B K Birla Institute of Engineering & Technology, Pilani, Rajasthan
Computer Architecture & Organization – 6CS4-04
Computer Science (CS B) – III Year/VI Semester
UNIT V
Computer Arithmetic

1. Introduction:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations. To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data.

Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation. If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms. In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods. A processor has an arithmetic processor (as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

2. Addition and Subtraction:

Addition and Subtraction with Signed –Magnitude Data

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

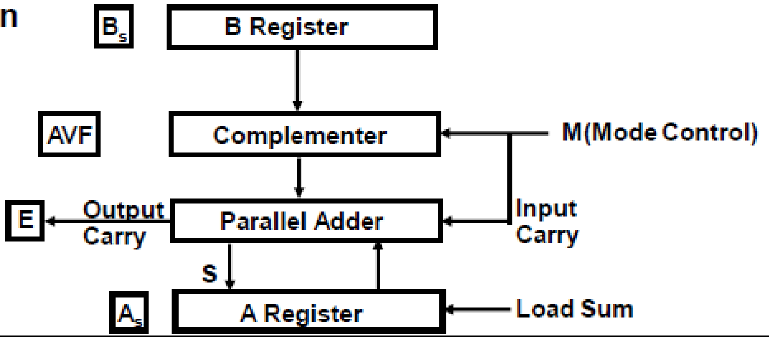
The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm) Addition and Subtraction of Signed-Magnitude Numbers

SIGNED MAGNITUDEADDITION AND SUBTRACTION

Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

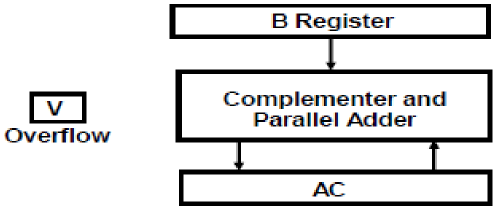
Operation	Add Magnitude	Subtract Magnitude		
		When A>B	When A<B	When A=B
(+A) + (+B)	+(A + B)			
(+A) + (- B)		+(A - B)	- (B - A)	+(A - B)
(- A) + (+B)		- (A - B)	+(B - A)	+(A - B)
(- A) + (- B)	- (A + B)			
(+A) - (+B)		+(A - B)	- (B - A)	+(A - B)
(+A) - (- B)	+(A + B)			
(- A) - (+B)	- (A + B)			
(- A) - (- B)		- (A - B)	+(B - A)	+(A - B)

Hardware Implementation

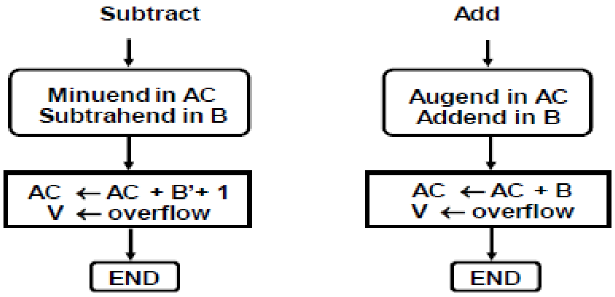


SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION

Hardware

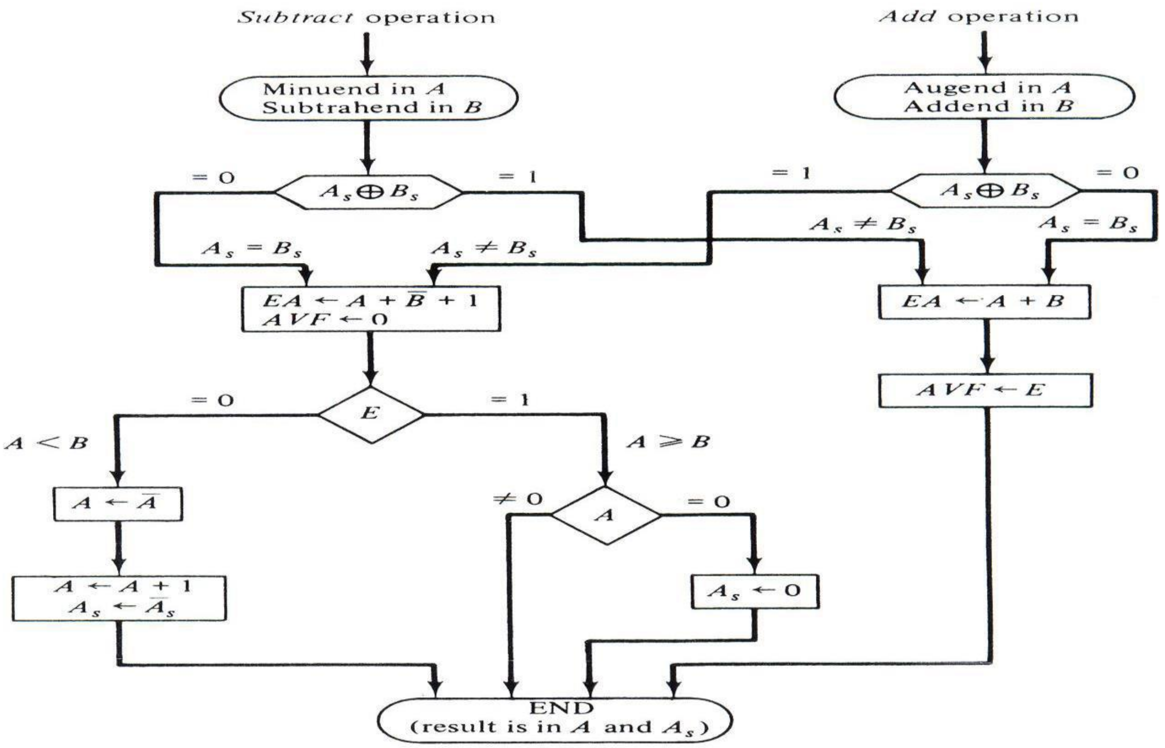


Algorithm



Algorithm:

- The flowchart is shown in Figure. The two signs A, and B, are compared by an exclusive-OR gate.
  If the output of the gate is 0 the signs are identical; If it is 1, the signs are different
- For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.
- The magnitudes are added with a microoperation EA A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.
- The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that A >= B and the number in A is the correct result. If this number is zero, the sign A must be made positive to avoid a negative zero.
- 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A A' +1.
- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.
- In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.
- The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.
- Figure shows a block diagram of the hardware for implementing the addition and subtraction operations.
- It consists of registers A and B and sign flip-flops As and Bs. Subtraction is done by adding A to the 2's complement of B.
- The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.
- The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.
- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

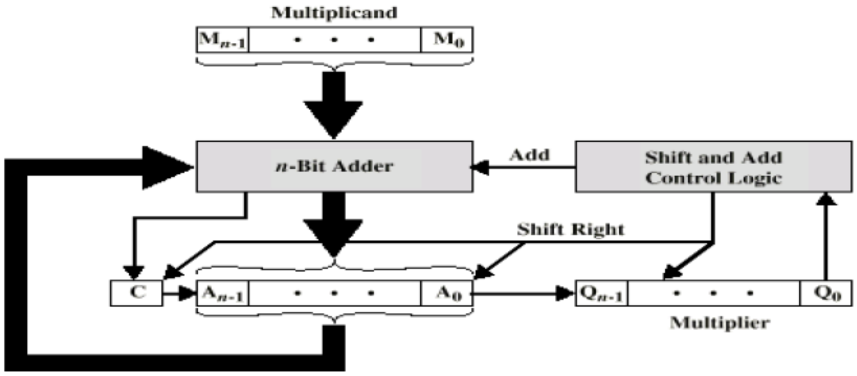


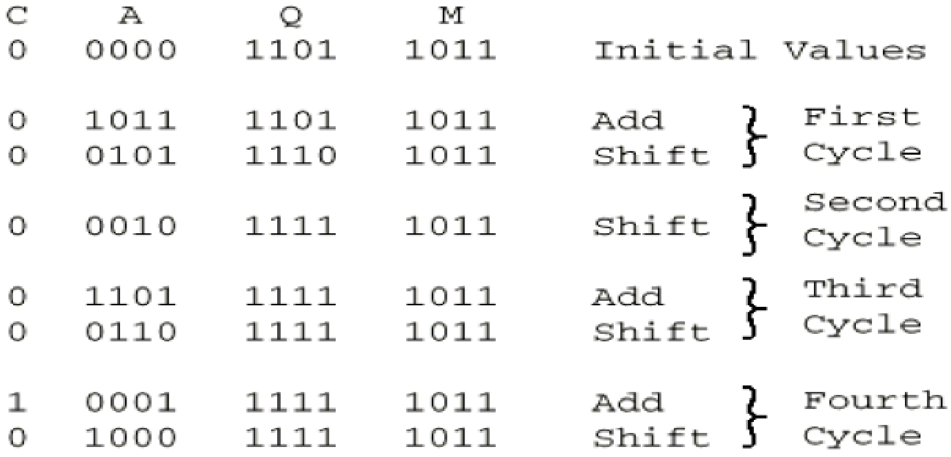
Flowchart for ADD and SUBTRACT Operation

