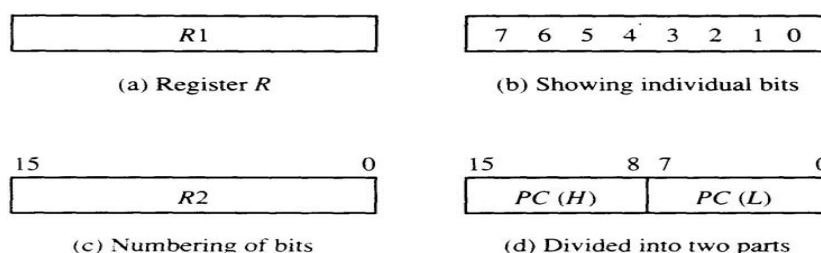The program counter register is called PC.
IR is the instruction register and R1 is a processor register
The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1
Refer to Figure 4.1 for the different representations of a register

Figure 4-1   Block diagram of register.

(a) Register R

(b) Showing individual bits

(c) Numbering of bits

(d) Divided into two parts

- Designate information transfer from one register to another by R2 ← R1
- This statement implies that the hardware is available
  - The outputs of the source must have a path to the inputs of the destination
  - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by

If $(P = 1)$ then $(R2 \leftarrow R1)$ or, $P: R2 \leftarrow R1$, where P is a control function that can be either 0 or 1

  - Every statement written in register transfer notation implies the presence of the required hardware construction
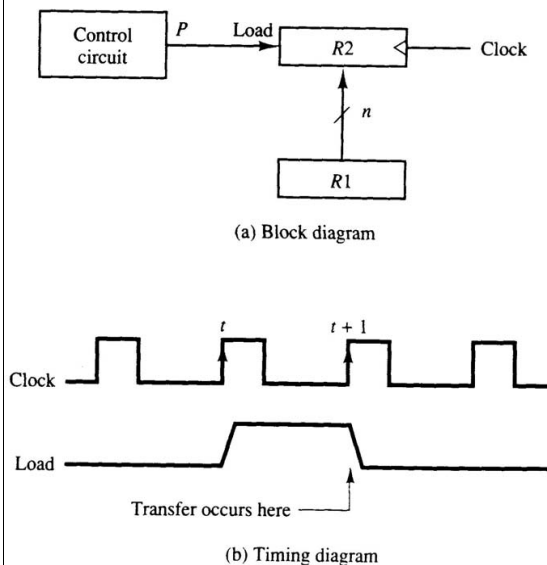
Figure 4-2   Transfer from R1 to R2 when $P = 1$.

(a) Block diagram

(b) Timing diagram

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	MAR, R2
Parentheses ( )	Denotes a part of a register	R2(0–7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Comma ,	Separates two microoperations	R2 ← R1, R1 ← R2

Arithmetic Micro-operations

There are four categories of the most common micro operations:

Register transfer: transfer binary information from one register to another

Arithmetic: perform arithmetic operations on numeric data stored in registers

Logic: perform bit manipulation operations on non-numeric data stored in registers

Shift: perform shift operations on data stored in registers

The basic arithmetic micro operations are addition, subtraction, increment, decrement, and shift

Example of addition: $R3 \leftarrow R1 + R2$

Subtraction is most often implemented through complementation and addition

Example of subtraction: $R3 \leftarrow R1 + \overline{R2} + 1$ (strikethrough denotes bar on top – 1's complement of R2)

Adding 1 to the 1's complement produces the 2's complement

Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting
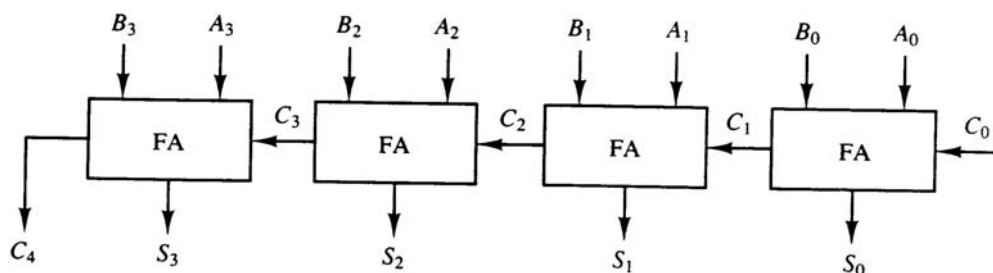
Figure 4-6   4-bit binary adder.

Multiply and divide are not included as micro operations

A micro operation is one that can be executed by one clock pulse

Multiply (divide) is implemented by a sequence of add and shift micro operations (subtract and shift)

To implement the add micro operation with hardware, we need the registers that hold the data and the digital component that performs the addition

A full-adder adds two bits and a previous carry

A binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of any length

A binary added is constructed with full-adder circuits connected in cascade

An n-bit binary adder requires n full-adders

The subtraction A-B can be carried out by the following steps

Take the 1"s complement of B (invert each bit)

Get the 2"s complement by adding 1

Add the result to A

The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

The increment micro operation adds one to a number in a register

This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one

If the increment is to be performed independent of a particular register, then use half-adders connected in cascade

An n-bit binary incrementer requires n half-adders

Each of the arithmetic micro operations can be implemented in one composite arithmetic circuit

The basic component is the parallel adder

Multiplexers are used to choose between the different operations

The output of the binary adder is calculated from the following sum: $D = A + Y + C_{in}$

Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by

    P: R1 □   R1⊕ R2
- Example: R1 = 1010 and R2 = 1100

    1010   Content of R1
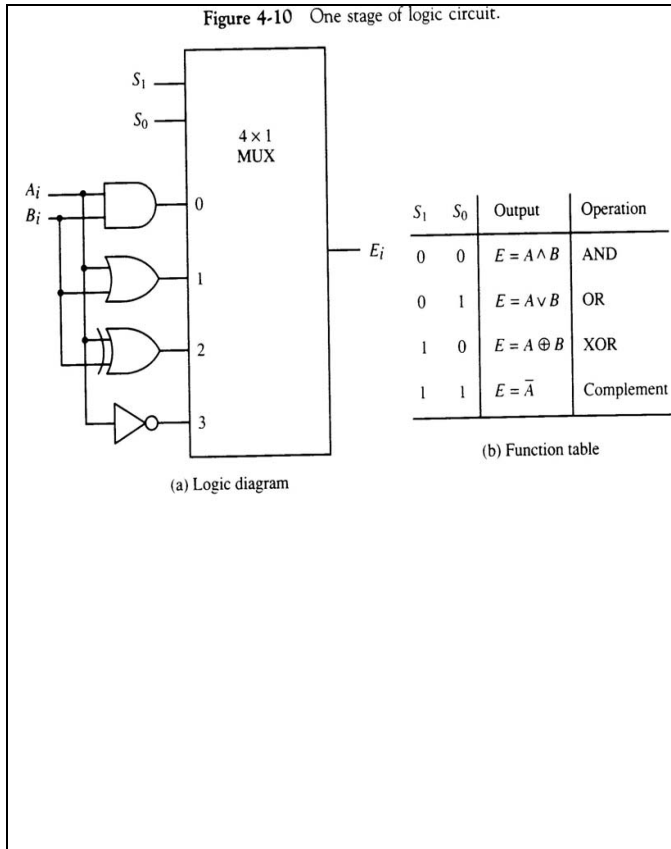   <u>1100</u>   Content of R2

    0110   Content of R1 after P = 1
- Symbols used for logical microoperations:
    - OR: □
    - AND: □
    - XOR: ⊕
- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation

- When + is in a control function, then OR
- Example:

P + Q: R1 ⬚ R2 + R3, R4 ⬚ R5 ⬚ R6

- There are 16 different logic operations that can be performed with two binary variables
- The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers
- All 16 microoperations can be derived from using four logic gates

Figure 4-10 One stage of logic circuit.

(a) Logic diagram

(b) Function table

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The selective-set operation sets to 1 the bits in A where there are corresponding 1"s in B

1010   A before
1100   B
(logic operand) 1110 A after A ⬚ A ⬚ B

- The selective-complement operation complements bits in A where there are corresponding 1"s in B

1010   A before
1100   B
(logic operand) 0110 A after
A ⬚ A⊕B

- The selective-clear operation clears to 0 the bits in A only where there are corresponding 1"s in B

1010   A before
1100   B (logic operand) 0010   A
after A ⬚ A ⬚ B

- The mask operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0"s in B

1010   A before
1100   B
(logic operand) 1000 A
after A ⬚ A ⬚ B

- The insert operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

0110 1010	A before
0000 1111	B (mask)
0000 1010	A after masking
0000 1010	A before
1001 0000	B (insert)
1001 1010	A after insertion

- The clear operation compares the bits in A and B and produces an all 0's result if the two number are equal

1010   A
1010   B
0000   A $\square$ A $\oplus$ B

Shift Microoperations

Shift microoperations are used for serial transfer of data

They are also used in conjunction with arithmetic, logic, and other data- processing operations

There are three types of shifts: logical, circular, and arithmetic

A logical shift is one that transfers 0 through the serial input

The symbols shl and shr are for logical shift-left and shift-right by one position R1 $\square$ shlR

The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information

The symbols cil and cir are for circular shift left and right

The arithmetic shift shifts a signed binary number to the left or right.

To the left is multiplying by 2, to the right is dividing by 2.

Arithmetic shifts must leave the sign bit unchanged.

A sign reversal occurs if the bit in R_{n-1} changes in value after the shift.

This happens if the multiplication causes an overflow.

An overflow flip-flop V_s can be usedto detect

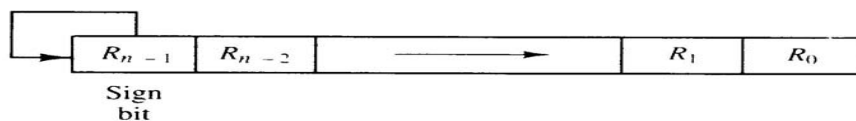theoverflow $V_s = R_{n-1} \oplus R_{n-2}$

Figure 4-11   Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift
- In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit
- The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register
- This can be constructed with multiplexers

Arithmetic Logic Unit
- The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers
- To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU
- The ALU performs an operation and the result is then transferred to a destination register
- The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period
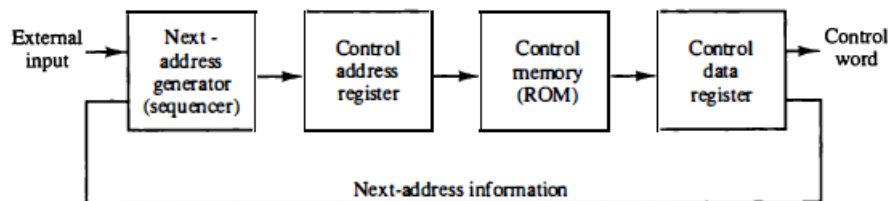
Micro Programmed Control

A control unit whose binary control variables are stored in memory is called a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram .Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk.

Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading.

A memory that is part of a control unit is referred to as a control memory.

Figure 7-1 Microprogrammed control organization.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory.

The control data register holds the present microinstruction while the next address is computed and read from memory.

The data register is sometimes called a pipeline register.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory. It should be mentioned that most computers based on the reduced instruction set computer (RISC).

Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a routine.
The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.

A mapping procedure is a rule that transforms the instruction code into a control memory address
1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return

Conditional Branching

Special Bits : The branch logic provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions

Branch Logic : The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented. This can be implemented with a multiplexer.

Mapping of Instruction: A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction.

Microinstruction Format

The microinstruction format for the control memory is shown in Fig. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 27 words.

15	14	11	10		0
I	Opcode		Address		

(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from Fl.

$$DR \leftarrow M[AR] \qquad \text{with F2} = 100$$
$$\text{and} \quad PC \leftarrow PC + 1 \qquad \text{with F3} = 101$$

Memory system design: Semiconductor memory technologies, memory organization.
Memory organization: Memory interleaving, concept of hierarchical memory organization, Cache memory, cache size vs. block size, mapping functions, Replacement algorithms, write policies.

Semiconductor Memory Technologies:

Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. Semiconductor memory is used in any electronics assembly that uses computer processing technology. The use of semiconductor memory has grown, and the size of these memory cards has increased as the need for larger and larger amounts of storage is needed.
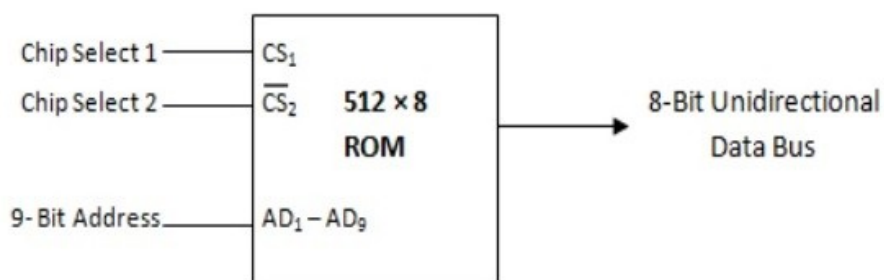
There are two main types or categories that can be used for semiconductor technology.

