LECTURER
NOTES ON
COMPUTER SYSTEM ARCHITECTURE

PREFERRED BY:ER.RAJKUMAR MISHRA

DIPLOMA 3RD SEM

(2023-24)

DEPARTMENT OF COMPUTER SCIENCE

GANESH INSTITUTE OF ENGINEERING AND TECHNOLOGY
(POLYTECHNIC)
(Approved by AICTE)
ANDHARUA,BHUBANESWAR

(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.

1. Basic structure of computer hardware

1.1Basic Structure of computer hardware
1.2Functional Units
1.3Computer components
1.4 Performance measures
1.5 Memory addressing & Operations

2. Instructions & instruction Sequencing

2.1 Fundamentals to instructions
2.2 Operands
2.3 Op Codes
2.4 Instruction formats
2.5 Addressing Modes

3. Processor System

3.1 Register Files
3.2 Complete instruction execution
- Fetch
- Decode
- Execution

3.3 Hardware control
3.4 Micro program control

4. Memory System

4.1Memory characteristics
4.2 Memory hierarchy
4.3 RAM and ROM organization
4.4 Interleaved Memory
4.5 Cache memory
4.6 Virtual memory

5. Input – Output System

5.1 Input - Output Interface
5.2 Modes of Data transfer
5.3 Programmed I/O Transfer
5.4 Interrupt driven I/O
5.5 DMA
5.6 I/O Processor

6. I/O Interface & Bus architecture

6.1 Bus and System Bus
6.2 Types of System Bus
- Data
- Address
- Control

6.1 Bus
6.3 Bus Structure
6.4 Basic Parameters of Bus design
6.5 SCSI
6.6 USB

7. Parallel Processing

7.1 Parallel Processing
7.2 Linear Pipeline
7.3 Multiprocessor
7.4 Flynn‟s Classification

INDEX

S. No	Unit	Topic	Page no
1	I	Functional Units of computer , Data Representation – integer , floating point, character	1
2	I	Computer Arithmetic: Addition, Subtraction, Multiplication and Division	5
3	II	X86 architecture	10
4	II	Register Transfer language	13
5	II	Hardwired Programmed Control	18
6	III	Semiconductor Memory Technologies	20
7	III	Memory Interleaving	22
8	III	Cache Memories	24
9	IV	I/O subsystems	28
10	IV	I/O Interrupts	29
11	IV	I/O Device Interfaces	34
12	V	Basic concepts of Pipelining	37
13	V	Pipeline Hazards	38
14	V	Introduction to Parallel processors	41
15	V	Concurrent access to memory and Cache Coherence	42

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.

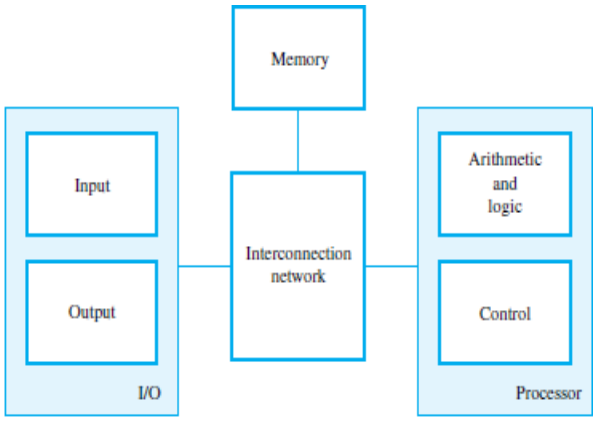


Figure 1.1 Basic functional units of a computer.