3. Multiplication Algorithm:

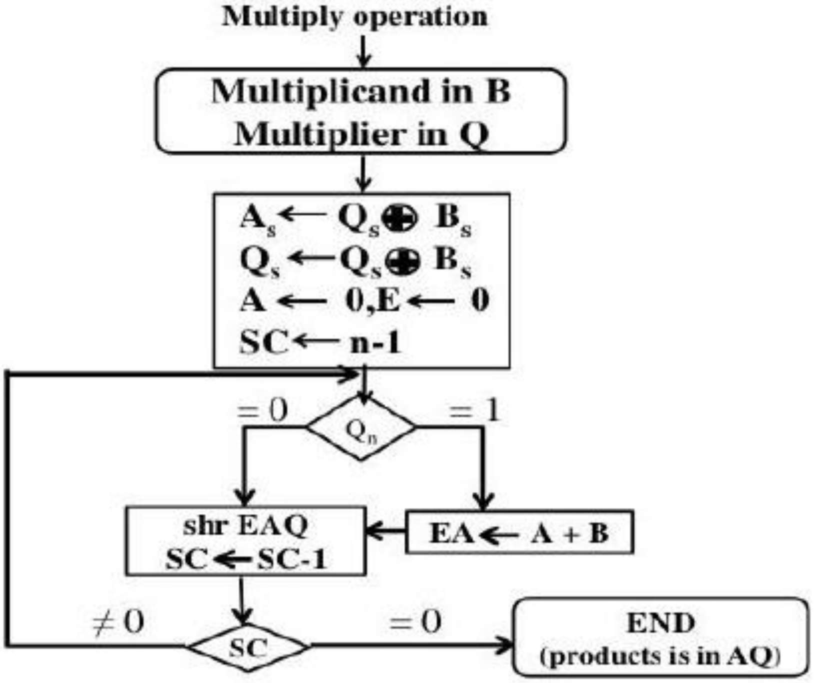
In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Now, the low order bit of the multiplier in Qn is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0 we stops the process.





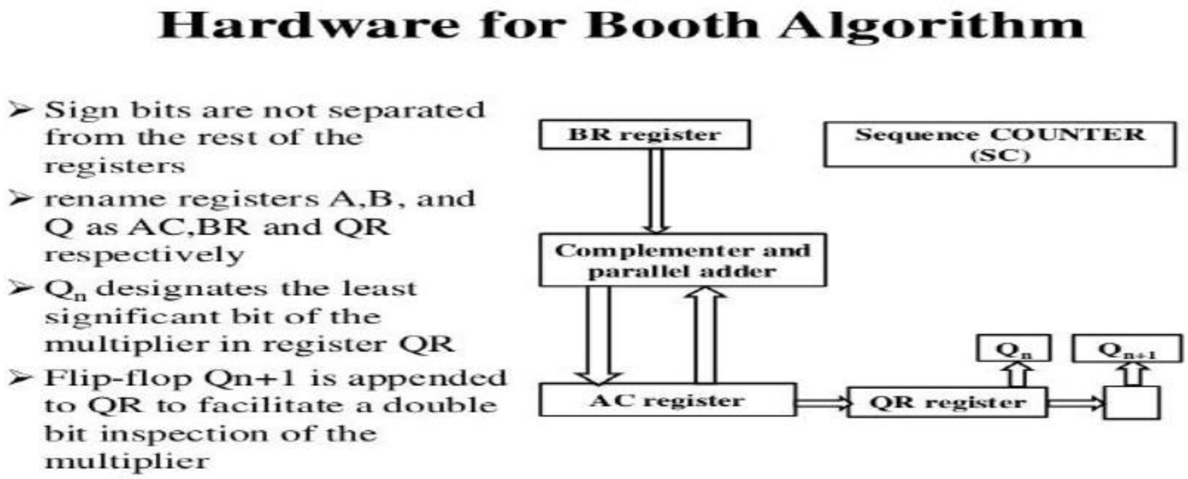
C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	



Flowchart for Multiply Operation

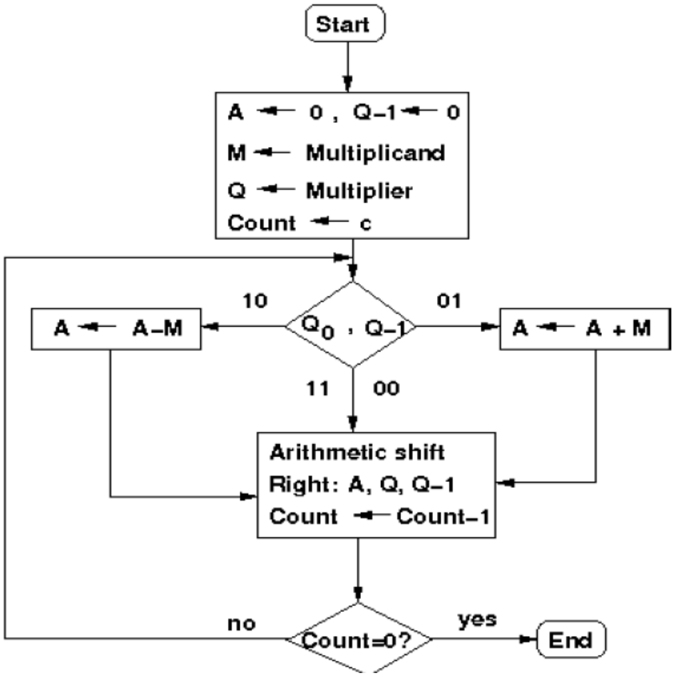
4. Booth's algorithm

- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} – 2^m$.
- For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 (k=3, m=1). The number can be represented as $2^{k+1} – 2^m = 2^4 – 2^1 = 16 – 2 = 14$. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 2^4 – M X 2^1 .
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.



Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- rename registers A,B, and Q as AC,BR and QR respectively
- Q_n designates the least significant bit of the multiplier in register QR
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier



- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.
- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:

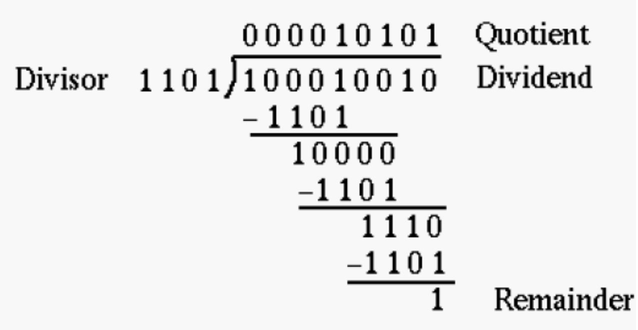
1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1’s in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 in a string of 0’s in the multiplier.
3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.

- The algorithm works for positive or negative multipliers in 2’s complement representation.
- This is because a negative multiplier ends with a string of 1’s and the last operation will be a subtraction of the appropriate weight.
- The two bits of the multiplier in Qn and Qn+1 are inspected.

- If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.
- When the two bits are equal, the partial product does not change.

5. Division Algorithms

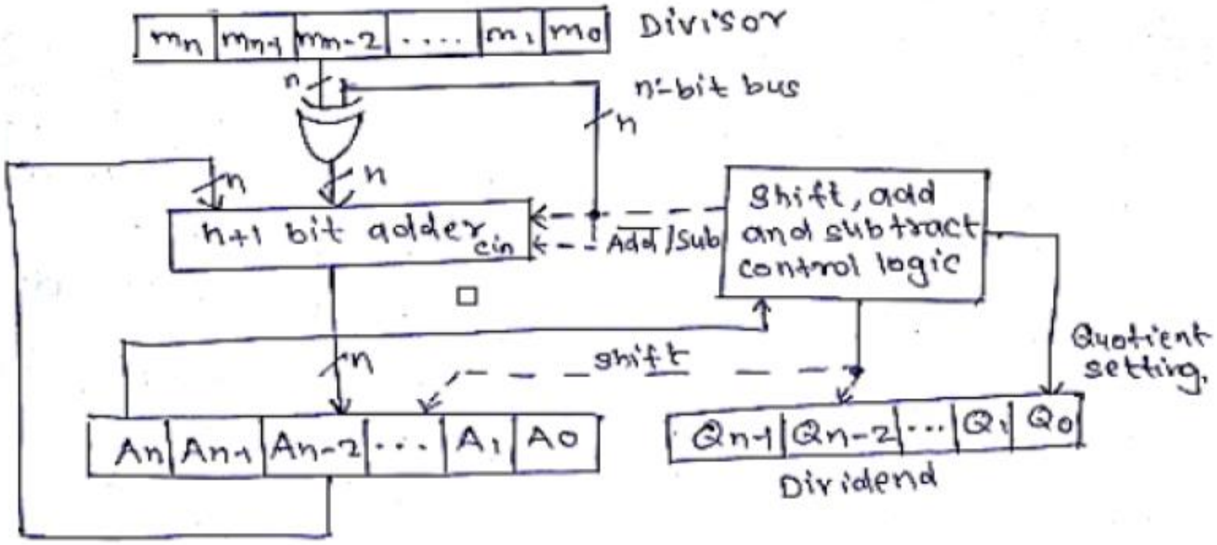
Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure



The divisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

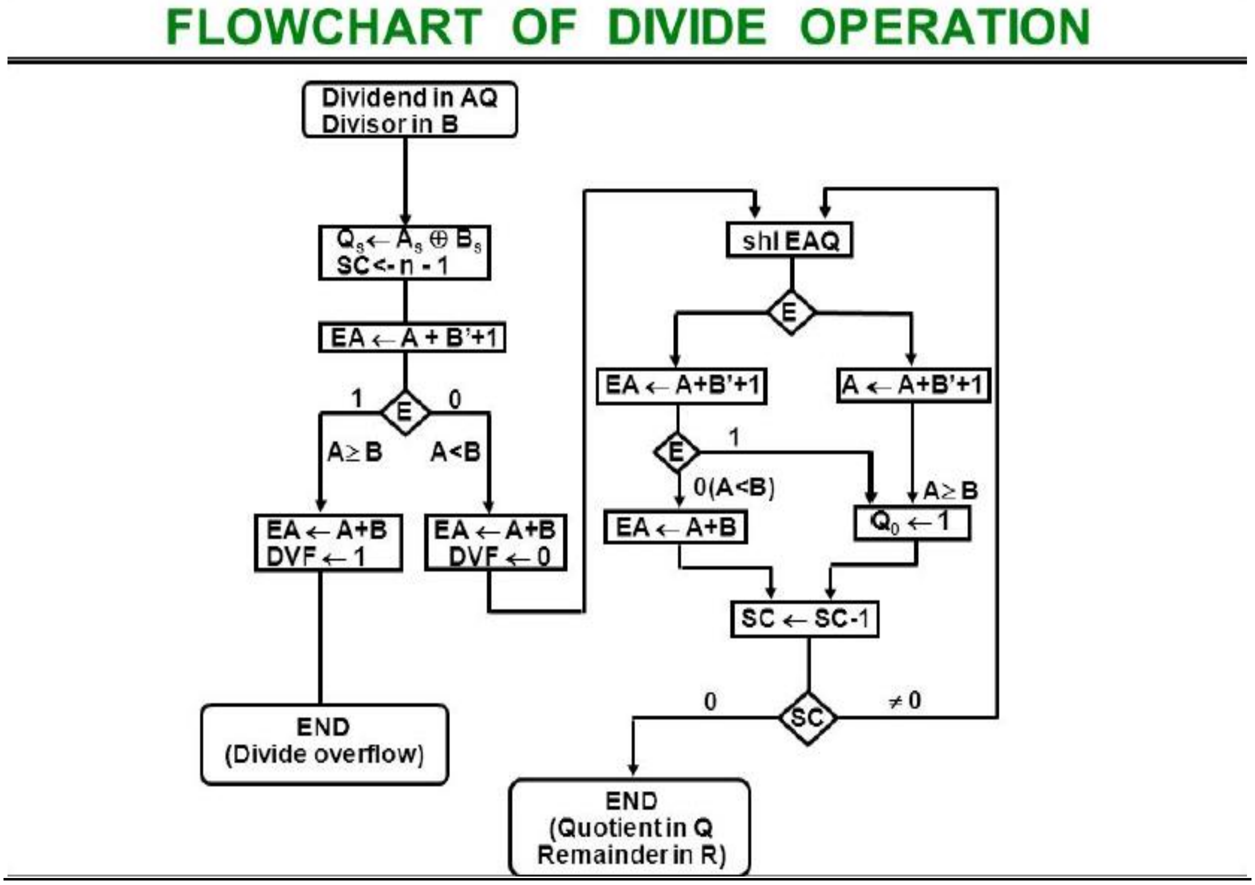
- **Hardware Implementation for Signed-Magnitude Data**

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes. The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Figure to clear the proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. E

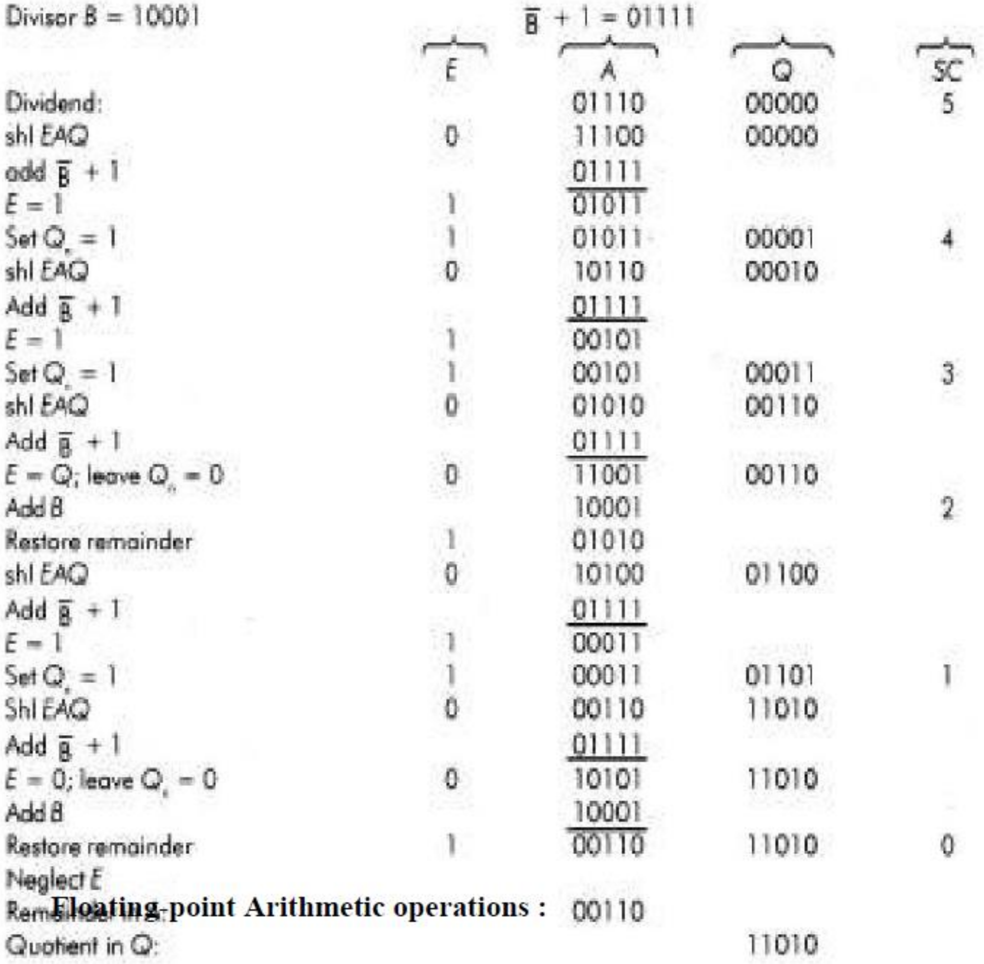


Hardware Implementation for Signed-Magnitude Data

Algorithm:



- **Example of Binary Division with Digital Hardware**



Divisor B = 10001 $\overline{B}$ + 1 = 01111

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B}$ + 1		01111		
E = 1	1	01011		
Set Q_n = 1	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B}$ + 1		01111		
E = 1	1	00101		
Set Q_n = 1	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B}$ + 1		01111		
E = Q; leave Q_n = 0	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B}$ + 1		01111		
E = 1	1	00011		
Set Q_n = 1	1	00011	01101	1
Shl EAQ	0	00110	11010	
Add $\overline{B}$ + 1		01111		
E = 0; leave Q_n = 0	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Floating-point Arithmetic operations :

Floating-point Arithmetic operations

In many high-level programming languages we have a facility for specifying floating-point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating- point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations:

There are two part of a floating-point number in a computer - a mantissa m and an exponent e.

The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus

$$\mathbf{m \times r^{e}}$$

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix r are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating-point number

$$\mathbf{.53725 \times 10^{3}}$$

A floating-point number is said to be normalized if the most significant digit of the mantissa in nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers for a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be + (24^7 – 1), which is approximately + 101^4. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

$$+ \mathbf{(1 - 2^{-35}) \times 2^{2047}}$$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because 2^{11}–1 = 2047. The largest number that can be accommodated is approximately 10615. The mantissa that can accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as (2^{35} –1). This is approximately equal to 1010, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas.

We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent. Consider the sum of the following floating-point numbers:

$$\mathbf{.5372400 \times 10^{2}}$$
$$\mathbf{+\ .1580000 \times 10^{-1}}$$

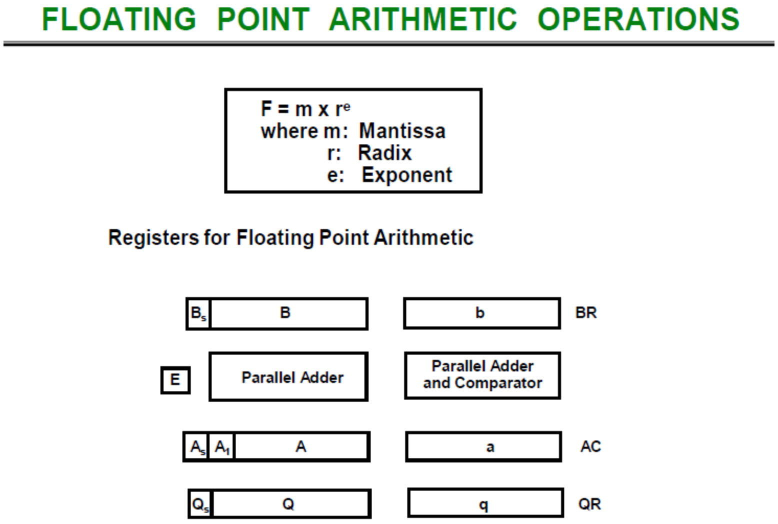
Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations – signed magnitude, signed 2's complement or signed 1's complement.

Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

6. Register Configuration

The register configuration for floating-point operations is shown in figure. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled. The register organization for floating-point operations is shown in Fig. Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.