RAM - Random Access Memory: As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.

Block Diagram Representing 128 x 8 RAM
(Random Access Memory)

ROM - Read Only Memory: A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.

The different memory types or memory technologies are detailed below:

DRAM: Dynamic RAM is a form of random access memory. DRAM uses a capacitor to store each bit of data, and the level of charge on each capacitor determines whether that bit is a logical 1 or 0. However these capacitors do not hold their charge indefinitely, and therefore the data needs to be refreshed periodically. As a result of this dynamic refreshing it gains its name of being a dynamic RAM. DRAM is the form of semiconductor memory that is often used in equipment including personal computers and workstations where it forms the main RAM for the computer.

EEPROM: This is an Electrically Erasable Programmable Read Only Memory. Data can be written to it and it can be erased using an electrical voltage. This is typically applied to an erase pin on the chip. Like other types of PROM, EEPROM retains the contents of the memory even when the power is turned off. Also like other types of ROM, EEPROM is not as fast as RAM.

EPROM: This is an Erasable Programmable Read Only Memory. This form of semiconductor memory can be programmed and then erased at a later time. This is normally achieved by exposing the silicon to ultraviolet light. To enable this to happen there is a circular window in the package of the EPROM to enable the light to reach the silicon of the chip. When the PROM is in use, this window is normally covered by a label, especially when the data may need to be preserved for an extended period. The PROM stores its data as a charge on a capacitor. There is a charge storage capacitor for each cell and this can be read repeatedly as required. However it is found that after many years the charge may leak away and the data may be lost. Nevertheless, this type of semiconductor memory used to be widely used in applications where a form of ROM was required, but where the data needed to be changed periodically, as in a development environment, or where quantities were low.

FLASH MEMORY: Flash memory may be considered as a development of EEPROM technology. Data can be written to it and it can be erased, although only in blocks, but data can be read on an individual cell basis. To erase and re-programme areas of the chip, programming voltages at levels that are available within electronic equipment are used. It is also non-volatile, and this makes it particularly useful. As a result Flash memory is widely used in many applications including memory cards for digital cameras, mobile phones, computer memory sticks and many other applications.

F-RAM: Ferroelectric RAM is a random-access memory technology that has many similarities to the standard DRAM technology. The major difference is that it incorporates a ferroelectric layer instead of the more usual dielectric layer and this provides its non-volatile capability. As it offers a non-volatile capability, F-RAM is a direct competitor to Flash.

MRAM: This is Magneto-resistive RAM, or Magnetic RAM. It is a non-volatile RAM memory technology that uses magnetic charges to store data instead of electric charges. Unlike technologies including DRAM, which require a constant flow of electricity to maintain the integrity of the data, MRAM retains data even when the power is removed. An additional advantage is that it only requires low power for active operation. As a result this technology could become a major player in the electronics industry now that production processes have been developed to enable it to be produced.

P-RAM / PCM: This type of semiconductor memory is known as Phase change Random Access Memory, P-RAM or just Phase Change memory, PCM. It is based around a phenomenon where a form of chalcogenide glass changes is state or phase between an amorphous state (high resistance) and a polycrystalline state (low resistance). It is possible to detect the state of an individual cell and hence use this for data storage. Currently this type of memory has not been widely commercialized, but it is expected to be a competitor for flash memory.

PROM: This stands for Programmable Read Only Memory. It is a semiconductor memory which can only have data written to it once - the data written to it is permanent. These memories are bought in a blank format and they are programmed using a special PROM programmer. Typically a PROM will consist of an array of fuseable links some of which are "blown" during the programming process to provide the required data pattern.

SDRAM: Synchronous DRAM. This form of semiconductor memory can run at faster speeds than conventional DRAM. It is synchronised to the clock of the processor and is capable of keeping two sets of memory addresses open simultaneously. By transferring data alternately from one set of addresses, and then the other, SDRAM cuts down on the delays associated with non-synchronous RAM, which must close one address bank before opening the next.
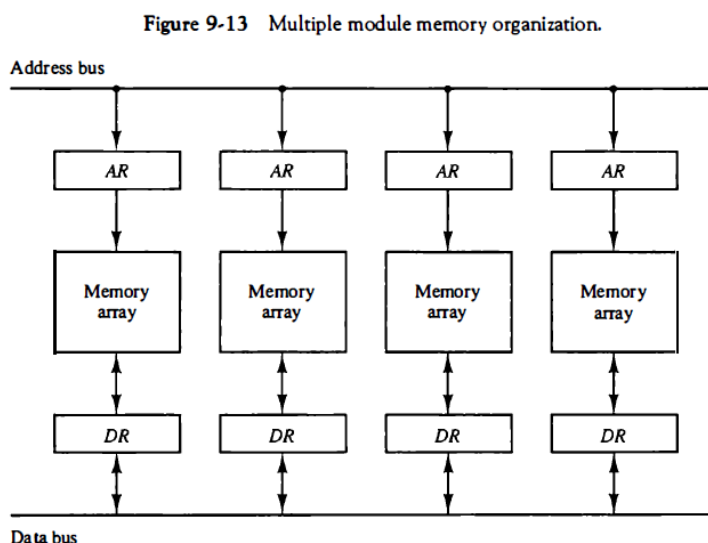
SRAM: Static Random Access Memory. This form of semiconductor memory gains its name from the fact that, unlike DRAM, the data does not need to be refreshed dynamically. It is able to support faster read and write times than DRAM (typically 10 ns against 60 ns for DRAM), and in addition its cycle time is much shorter because it does not need to pause between accesses. However it consumes more power, is less dense and more expensive than DRAM. As a result of this it is normally used for caches, while DRAM is used as the main semiconductor memory technology.

MEMORY ORGANIZATION

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.

Similarly, 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. Figure 9-13 shows a memory unit with four modules. Each memory array has its own address register AR and data register DR.

Figure 9-13 Multiple module memory organization.