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	Values represented		
$b_3b_2b_1b_0$	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

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

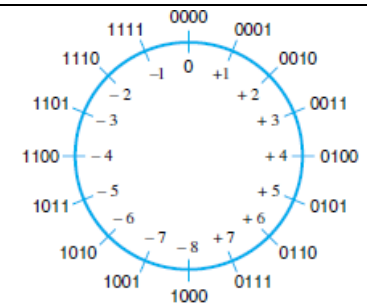
The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the *sum* is 0 and the *carry-out* is 1. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the *carry-in*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

	0010	(+2)	(b) 0100	(+4)
	+ 0011	(+3)	+ 1010	(−6)
	0101	(+5)	1110	(−2)
	1011	(−5)	(d) 0111	(+7)
	+ 1110	(−2)	+ 1101	(−3)
	1001	(−7)	0100	(+4)

Floating-Point Numbers

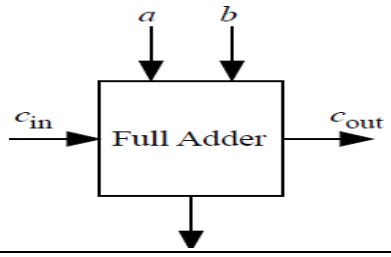
If we use a full word in a 32-bit word length computer to represent a signed integer in 2"s-complement representation, the range of values that can be represented is -2^{31} to $+2^{31} - 1$.
Since the position of the binary point in a floating-point number varies, it must be indicated explicitly in 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	Bit positions 654							
3210	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.

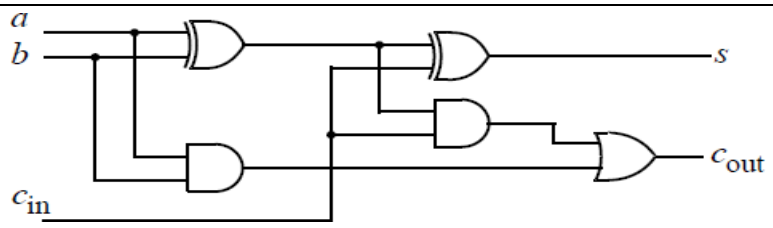


A one-bit full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs(a,b, and cin) and two outputs(s, and cout)____as illustrated in Figure 1.

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

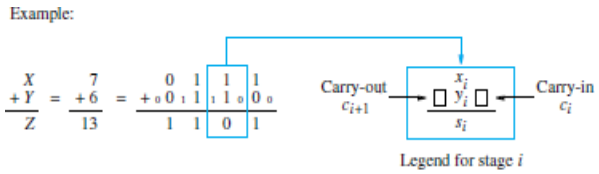


Figure 9.1 Logic specification for a stage of binary addition.

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

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

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

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 ai and bi are 1, or

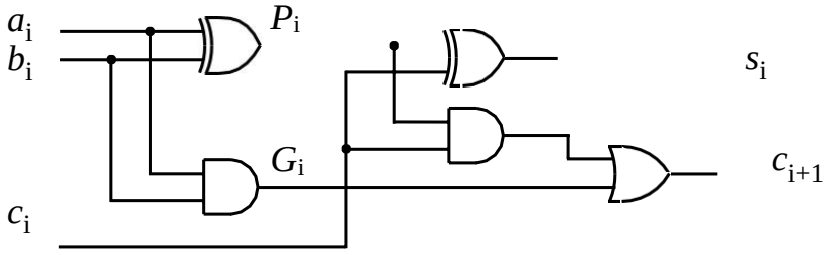
(2) when one of the two bits is 1 and the carry-in is 1 .

Thus, one can write,

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

$$s_i = (a_i \oplus b_i) \oplus c_i \quad (4)$$

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.

$$c_{i+1} = G_i + P_i.c_i \quad (5)$$

$$s_i = P_i \oplus c_i \quad (6)$$

$$G_i = a_i.b_i \quad (7)$$

$$P_i = a_i \oplus b_i \quad (8)$$

(9)

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

$$c_1 = G_0 + P_0.c_0 \quad (10)$$

$$c_2 = G_1 + P_1.G_0 + P_1.P_0.c_0 \quad (11)$$

$$c_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \quad (12)$$

$$c_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0 \quad (13)$$

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.