FLOATING POINT ARITHMETIC OPERATIONS

$F = m \times r^e$
where m: Mantissa
r: Radix
e: Exponent

Registers for Floating Point Arithmetic

Fig: Registers for Floating Point Arithmetic Operations

- Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in As, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of As, A and a
- In the similar way, register BR is subdivided into Bs, B, and b and QR into Qs, Q and q. A parallel-adder adds the two mantissas and loads the sum into A and the carry into E. A separate parallel adder can be used for the exponents. The exponents do not have a district sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating point numbers are so large that the chance of an exponent overflow is very remote and so the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.
- The number in the mantissa will be taken as a fraction, so they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation
- The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

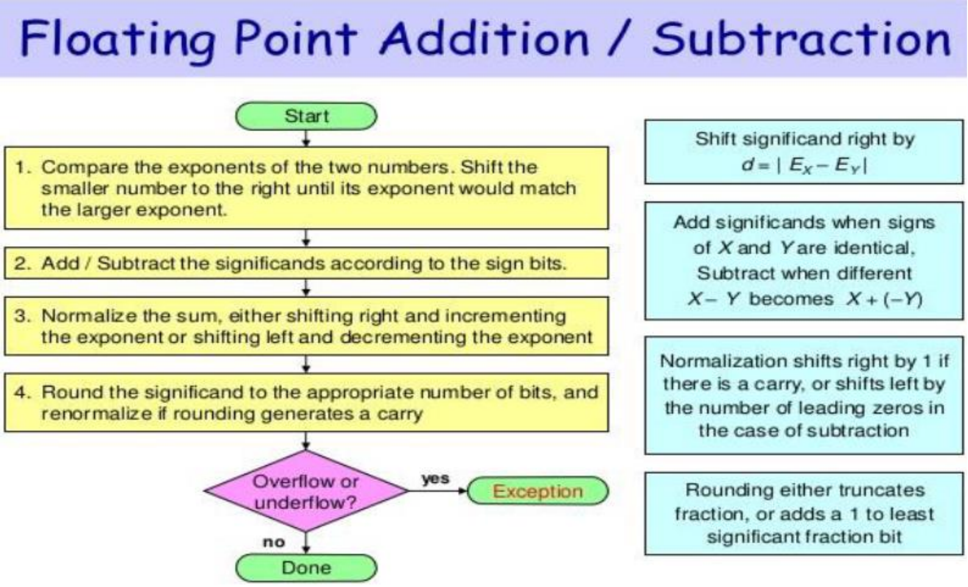
7. Addition and Subtraction of Floating Point Numbers

During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas

4. Normalize the result

- A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.
- If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A1, is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until A1 = 1. When A1 = 1, the mantissa is normalized and the operation is completed.



Floating Point Addition / Subtraction

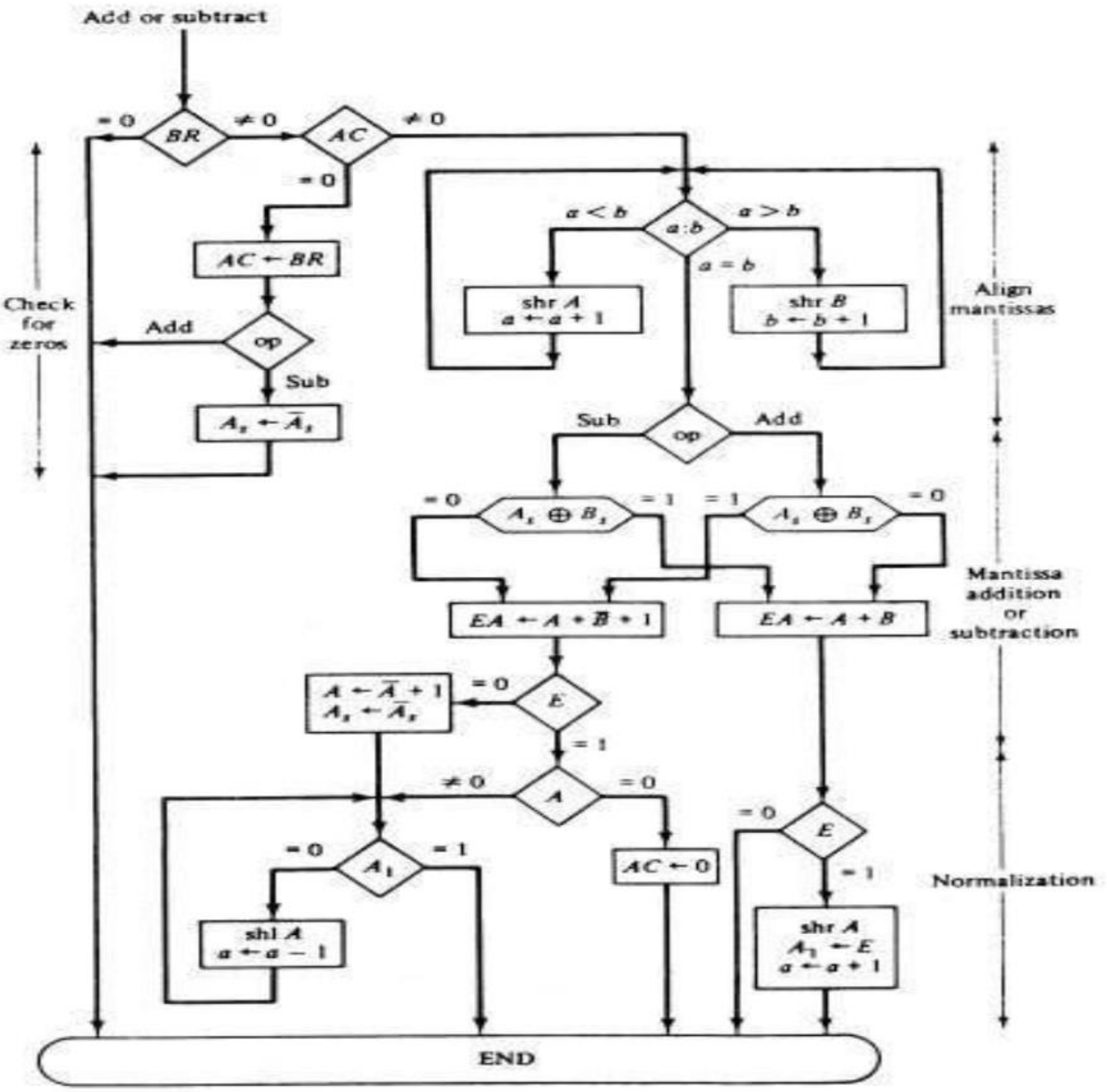
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.
2. Add / Subtract the significands according to the sign bits.
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry

Shift significand right by $d = | E_X - E_Y |$

Add significands when signs of X and Y are identical, Subtract when different $X - Y$ becomes $X + (-Y)$

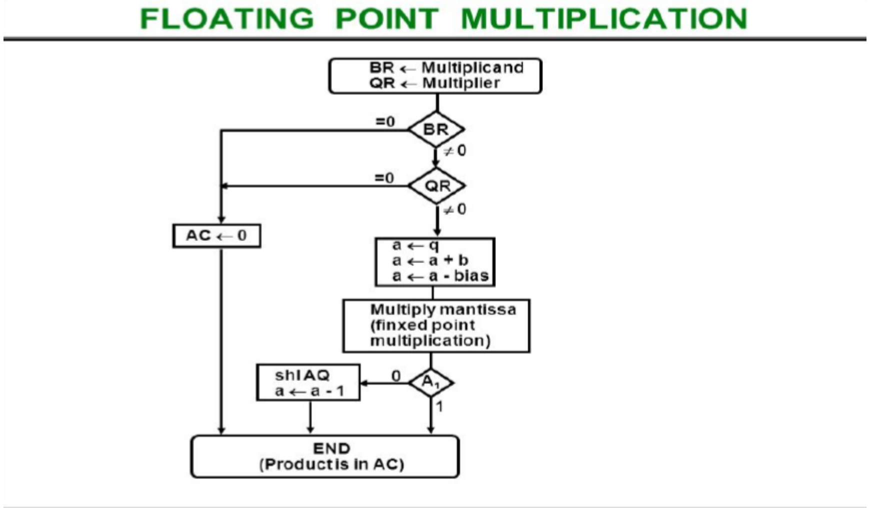
Normalization shifts right by 1 if there is a carry, or shifts left by the number of leading zeros in the case of subtraction

Rounding either truncates fraction, or adds a 1 to least significant fraction bit

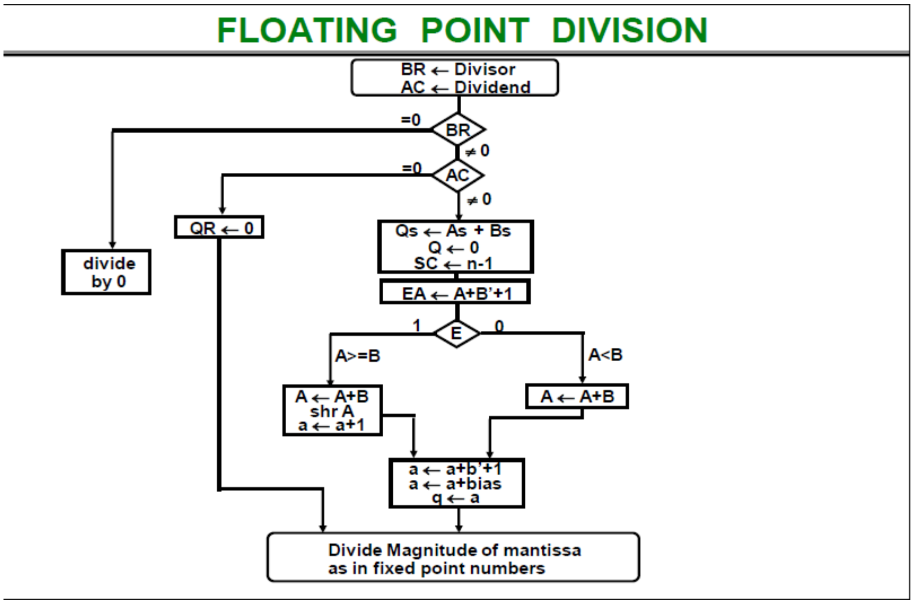


Algorithm for Floating Point Addition and Subtraction

Multiplication:



Floating Point Division



8. Input-Output Interface

- **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 the communication** link is to resolve the differences that exist between the central computer and each peripheral.
- **The major differences are:**

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.