The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

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. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

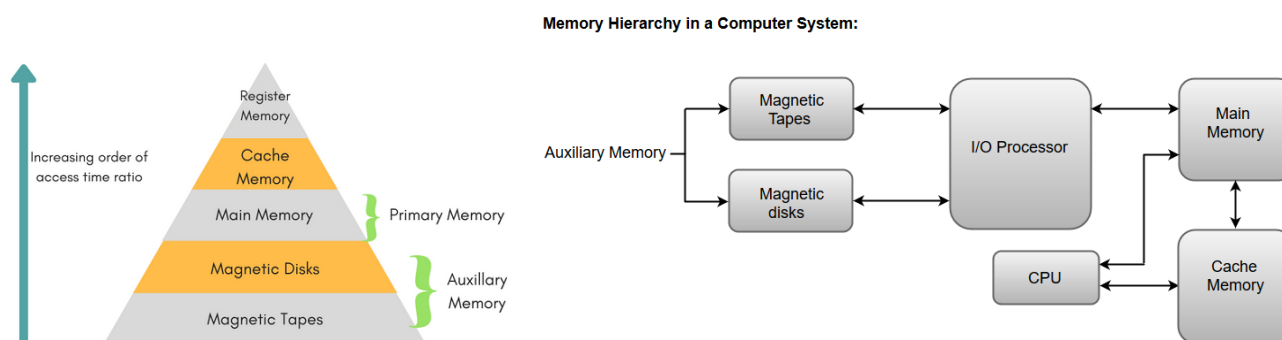## Concept of Hierarchical Memory Organization
This Memory Hierarchy Design is divided into 2 main types:
External Memory or Secondary Memory
Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via I/O Module.
Internal Memory or Primary Memory
Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

Memory Hierarchy in a Computer System:

Characteristics of Memory Hierarchy

Capacity:
It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.
Access Time:
It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.
Performance:
Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.
Cost per bit:
As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Cache Memories:
  The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.
  Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.
  The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.
  When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's *replacement algorithm*.

Cache Hits
  The processor does not need to know explicitly about the existence of the cache. It simply issues Read andWrite requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.
If it does, the Read orWrite operation is performed on the appropriate cache location. In this case, a *read* or *write hit* is said to have occurred.

Cache Misses
  A Read operation for a word that is not in the cache constitutes a *Read miss*. It causes the block of words containing the requested word to be copied from the main memory into the cache.

Cache Mapping:
  There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.
Direct mapping
  The simplest way to determine cache locations in which to store memory blocks is the *direct-mapping* technique. In this technique, block *j* of the main memory maps onto block *j* modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.
  For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.
  With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block.
  When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.
  The direct-mapping technique is easy to implement, but it is not very flexible.
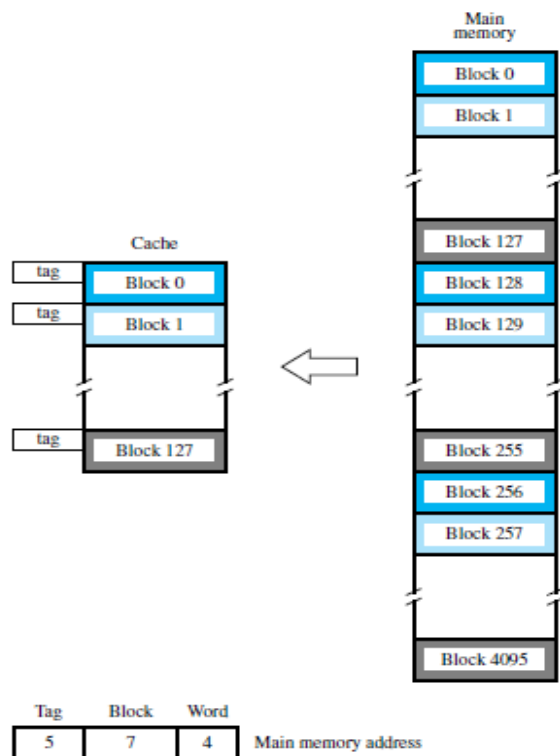
Tag	Block	Word
5	7	4	Main memory address

Figure 8.16   Direct-mapped cache.

Associative Mapping

In Associative mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique.

Tag	Word
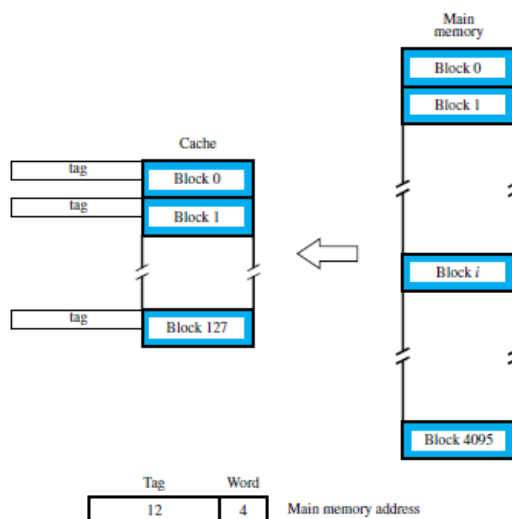| 12 | 4 | Main memory address

Figure 8.17   Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced.

To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an *associative search*.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.
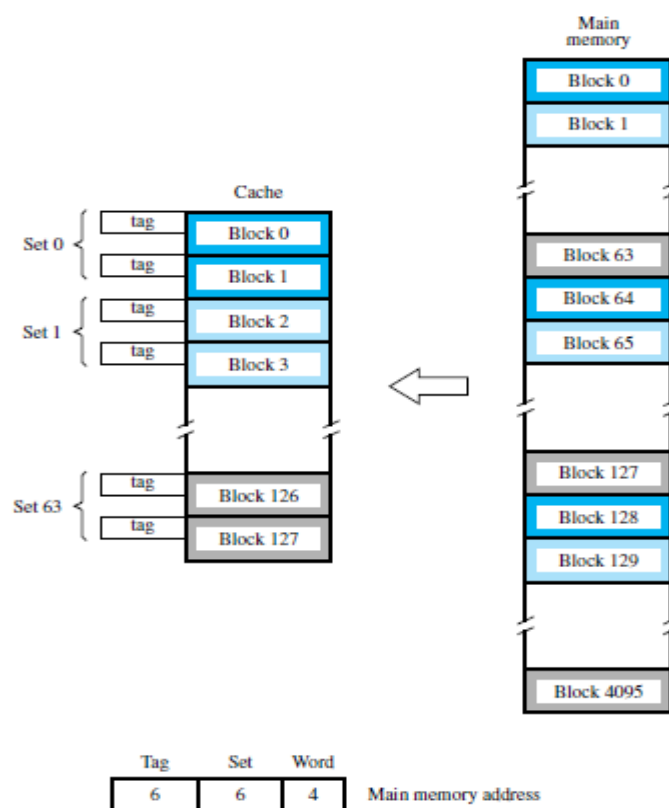

Figure 8.18   Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set.

Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.

This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases.

Write Policies

The write operation is proceeding in 2 ways.