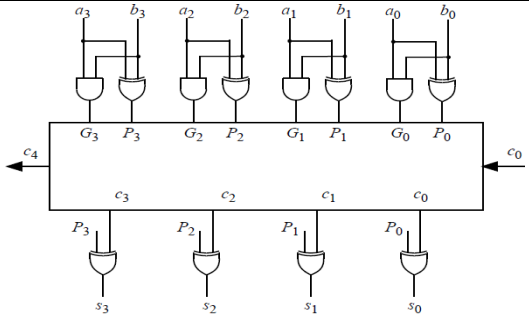
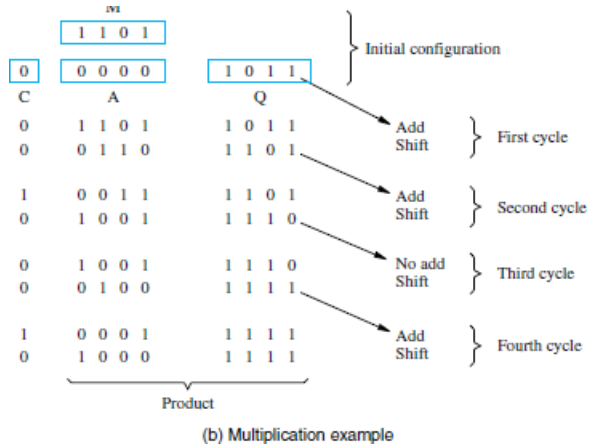


Figure 5: 4-Bit carry lookahead adder implementation detail.

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.

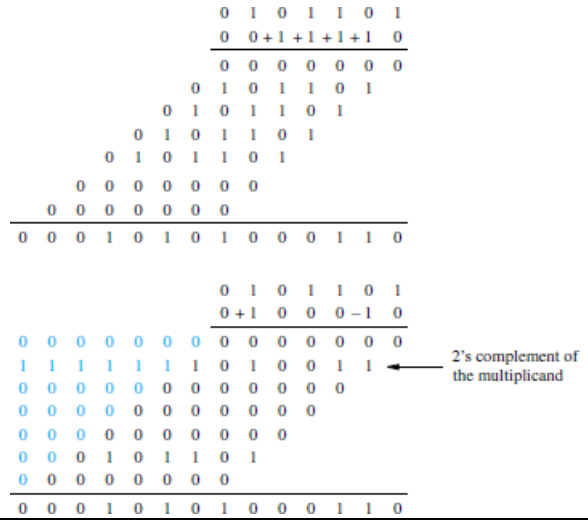
Consider the above figure in which the multiplier and multiplicand values are given as 1011 and 1101 which are loaded into the Q and A registers respectively.

Initially the register C is zero and hence the A register is zero, which stores the carry in addition.

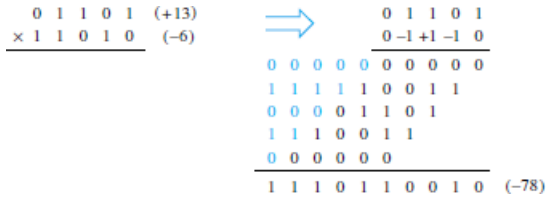
Since the B0 =1, then the number in the B is added to the bits of A and produce the addition result as 1101, and the Q and A register are shifted their values one bit right so the new values during the first cycle are 0110 and 1101 respectively.

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

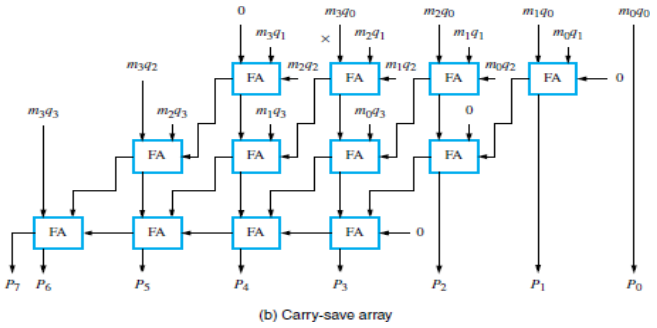
Booth Multiplier



The Booth algorithm generates a 2*n*-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