3. Data codes and formats in peripherals differ from the word format in the CPU and memory.

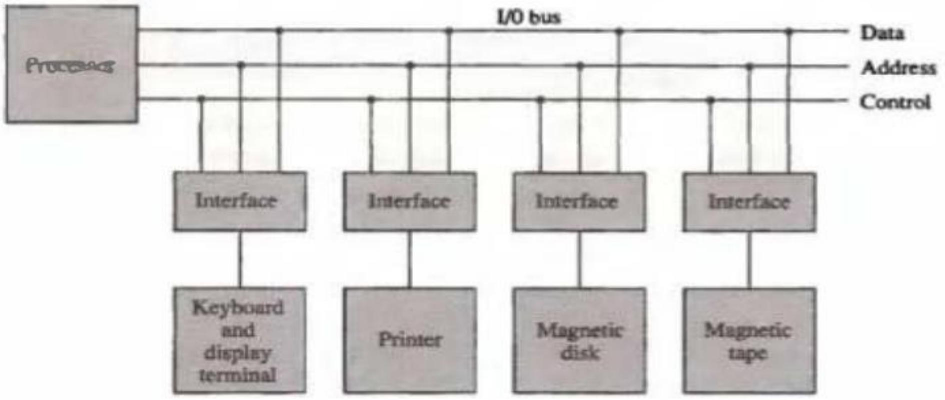
4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

- **To resolve these differences**, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called interface units because they interface between the processor bus and the peripheral device.
- **In addition**, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

9. I/O Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig. 1.

Figure 1 Connection of I/O bus to input-output devices



- **The I/O bus** consists of data lines, address lines, and control lines.
- **The magnetic disk**, printer, and terminal are employed in any general-purpose computer. The magnetic tape is used in some computers for backup storage.
- **Each peripheral device** has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller.
- **It also synchronizes** the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device.
- **For example, the** printer controller controls the paper motion, the print timing, and the selection of printing characters.
- **A controller** may be housed separately or may be physically integrated with the peripheral.

- **The I/O bus** from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines.
- **When the interface** detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.
- **At the same time** that the address is made available In the address lines, the processor provides a function code in the control lines.
- **The interface** selected responds to the function code and proceeds to execute it.
- **The function** code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing.
- **There are four types** of commands that an interface may receive. They are classified as control, status, data output, and data input.
- **A control command** is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction.
- **The particular control command** issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.
- **A status command** is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface.
- **These errors** are designated by setting bits in a status register that the processor can read at certain intervals.
- **A data output** command causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit.
- **The computer** starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command.
- **When the tape** is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.
- **The data input command** is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register.

- **The processor checks** if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

10. I/O versus Memory Bus

- **In addition** to communicating 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 three ways that computer buses can be used to communicate with memory and I/O:

- **Use two separate buses**, one for memory and the other for I/O.

- **Use one common bus** for both memory and I/O but have separate control lines for each.

- **Use one common bus** for memory and I/O with common control lines.

- **In the first method**, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O.

- **This is done** IOP in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU).

- **The memory communicates** with both the CPU and the IOP through a memory bus.

- **The IOP communicates** also with the input and output devices through a separate I/O bus with its own address, data and control lines.

- **The purpose of the IOP** is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a data channel.

11. Isolated versus Memory-Mapped I/O

- **Many computers** use one common bus to transfer information between memory or I/O and the CPU.

- **The distinction** between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines.

- **The I/O read and I/O write** control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer.

- **This configuration isolates** all I/O interface addresses from the addresses assigned to memory and is referred to as the isolated I/O method for assigning addresses in a common bus.

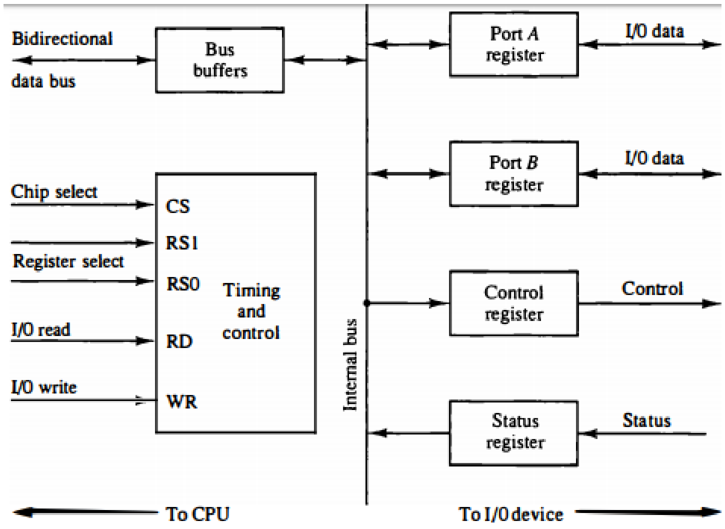
- **In the isolated I/O configuration**, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register.

- **When the CPU fetches** and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines.

- **At the same time**, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word.
- **On the other hand**, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line.
- **This informs** the external components that the address is for a memory word and not for an I/O interface.
- **The isolated** I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space.
- **The other alternative** is to use the same address space for both memory and I/O.
- **This is the case** in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses.
- **This configuration** is referred to as memory-mapped I/O. The computer treats an interface register as being part of the memory system.
- **The assigned addresses** for interface registers cannot be used for memory words, which reduces the memory address range available.
- **In a memory-mapped** I/O organization there are no specific input or output instructions.
- **The CPU can manipulate** I/O data residing in interface registers with the same instructions that are used to manipulate memory words.
- **Each interface** is organized as a set of registers that respond to read and write requests in the normal address space.
- **Typically, a segment** of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.
- **Computers with memory-mapped** I/O can use memory-type instructions to access I/O data.
- **It allows the computer** to use the same instructions for either input-output transfers or for memory transfers.
- **The advantage is that the load** and store instructions used for reading and writing from memory can be used to input and output data from I/O registers.
- **In a typical computer**, there are more memory-reference instructions than I/O instructions. With memory mapped I/O all instructions that refer to memory are also available for I/O.

➢ **Example of I/O Interface**

- **An example** of an I/O interface unit is shown in block diagram form in Fig.



CS	RS1	RS0	Register selected
0	×	×	None: data bus in high–impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

Figure ·2 Example of I/O interface unit.

- **It consists** of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits.
- **The interface** communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface.
- **The I/O read** and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.
- **The I/O data** to and from the device can be transferred into either port A or port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output.
- **If the interface** is connected to a printer, it will only output data, and if it services a character reader, it will only input data.
- **A magnetic disk** unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines.
- **A command** is passed to the I/O device by sending a word to the appropriate interface register.

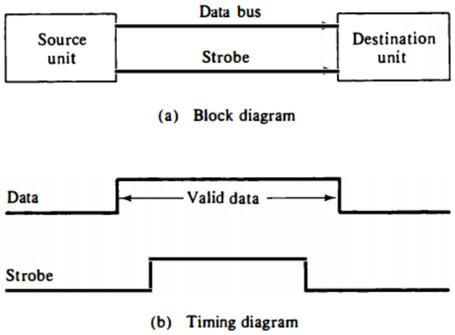
- **In a system like this**, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers.
- **Thus the transfer of data**, control, and status information is always via the common data bus.
- **The distinction between data**, control, or status information is determined from the particular interface register with which the CPU communicates.
- **The control register** receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes.
- **For example**, port A may be defined as an input port and port B as an output port.
- **A magnetic** tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction.
- **The bits** in the status register are used for status conditions and for recording errors that may occur during the data transfer.
- **For example**, a status bit may indicate that port A has received a new data item from the I/O device.
- **Another bit** in the status register may indicate that a parity error has occurred during the transfer.
- **The interface** registers communicate with the CPU through the bidirectional data bus.
- **The address bus** selects the interface unit through the chip select and the two register select inputs.
- **A circuit must** be provided externally (usually, a decoder) to detect the address assigned to the interface registers.
- **This circuit enables** the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the address bus.
- **These two inputs** select one of the four registers in the interface as specified in the table accompanying the diagram.
- **The content** of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled.
- **The CPU transfers** binary information into the selected register via the data bus when the I/O write input is enabled.

12. Asynchronous Data Transfer

- **The internal operations** in a digital system are synchronized by means of clock pulses supplied by a common pulse generator.
- **Clock pulses** are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse.
- **Two units**, such as a CPU and an I/O interface, are designed independently of each other.
- **If the registers** in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous.
- **In most cases**, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.
- **Asynchronous data transfer** between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted.
- **One way of achieving** this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
- **Another method** commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus.
- **The unit receiving** the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as handshaking.
- **The strobe pulse method** and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units.
- **In the general** case we consider the transmitting unit as the source and the receiving unit as the destination.
- **For example**, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer.
- **It is customary** to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in the buses.
- **The sequence of control** during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

i. Strobe Control

- **The strobe control** method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure 3(a) shows a source-initiated transfer.



(a) Block diagram

(b) Timing diagram

Figure 3 Source-initiated strobe for data transfer.