- Write-through protocol
- Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

- This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit.
- The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block.
- To overcome the read miss Load –through / Early restart protocol is used.
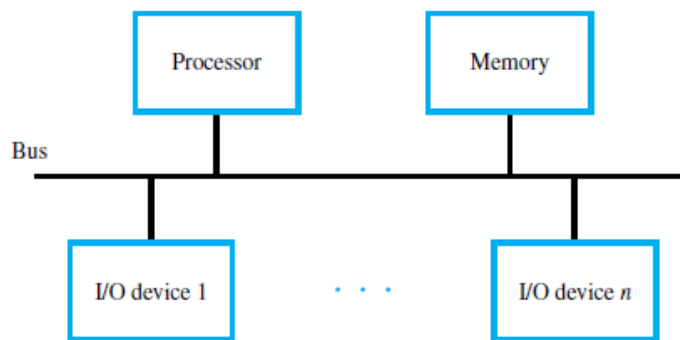
> **Peripheral devices and their characteristics:** Input-output subsystems, I/O device interface, I/O transfers – program controlled, interrupt driven and DMA, privileged and non-privileged instructions, software interrupts and exceptions. Programs and processes – role of interrupts in process state transitions, I/O device interfaces – SCII, USB

Input-output subsystems

The Input/output organization of computer depends upon the size of computer and the peripherals connected to it. The I/O Subsystem of the computer provides an efficient mode of communication between the central system and the outside environment**.**

The most common input output devices are: Monitor, Keyboard, Mouse, Printer, Magnetic tapes Input Output Interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.

The Major Differences are:-
- Peripherals are electromechanical and electromagnetic devices and CPU and memory are electronic devices. Therefore, a conversion of signal values may be needed.
- The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a synchronization mechanism may be needed.
- Data codes and formats in the peripherals differ from the word format in the CPU and memory.
- The operating modes of peripherals are different from each other and must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and Peripherals to supervises and synchronizes all input and out transfers. These components are called Interface Units because they interface between the processor bus and the peripheral devices.

I/O device interface
The I/O Bus consists of data lines, address lines and control lines. The I/O bus from the processor is attached to all peripherals interface. To communicate with a particular device, the processor places a device address on address lines. Each Interface decodes the address and control received from the I/O bus, interprets them for peripherals and provides signals for the peripheral controller. It is also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller.

For example, the printer controller controls the paper motion, the print timing. The control lines are referred as I/O command. The commands are as following:

Control command- A control command is issued to activate the peripheral and to inform it what to do.

Status command- A status command is used to test various status conditions in the interface and the peripheral.

Data Output command- A data output command causes the interface to respond by transferring data from the bus into one of its registers.

Data Input command- The data input command is the opposite of the data output.

In this case the interface receives on item of data from the peripheral and places it in its buffer register.
I/O Versus Memory Bus

To communicate with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address and read/write control lines. There are 3 ways that computer buses can be used to communicate with memory and I/O:

1. Use two Separate buses, one for memory and other for I/O.
2. Use one common bus for both memory and I/O but separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

Figure 7.2 I/O interface for an input device.

Programmed I/O Mode:

In this mode of data transfer the operations are the results in I/O instructions which is a part of computer program. Each data transfer is initiated by a instruction in the program. Normally the transfer is from a CPU register to peripheral device or vice-versa. Once the data is initiated the CPU starts monitoring the interface to see when next transfer can made. The instructions of the program keep close tabs on everything that takes place in the interface unit and the I/O devices.

The transfer of data requires three instructions:
- Read the status register.
- Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
- Read the data register.

Figure   Data transfer from I/O device to CPU.

In this technique CPU is responsible for executing data from the memory for output and storing data in memory for executing of Programmed I/O as shown in Fig.

Drawback of the Programmed I/O:

The main drawback of the Program Initiated I/O was that the CPU has to monitor the units all the times when the program is executing. Thus the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process and the CPU time is wasted a lot in keeping an eye to the executing of program.

Interrupt-Initiated I/O:

In this method an interrupt facility an interrupt command is used to inform the device about the start and end of transfer. In the meantime the CPU executes other program. When the interface determines that the device is ready for data transfer it generates an Interrupt Request and sends it to the computer.

When the CPU receives such an signal, it temporarily stops the execution of the program and branches to a service program to process the I/O transfer and after completing it returns back to task, what it was originally performing.

In this type of IO, computer does not check the flag. It continues to perform its task. Whenever any device wants the attention, it sends the interrupt signal to the CPU.CPU then deviates from what it was doing, store the return address from PC and branch to the address of the subroutine.

There are two ways of choosing the branch address:

Vectored Interrupt: In vectored interrupt the source that interrupts the CPU provides the branch information. This information is called interrupt vectored.

Non-vectored Interrupt: In non-vectored interrupt, the branch address is assigned to the fixed address in the memory.

Direct Memory Access (DMA):

In the Direct Memory Access (DMA) the interface transfer the data into and out of the memory unit through the memory bus. The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called Direct Memory Access (DMA).
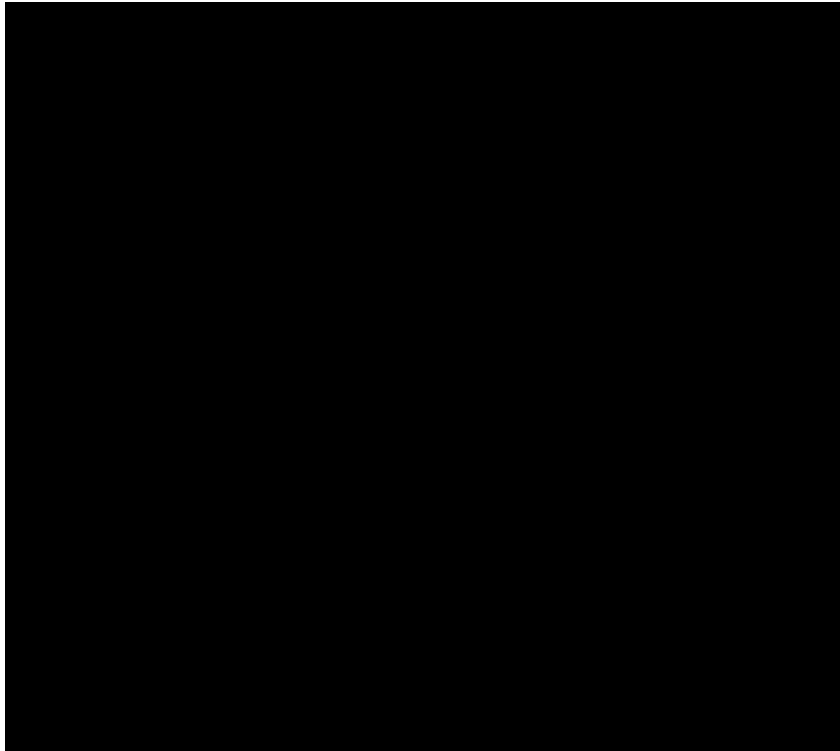
During the DMA transfer, the CPU is idle and has no control of the memory buses. A DMA Controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessor is to disable the buses through special control signals such as:

- Bus Request (BR)
- Bus Grant (BG)

These two control signals in the CPU that facilitates the DMA transfer. The Bus Request (BR) input

is used by the DMA controller to request the CPU. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, data bus and read write lines into a high Impedance state. High Impedance state means that the output is disconnected.