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

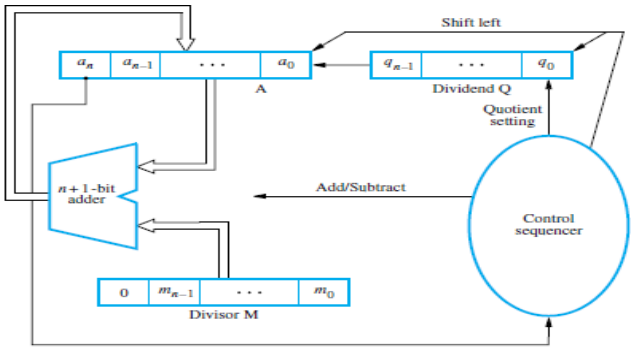
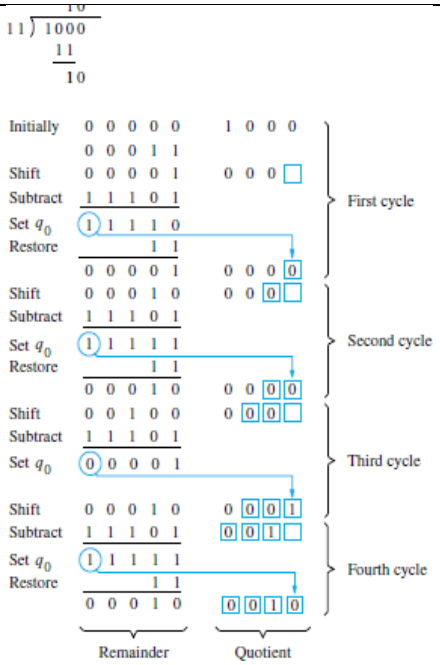


Figure 9.23 Circuit arrangement for binary division.



Restoring Division

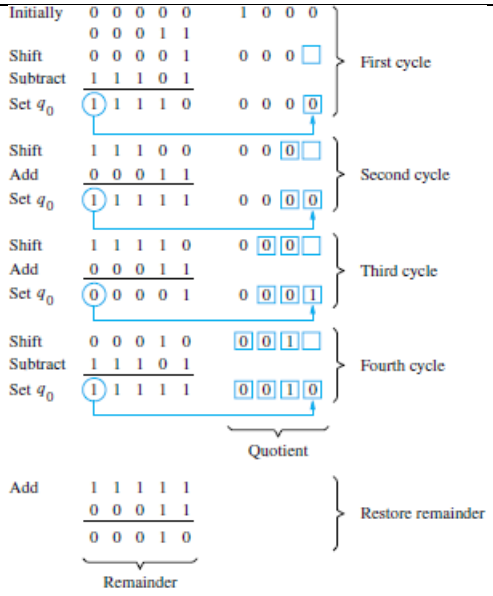


Figure 9.25 A non-restoring division example.

Non restoring division

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.

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.

Do the following three steps *n* times:

1. Shift A and Q left one bit position.
2. Subtract M from A, and place the answer back in A.
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.

If A is positive, we shift left and subtract M, that is, we perform 2A− M. If Ais negative, we restore it by performing A+ M, and then we shift it left and subtract M.

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.*

Stage 1: Do the following two steps *n* times:

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. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

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.