- **The data bus** carries the binary information from source unit to the destination unit.
- **Typically**, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.
- **As shown** in the timing diagram of Fig. 3(b), the source unit first places the data on the data bus.
- **After a brief delay** to ensure that the data settle to a steady value, the source activates the strobe pulse.
- **The information** on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data.
- **Often, the destination** unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers.
- **The source removes** the data from the bus a brief period after it disables its strobe pulse.
- **Actually**, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valid data. New valid data will be available only after the strobe is enabled again.
- **Figure 4** shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data.
- **The source unit** responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can be used again to trigger a destination register
- **The destination unit** then disables the strobe. The source removes the data from the bus after a predetermined time interval.
- **In many computers** the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external units how to transfer data.
- **For example**, the strobe of Fig. 3 could be a memory-write control signal from the CPU to a memory unit. The source, being the CPU, places a word on the data bus and informs the memory unit, which is the destination, that this is a write operation.

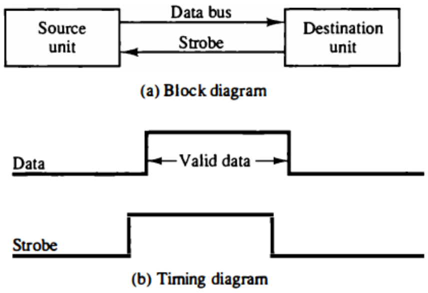
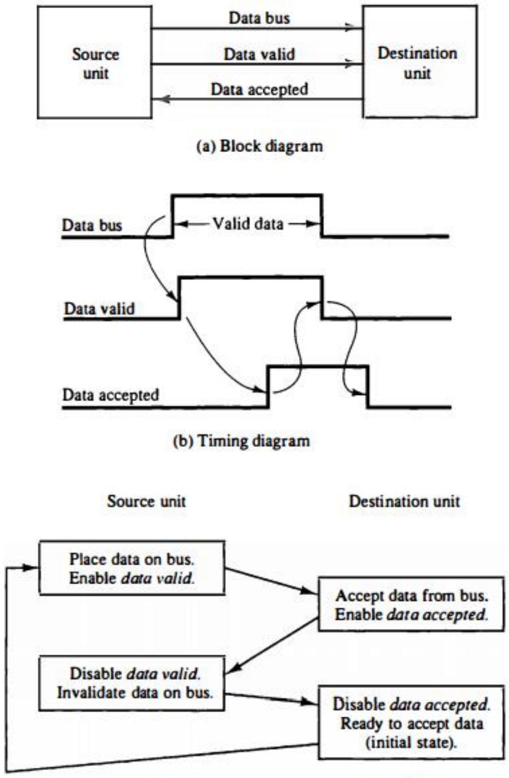


Figure 4 Destination-initiated strobe for data transfer.

- **Similarly**, the strobe of Fig. 4 could be a memory-read control signal from the CPU to a memory unit.
- **The destination**, the CPU, initiates the read operation to inform the memory, which is the source, to place a selected word into the data bus.
- **The transfer of data** between the CPU and an interface unit is similar to the strobe transfer just described.
- **Data transfer** between an interface and an I/O device is commonly controlled by a set of handshaking lines.

ii. Handshaking

- **The disadvantage** of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus.
- **Similarly**, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus.
- **The handshake** method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer.
- **The basic principle** of the two-wire handshaking method of data transfer is as follows.
- **One control line** is in the same direction as the data flow in the bus from the source to the destination.
- **It is used by the source** unit to inform the destination unit whether there are valid data in the bus.
- **The other control line** is in the other direction from the destination to the source.
- **It is used by the destination unit** to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.
- **Figure 5 shows the data transfer** procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit.
- **The timing diagram shows** the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time.


Source unit
Data bus
Data valid
Data accepted
Destination unit
Data bus
Valid data
Data valid
Data accepted
Source unit
Destination unit
Place data on bus. Enable *data valid.*
Accept data from bus. Enable *data accepted.*
Disable *data valid.* Invalidate data on bus.
Disable *data accepted.* Ready to accept data (initial state).

(a) Block diagram

(b) Timing diagram

Figure ·5 Source-initiated transfer using handshaking.

- **The source unit** initiates the transfer by placing the data on the bus and enabling its data valid signal.
- **The data accepted signal** is activated by the destination unit after it accepts the data from the bus.
- **The source unit then disables its data valid signal,** which invalidates the data on the bus.
- **The destination unit then disables** its data accepted signal and the system goes into its initial state.
- **The source does not** send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal.
- **This scheme** allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.
- **The destination-initiated** transfer using handshaking lines is shown in Fig. 6. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.
- **The source unit** in this case does not place data on the bus until after it receives the ready for data signal from the destination unit.
- **From there on**, the handshaking procedure follows the same pattern as in the source-initiated case.
- **Note that the sequence of events** in both cases would be identical if we consider the ready for data signal as the complement of data accepted.

- **In fact, the only difference** between the source-initiated and the destination-initiated transfer is in their choice of initial state.

- **The handshaking scheme** provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed.

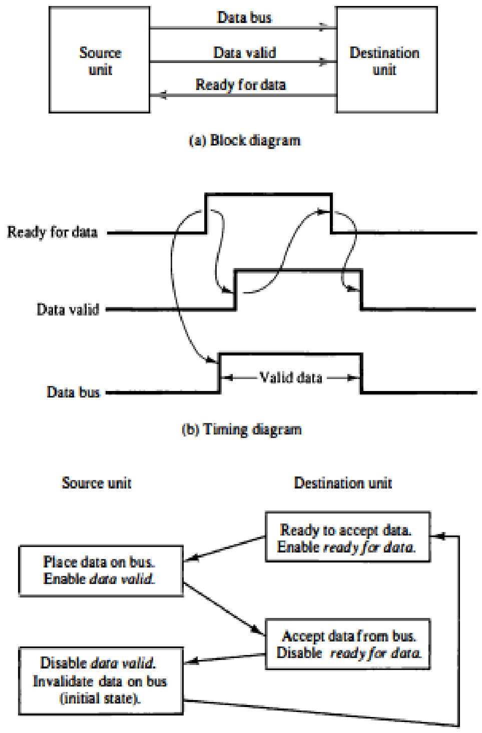
- **Such an error** can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time.

- **The timeout** is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals.

- **If the return** handshake signal does not respond within a given time period, the unit assumes that an error has occurred.

- **The timeout signal** can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action

Figure ·6 Destination-initiated transfer using handshaking.


Source unit
Data bus
Data valid
Ready for data
Destination unit
Ready for data
Data valid
Data bus
Valid data
Source unit
Destination unit
Ready to accept data. Enable *ready for data.*
Place data on bus. Enable *data valid.*
Accept data from bus. Disable *ready for data.*
Disable *data valid.* Invalidate data on bus (initial state).

(a) Block diagram

(b) Timing diagram

(c) Sequence of events

13. Asynchronous Serial Transfer

- **The transfer of data** between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time.

- **This means that an n-bit message** must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time.

- **This method** requires the use of one pair of conductors or one conductor and a common ground.

- **Parallel transmission** is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.
- **Serial transmission** can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses.
- **In long distant** serial transmission, each unit is driven by a separate clock of the same frequency.
- **Synchronization signals** are transmitted periodically between the two units to keep their clocks in step with each other. In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted.
- **This is in contrast** to synchronous transmission, where bits must be transmitted continuously to keep the clock frequency in both units synchronized with each other.
- **A serial asynchronous** data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code.
- **With this technique**, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character.
- **The last bit called the stop bit** is always a 1. An example of this format is shown in Fig. 7.
- **A transmitted character** can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.

2. The initiation of a character transmission is detected from the start bit, which is always 0.

3. The character bits always follow the start bit.

4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

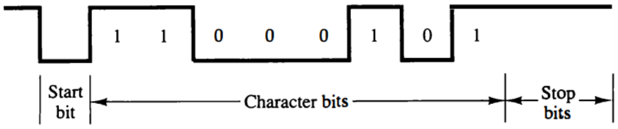
Using these rules, the receiver can detect the start bit when the line goes from 1 to 0.

- **A clock in the receiver** examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent.
- **The stop bits are always** in the 1-state and frame the end of the character to signify the idle or wait state.
- **At the end of the character** the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize.
- **The length of time that** the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted.

- **The stop time ensures** that a new character will not follow for one or two bit times.

- **As an illustration**, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists **of a start bit**, eight information bits, and two stop bits, for a total of 11 bits.

Figure 7 Asynchronous serial transmission.



- **Ten characters per** second means that each character takes 0.1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms.

- **The baud rate is defined** as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

- **The terminal has** a keyboard and a printer. Every time a key is depressed, the terminal sends 11 bits serially along a wire. To print a character in the printer, an 11-bit message must be received along another wire. The terminal interface consists of a transmitter and a receiver.

- **The transmitter accepts** an 8-bit character from the computer and proceeds to send a serial 11-bit message into the printer line. The receiver accepts a serial 11-bit message from the keyboard line and forwards the 8-bit character code into the computer.

- **Integrated circuits** are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver transmitter (UART).

14. Modes of Transfer

- **Binary information** received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit.

- **The CPU merely** executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit.

- **Data transfer** between the central computer and I/O devices may be handled in a variety of modes.

- **Some modes** use the CPU as an intermediate path; others transfer the data directly to and from the memory unit.

- **Data transfer** to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

- **Programmed I/O** operations are the result of I/O instructions written in the computer program.