The CPU activates the Bus Grant (BG) output to inform the external DMA that the Bus Request (BR) can now take control of the buses to conduct memory transfer without processor.
When the DMA terminates the transfer, it disables the Bus Request (BR) line. The CPU disables the Bus Grant (BG), takes control of the buses and return to its normal operation.

The transfer can be made in several ways that are:

- DMA Burst
- Cycle Stealing

DMA Burst: In DMA Burst transfer, a block sequence consisting of a number of memory words is transferred in continuous burst while the DMA controller is master of the memory buses.
Cycle Stealing: Cycle stealing allows the DMA controller to transfer one data word at a time, after which it must returns control of the buses to the CPU.

DMA Controller:
The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. The DMA controller has three registers:

- Address Register
- Word Count Register
- Control Register

Address Register: Address Register contains an address to specify the desired location in memory.
Word Count Register: WC holds the number of words to be transferred. The register is incre/decre by one after each word transfer and internally tested for zero.
Control Register: Control Register specifies the mode of transfer

       The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (Register select) inputs. The RD (read) and WR (write) inputs are bidirectional.
When the BG (Bus Grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG =1, the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control.
DMA Transfer:
       The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can transfer between the peripheral and the memory.

       When BG = 0 the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG=1, the RD and WR are output lines from the DMA controller to the random access memory to specify the read or write operation of data.

Privileged Instructions and Non- Privileged Instructions:

Instructions are divided into two categories:
- non-privileged instructions
- privileged instructions.

A non-privileged instruction is an instruction that any application or user can execute.

Examples of non-privileged instructions:

```
1  movl
2  addl
3  call
4  ret
```

A privileged instruction, on the other hand, is an instruction that can only be executed in kernel mode. Instructions are divided in this manner because privileged instructions could harm the kernel.

Examples of privileged instructions:

```
1  insl
2  outb
3  inb
4  int
```

Exceptions and Software interrupts:

Exceptions and interrupts are unexpected events that disrupt the normal flow of instruction execution. An exception is an unexpected event from within the processor. An interrupt is an unexpected event from outside the processor. You are to implement exception and interrupt handling in your multicycle CPU design.

External interrupts come from input input (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction.

Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure.

A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.
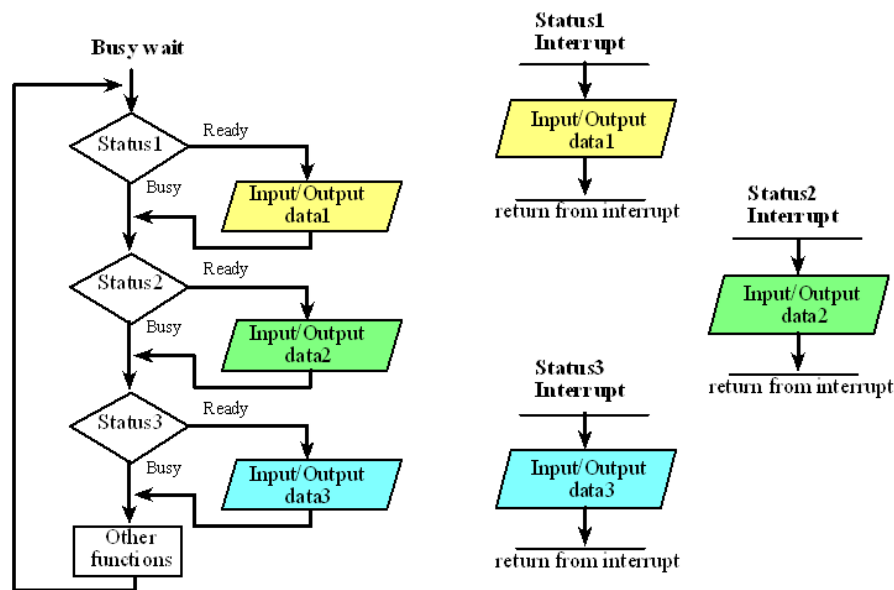

Programs and processes-Role of interrupts in process state transitions

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer).

When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **BX LR**). A new thread is created for each interrupt request.

It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt. In a **multi-threaded** system, we consider the threads as cooperating to perform an overall task. Consequently we will develop ways for the threads to communicate (e.g., FIFO) and to synchronize with each other. Most embedded systems have a single common overall goal.

On the other hand, general-purpose computers can have multiple unrelated functions to perform. A **process** is also defined as the action of software as it executes. Processes do not necessarily cooperate towards a common shared goal. Threads share access to I/O devices, system resources, and global variables, while processes have separate global variables and system resources. Processes do not share I/O devices.

Busy wait

Status1 — Ready → Input/Output data1
Busy

Status2 — Ready → Input/Output data2
Busy

Status3 — Ready → Input/Output data3
Busy

Other functions

Status1 Interrupt → Input/Output data1 → return from interrupt

Status2 Interrupt → Input/Output data2 → return from interrupt

Status3 Interrupt → Input/Output data3 → return from interrupt

I/O Device Interfaces

SCSI:

The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 . In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s. The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options.

A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time. There are also several options for the electrical signaling scheme used. Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa. A packet may contain a block of data, commands from the processor to the device, or status information about the device.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory. A controller connected to a SCSI bus is one of two types – an initiator or a target. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. Clearly, the controller on the processor side, such as the SCSI controller, must be able to operate as an initiator. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. While a particular connection is suspended, other device can use the bus to transfer information. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it.

Then the controller starts a data transfer operation to receive a command from the initiator.

The processor sends a command to the SCSI controller, which causes the following sequence of event to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of these transfers, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends as interrupt to the processor to inform it that the requested operation has been completed.