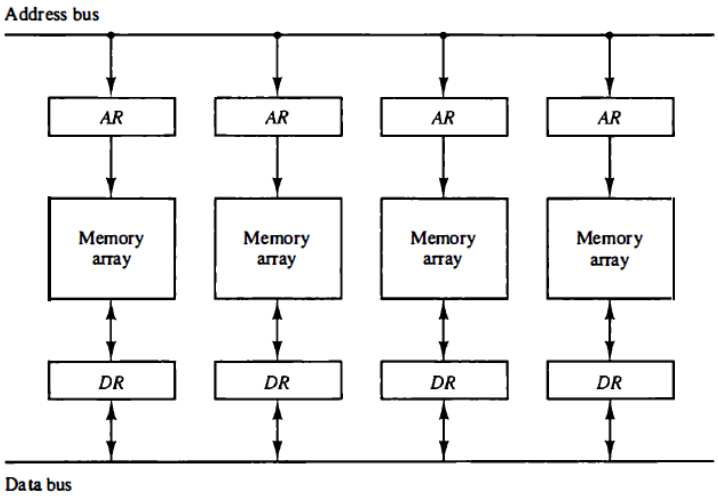
MEMORY ORGANIZATION

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

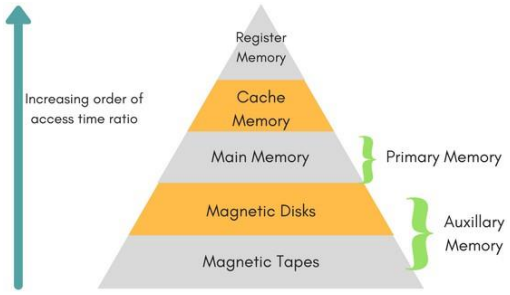
This Memory Hierarchy Design is divided into 2 main types:

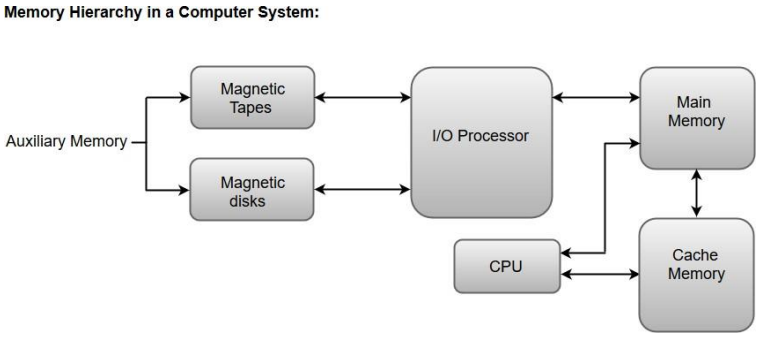
External Memory or Secondary Memory

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.





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.

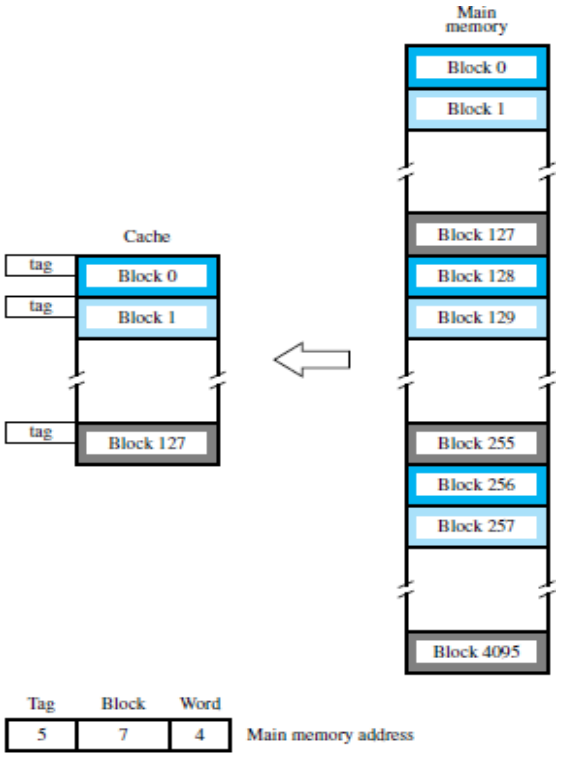


Figure 8.16 Direct-mapped cache.

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.