- **Each data item** transfer is initiated by an instruction in the program.
- **Usually, the transfer** is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU.
- **Once a data transfer** is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.
- **In the programmed** I/O method, 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 since it keeps the processor busy needlessly.
- **It can be avoided** by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program.
- **The interface** meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.
- **Transfer of data under** programmed I/O is between CPU and peripheral.
- **In direct memory access (DMA)**, the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks.
- **When the transfer is made**, the DMA requests memory cycles through the memory bus.
- **When the request** is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer.
- **Since peripheral speed** is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

15. Example of Programmed I/O

- **In the programmed I/O method**, the I/O device does not have direct access to memory.
- **A transfer from an I/O** device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory.
- **Other instructions** may be needed to verify that the data are available from the device and to count the numbers of words transferred.
- **An example of data transfer** from an I/O device through an interface into the CPU is shown in Fig. 10. The device transfers bytes of data one at a time as they are available.
- **When a byte of data** is available, the device places it in the I/O bus and enables its data valid line.
- **The interface accepts** the byte into its data register and enables the data accepted line.

- **The interface sets** a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.
- **This is according** to the handshaking procedure established in Fig. 5.
- **A program is written** for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device.
- **This is done by reading** the status register into a CPU register and checking the value of the flag bit.
- **If the flag is equal** to 1, the CPU reads the data from the data register.
- **The flag bit** is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed.
- **Once the flag is cleared**, the interface disables the data accepted line and the device can then transfer the next data byte.
- **A flowchart of the program** that must be written for the CPU is shown in Fig. 11. It is assumed that the device is sending a sequence of bytes that must be stored in memory.
- **The transfer of each byte requires three instructions:**

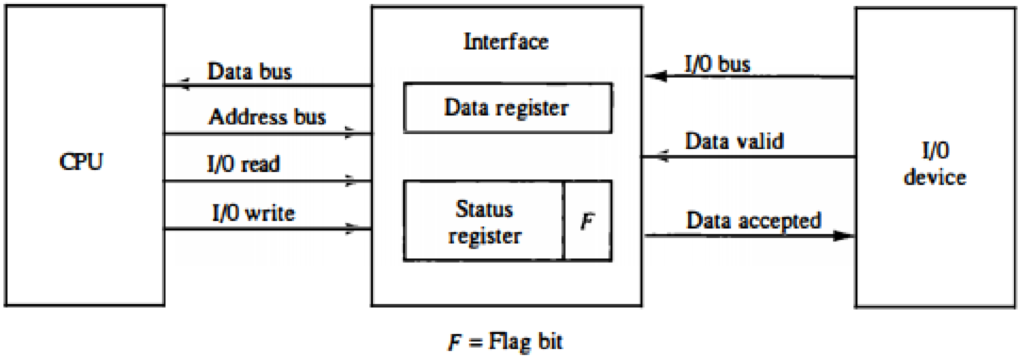
1. Read the status register.

2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.

3. Read the data register.

- **Each byte** is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer.

Figure 10 Data transfer from I/O device to CPU.



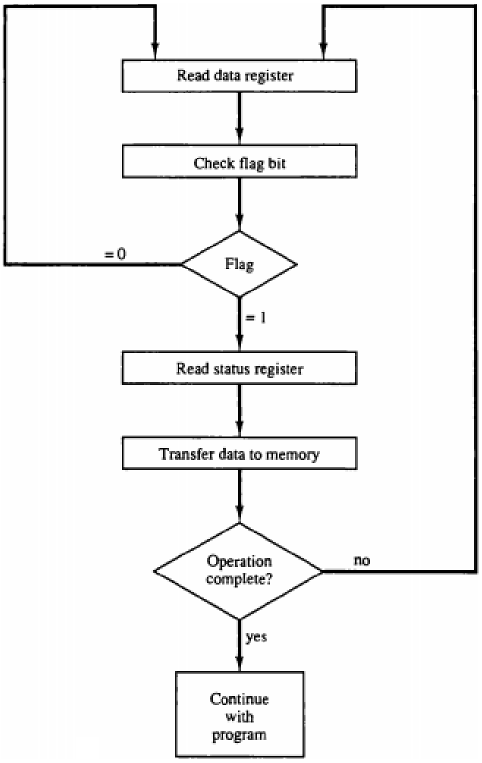


Figure ·11 Flowchart for CPU program to input data.

- **The programmed** I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously.
- **The difference** in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient.
- **To see why this** is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 µS.
- **Assume that** the input device transfers its data at an average rate of 100 bytes per second.
- **This is equivalent to one byte** every 10,000 µS.
- **This means** that the CPU will check the flag 10,000 times between each transfer.
- **The CPU is wasting** time while checking the flag instead of doing some other useful processing task.

16. Interrupt-Initiated I/O

- **An alternative** to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data.
- **This mode of transfer** uses the interrupt facility. While the CPU is running a program, it does not check the flag.
- **However**, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.
- **The CPU deviates** from what it is doing to take care of the input or output transfer.

- **After the transfer** is completed, the computer returns to the previous program to continue what it was doing before the interrupt.
- **The CPU responds** to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.
- **The way that the processor chooses** the branch address of the service routine varies from one unit to another.
- **In principle**, there are two methods for accomplishing this.
- **One is called vectored** interrupt and the other, non vectored interrupt. In a non vectored interrupt, the branch address is assigned to a fixed location in memory.
- **In a vectored interrupt**, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector.
- **In some computers the interrupt vector** is the first address of the I/O service routine.
- **In other computers the interrupt vector** is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

17. Priority Interrupt

- **Data transfer** between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU.
- **The readiness** of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer.
- **Some processors** also push the current PSW (program status word) onto the stack and load a new PSW for the service routine.
- **In a typical application** a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt.
- **There is also the possibility** that several sources will request service simultaneously. In this case the system must also decide which device to service first.
- **A priority interrupt** is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously.
- **The system may also** determine which conditions are permitted to interrupt the computer while another interrupt is being serviced.
- **Higher-priority interrupt** levels are assigned to requests which, if delayed or interrupted, could have serious consequences.
- **Devices with high speed** transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority.
- **When two devices** interrupt the computer at the same time, the computer services the device, with the higher priority first.
- **Establishing the priority** of simultaneous interrupts can be done by software or hardware.

- **A polling procedure** is used to identify the highest-priority source by software means.
- **In this method** there is one common branch address for all interrupts.
- **The program** that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt.
- **The highest-priority** source is tested first, and if its interrupt signal is on, control branches to a service routine for this source.
- **Otherwise**, the next-lower-priority source is tested, and so on. Thus the initial service routine for all interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines.
- **The particular** service routine reached belongs to the highest-priority device among all devices that interrupted the computer.
- **The disadvantage** of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt unit can be used to speed up the operation.
- **A hardware priority-interrupt** unit functions as an overall manager in an interrupt system environment.
- **It accepts interrupt** requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination.
- **To speed up the operation**, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit.
- **The hardware priority function** can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

18. Daisy-Chaining Priority

- **The daisy-chaining** method of establishing priority consists of a serial connection of all devices that request an interrupt.
- **The device** with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain.
- **This method** of connection between three devices and the CPU is shown in Fig. 12. The interrupt request line is common to all devices and forms a wired logic connection.
- **If any device** has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU.
- **When no interrupts** are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.
- **This is equivalent** to a negative logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input.

- **The acknowledge** signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt.

- **If device 1 has a pending interrupt**, it blocks the acknowledge signal from the next device by placing a 0 in the PO output.
- **It then proceeds** to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.
- **A device** with a 0 in its PI input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked.
- **A device** that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output.
- **If the device** does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output.

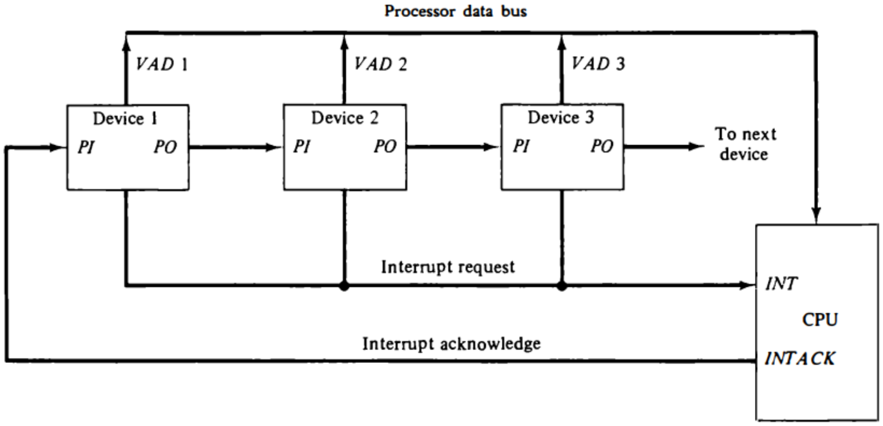


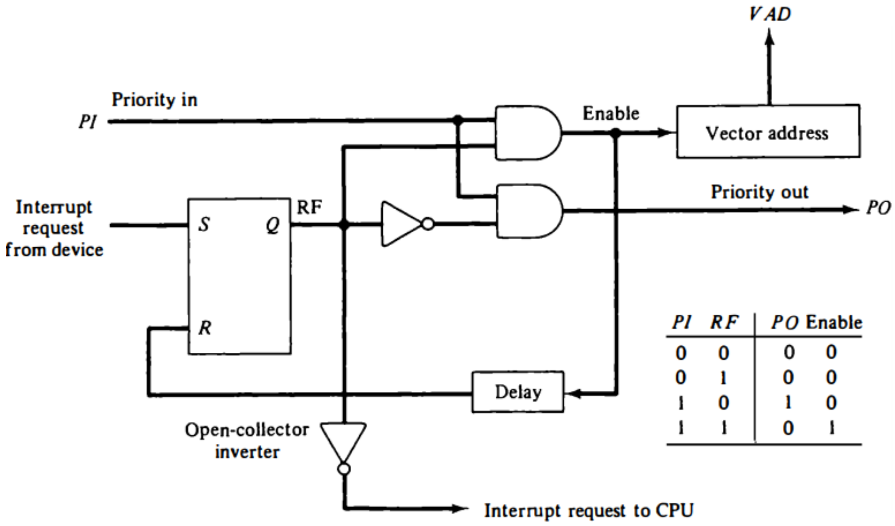
Figure 12 Daisy-chain priority interrupt.