This scenario show that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. In this context, a "higher level" means that the messages refer to operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of operation of the particular device involved in a data transfer. In the preceding example, the processor need not be involved in the disk seek operation.
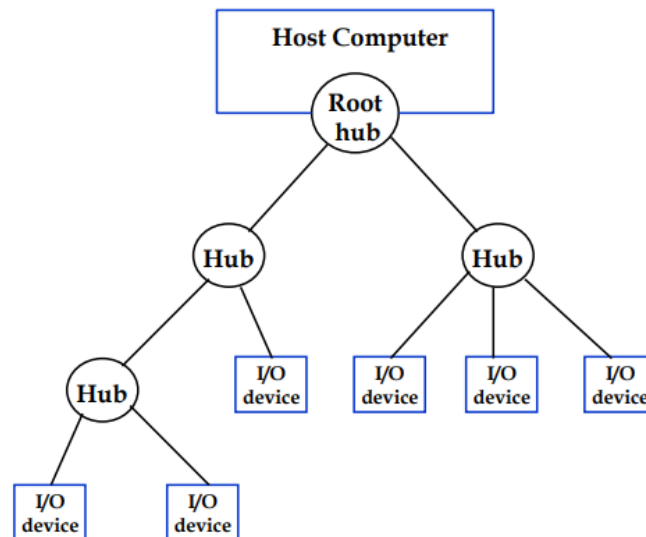
USB

The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer. ‹
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections. ‹
- Enhance user convenience through a "plug-and-play" mode of operation.

USB Structure

- A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements.
- Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew.
- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure. Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O device. At the root of the tree, a root hub connects the entire tree to the host computer.
- The tree structure enables many devices to be connected while using only simple point-to-point serial links.
- Each hub has a number of ports where devices may be connected, including other hubs.

- In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed device will respond to that message.
- A message sent from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

USB Protocols:
- All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information.
- The information transferred on the USB can be divided into two broad categories: control and data. ‹ Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error. Data packets carry information that is delivered to a device. For example, input and output data are transferred inside data packets

Pipelining: Basic concepts of pipelining, throughput and speedup, pipeline hazards.
Parallel Processors: Introduction to parallel processors, Concurrent access to memory and cache coherency.

Basic concepts of pipelining:

Performance of a computer can be increased by increasing the performance of the CPU.
This can be done by executing more than one task at a time. This procedure is referred to as pipelining.
The concept of pipelining is to allow the processing of a new task even though the processing of previous task has not ended.
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. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
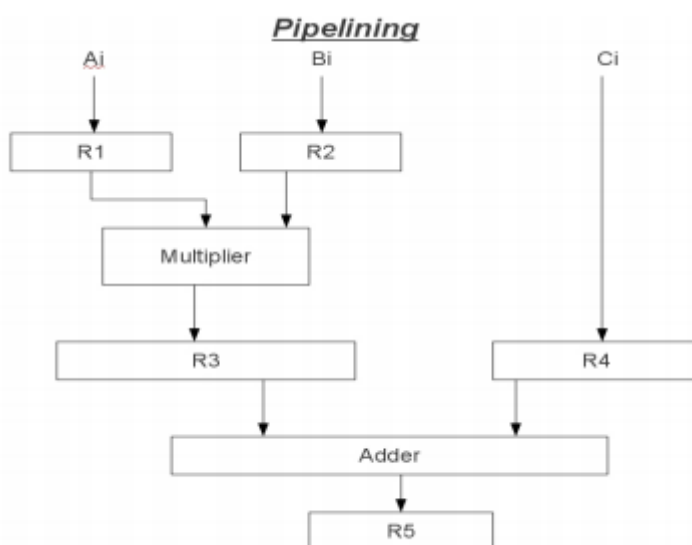
Consider the following operation: **Result=(A+B)*C**
First the A and B values are Fetched which is nothing but a "Fetch Operation".
The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.
The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed.
Finally the Result is again stored in the "Result" variable.

In this process we are using up-to 5 pipelines which are
Fetch Operation (A), Fetch Operation(B)
Addition of (A & B), Fetch Operation(C)
Multiplication of ((A+B), C)
Load ((A+B)*C)

Pipelining

Now consider the case where 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 k t, 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)t, . Therefore, to complete
n tasks using a k-segment pipeline requires k + (n - 1) clock cycles. For example, the diagram of Fig. shows four segments and six tasks.
The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the diagram.

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	—	—	—	—	$A_7 * B_7 + C_7$

Throughput and Speedup

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 inaeasing the computational speed of a computer system. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput.

Throughput: 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. However, technological developments have reduced hardware costs to the point where parallel processing techniques a.re economically feasible.

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

$$S = Tseq / Tpipe = n*m / (m+n -1)$$

the maximum speedup, also called ideal speedup, of a pipeline processor with m stages over an equivalent nonpipelined processor is m. In other words, the ideal speedup is equal to the number of pipeline stages. That is, when n is very large, a pipelined processor can produce output approximately m times faster than a nonpipelined processor. When n is small, the speedup decreases.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

Hazards

The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 4.27 had a fourth instruction, we would see that in the same clock cycle the fi rst instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads must wait that have completed drying and are ready to fold as well as those that have finished washing and are ready to dry.

In a pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):
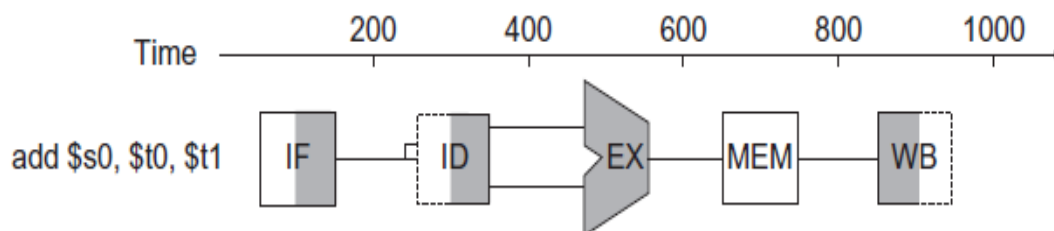
add $s0, $t0, $t1
sub $t2, $s0, $t3

FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too oft en and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.
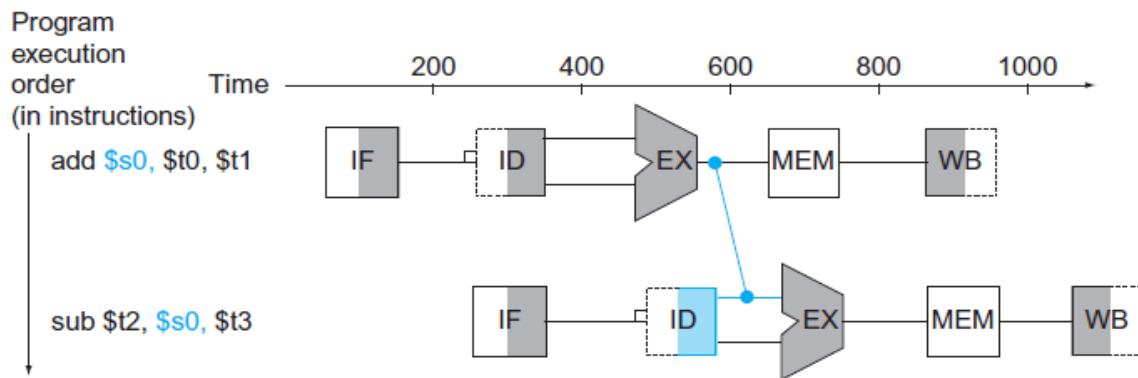
FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path

It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of $s0 instead of an add. As we can imagine from looking at Figure 4.29, the desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**, as Figure 4.30 shows. This figure shows an important pipeline concept, officially called a **pipeline stall**, but oft en given the nickname **bubble**. We shall see stalls elsewhere in the pipeline.