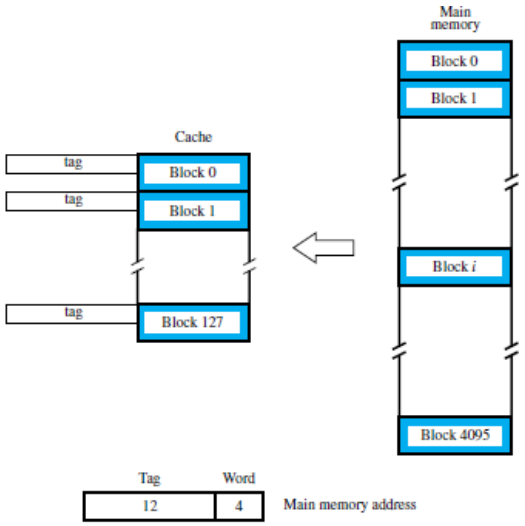


Figure 8.17 Associative-mapped cache.

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.

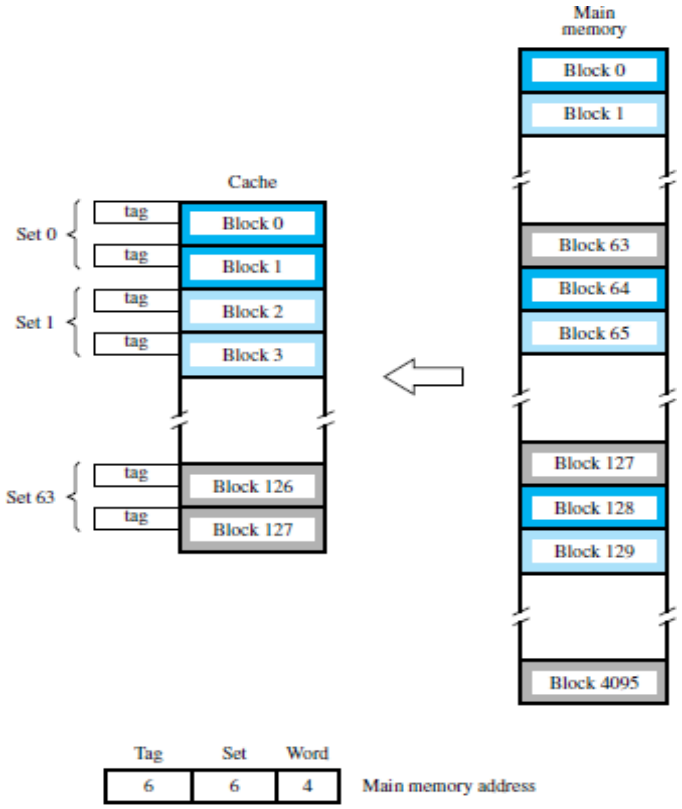


Figure 8.18 Set-associative-mapped cache with two blocks per set.

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, thereplacement 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, thecache 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 inthe 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 iscalled the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many accesspatterns, 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:

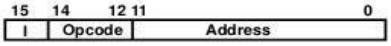
- This technique is to update only the cache location and to mark it as withassociated flag bit called dirty/modified bit.
- The word in the main memory will be updated later, when the block containing thismarked word is to be removed from the cache to make room for a new block.
- To overcome the read miss Load –through / Early restart protocol is used.

Instruction Format:

BASIC COMPUTER INSTRUCTIONS

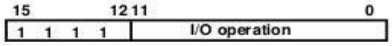
• **Basic Computer Instruction Format**

Memory-Reference Instructions (OP-code = 000 ~ 110)



Register-Reference Instructions (OP-code = 111, I = 0)

Input-Output Instructions (OP-code =111, I = 1)



Computer instructions are the basic components of a machine language program. They are also known as *macro operations*, since each one is comprised of sequences of micro operations.

Each instruction initiates a sequence of micro operations that fetch operands from registers or memory, possibly perform arithmetic, logic, or shift operations, and store results in registers or memory.