- **Thus the device** with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus.
- **The daisy chain** arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU.
- **The farther** the device is from the first position; the lower is its priority.
- **Figure 13** shows the internal logic that must be included within each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU.
- **The output of the RF flip-flop** goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line.
- **If PI = 0, both PO** and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled.
- **This condition** passes the acknowledge signal to the next device through PO.
- **The device** is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus.

- **It is assumed** that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

19. Parallel Priority Interrupt

- **The parallel priority** interrupt method uses a register whose bits are set separately by the interrupt signal from each device.
- **Priority** is established according to the position of the bits in the register.
- **In addition** to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request.



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

Figure 13 One stage of the daisy-chain priority arrangement.

- **The mask register** can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced.
- **It can also provide** a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.
- **The priority logic** for a system of four interrupt sources is shown in Fig. 14.
- **It consists** of an interrupt register whose individual bits are set by external conditions and cleared by program instructions.
- **The magnetic disk,** being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard.
- **The mask register** has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register.
- **Each interrupt bit** and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder.
- **In this way** an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.
- **Another output** from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system.

- **The outputs** of IST ANDed with IEN provide a common interrupt signal for the CPU.

- **The interrupt** acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus.

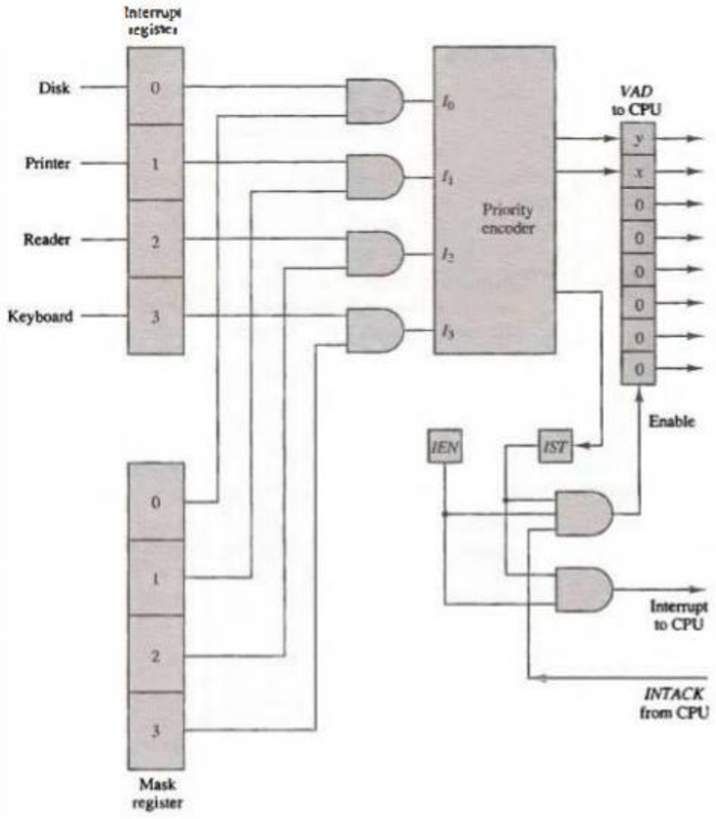


Figure : 14 Priority interrupt hardware.

20. Direct Memory Access (DMA)

- **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

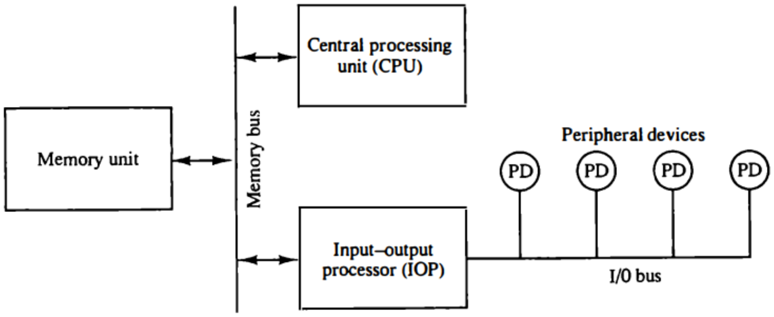


Figure : 19 Block diagram of a computer with I/O processor.

- **The data formats** of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources.

- **For example,** it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory.

- **Data are gathered** in the IOP at the device rate and bit capacity while the CPU is executing its own program.

- **After the input** data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU.

- **Similarly**, an output word transferred from memory to the lOP is directed from the IOP to the output device at the device rate and bit capacity.

- **The communication** between the IOP and the devices attached to it is similar to the program control method of transfer. Communication with the memory is similar to the direct memory access method.

- **The way by which the CPU and IOP** communicate depends on the level of sophistication included in the system.

- **In very-large-scale** computers, each processor is independent of all others and any one processor can initiate an operation. In most computer systems, the CPU is the master while the IOP is a slave processor.

- **The CPU is assigned** the task of initiating all operations, but 110 instructions are executed in the IOP.

- **CPU instructions** provide operations to start an 110 transfer and also to test 110 status conditions needed for making decisions on various 110 activities.

- **The IOP**, in turn, typically asks for CPU attention by means of an interrupt.

- **It also responds** to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program.

- **When an 110 operation** is desired, the CPU informs the IOP where to find the 110 program and then leaves the transfer details to the IOP.

- **Instructions** that are read from memory by an IOP are sometimes called commands, to distinguish them from instructions that are read by the CPU. Otherwise, an instruction and a command have similar functions.

- **Commands** are prepared by experienced programmers and are stored in memory. The command words constitute the program for the IOP.

- **The CPU** informs the IOP where to find the commands in memory when it is time to execute the 110 program.

21. CPU-IOP Communication

- **The communication** between CPU and IOP may take different forms, depending on the particular computer considered.

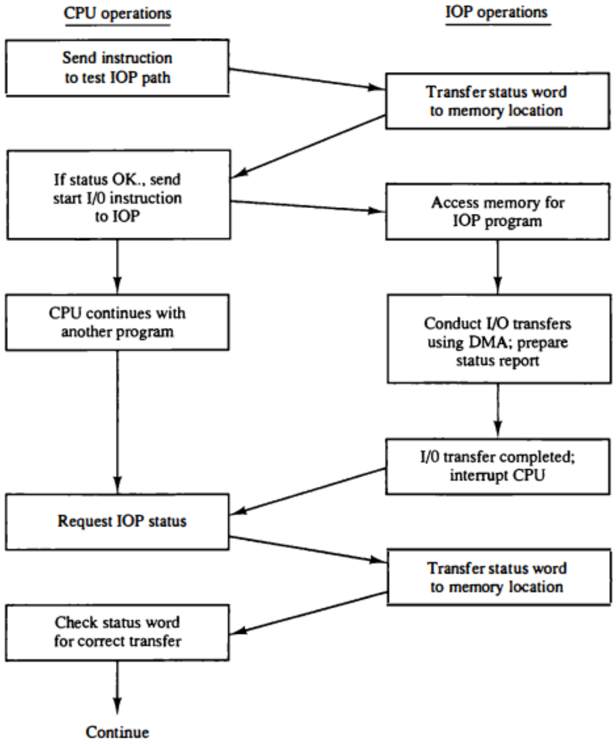
- **In most cases** the memory unit acts as a message center where each processor leaves information for the other.

- **To appreciate the operation** of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate.

- **This is a simplified** example that omits many operating details in order to provide an overview of basic concepts.

- **The sequence** of operations may be carried out as shown in the flowchart of Fig. 20.
- **The CPU** sends an instruction to test the IOP path.

Figure 20 CPU-IOP communication.


CPU operations
IOP operations
Send instruction to test IOP path
Transfer status word to memory location
If status OK., send start I/0 instruction to IOP
Access memory for IOP program
CPU continues with another program
Conduct I/O transfers using DMA; prepare status report
I/0 transfer completed; interrupt CPU
Request IOP status
Transfer status word to memory location
Check status word for correct transfer
Continue

- **The IOP responds** by inserting a status word in memory for the CPU to check.
- **The bits of the status word** indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer.
- **The CPU refers** to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer.
- **The memory address** received with this instruction tells the IOP where to find its program.
- **The CPU can now continue** with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer.
- **When the IOP terminates** the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP.
- **The IOP responds** by placing the contents of its status report into a specified memory location. The status word indicates whether the transfer has been completed or if any errors occurred during the transfer.
- **From inspection** of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily without errors.
- **The IOP takes** care of all data transfers between several I/O units and the memory while the CPU is processing another program.

- **The IOP and CPU** are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.
- **It is not possible** to saturate the memory by I/O devices in most systems, as the speed of most devices is much slower than the CPU.
- **However**, some very fast units, such as magnetic disks, can use an appreciable number of the available memory cycles.
- **In that case**, the speed of the CPU may deteriorate because it will often have to wait for the IOP to conduct memory transfers.