Control Hazards

The third type of hazard is called a **control hazard**, arising from the need to make a decision based on the results of one instruction while others are executing. Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until aft er the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here is the first of two solutions to control hazards in the laundry room and its computer equivalent.

Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula.

This conservative option certainly works, but it is slow.

Parallel Processors
Introduction to parallel processors:

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 in a easing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.
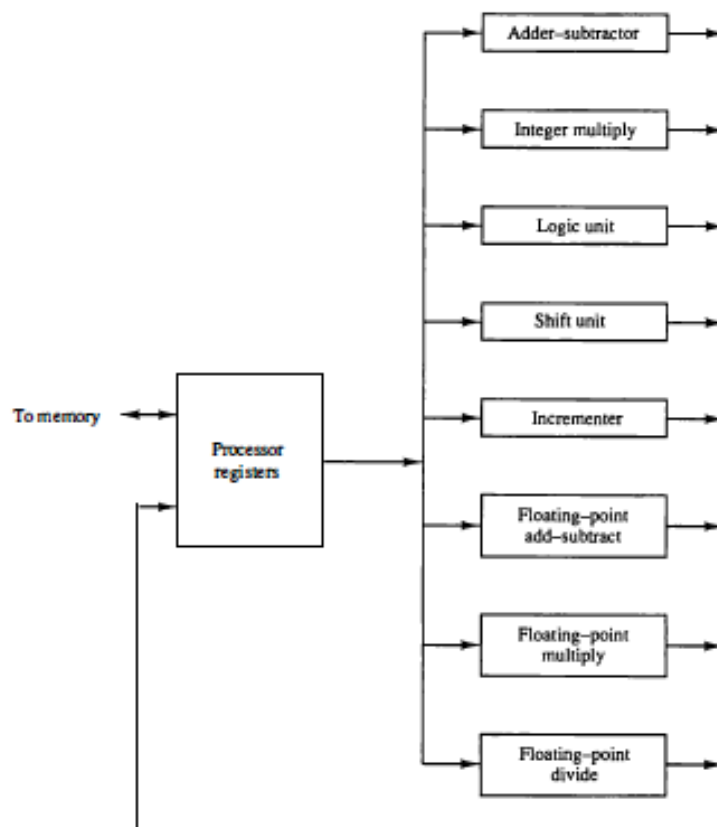
The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques a.re economically feasible.

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. Shift registers operate in serial fashion one bit at a time, while registers with parallel load operate with all the bits of the word simultaneously.

Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel. 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.

Figure 9-1   Processor with multiple functional units.

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. 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 constitutes a data stream . Parallel processing may occur in the instruction stream, in the data stream, 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)

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 in this case may be achieved by means of multiple functional units or by pipeline processing.

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.

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

MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.

Concurrent access to memory and cache coherency:
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.
Write-through policy: In the write-through policy, both cache and main memory are updated with every write operation.
Write-back policy: 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. 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.
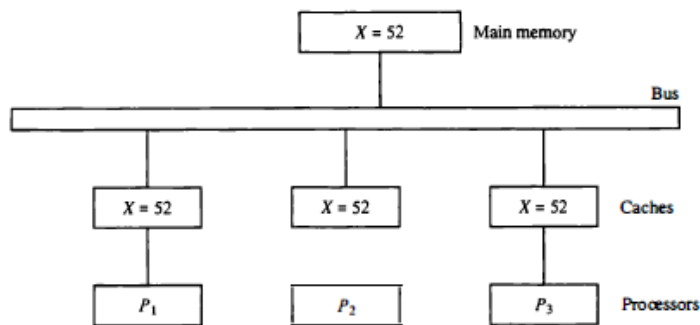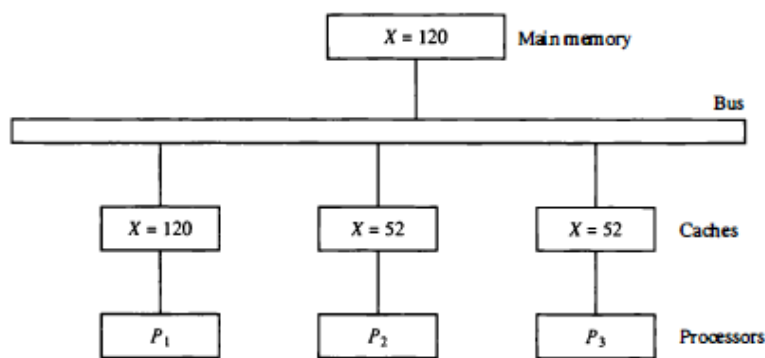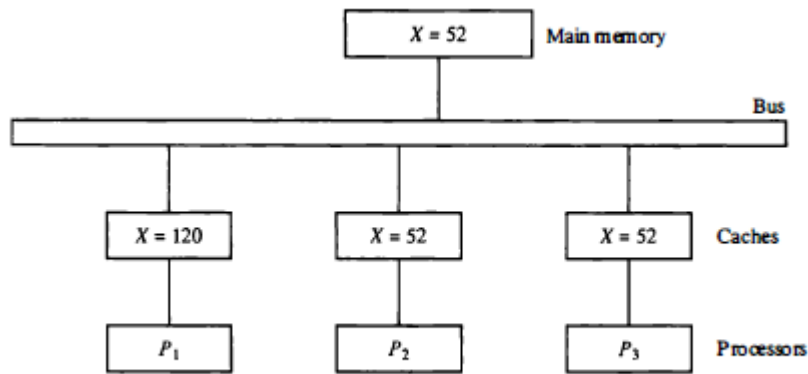

Figure 13-12 Cache configuration after a load on X.

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. 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.

Figure 13-13 Cache configuration after a store to X by processor P₁.

(a) With write-through cache policy

| X = 52 | Main memory |

Bus

| X = 120 | X = 52 | X = 52 | Caches |

| P_1 | P_2 | P_3 | Processors |

(b) With write-back cache policy

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. VO-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.