Instructions are encoded as binary *instruction codes*. Each instruction code contains of a *operation code*, or *opcode*, which designates the overall purpose of the instruction (e.g. add, subtract, move, input, etc.). The number of bits allocated for the opcode determined how many different instructions the architecture supports.

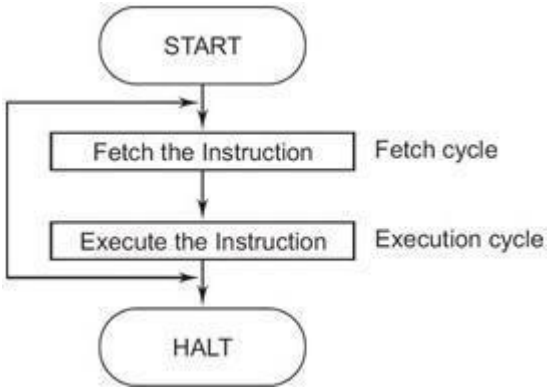
In addition to the opcode, many instructions also contain one or more *operands*, which indicate where in registers or memory the data required for the operation is located. For example, and add instruction requires two operands, and a not instruction requires one.

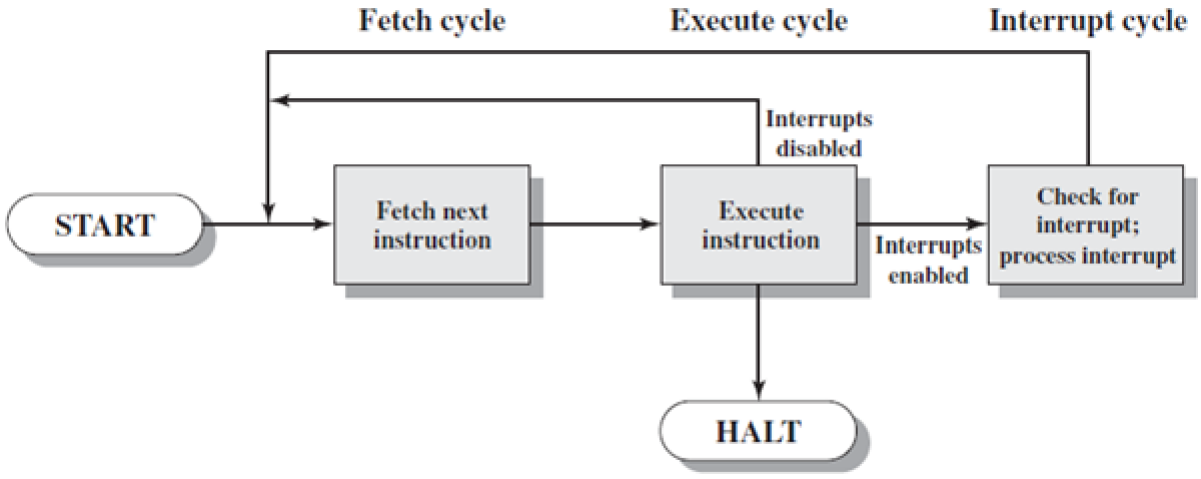
```
 15   12 11        6 5        0
+-----------------------------------+
| Opcode |  Operand     |  Operand  |
+-----------------------------------+
```

The opcode and operands are most often encoded as unsigned binary numbers in order to minimize the number of bits used to store them. For example, a 4-bit opcode encoded as a binary number could represent up to 16 different operations.

The *control unit* is responsible for decoding the opcode and operand bits in the instruction register, and then generating the control signals necessary to drive all other hardware in the CPU to perform the sequence of micro operations that comprise the instruction.

INSTRUCTION CYCLE:





Instruction Cycle with Interrupts

The instruction register (IR):- Holds the instructions that are currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counter PC:-

This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed.

Besides IR and PC, there are n-general purpose registers R0 through R_{n-1}.

The other two registers which facilitate communication with memory are: -

1. **MAR – (Memory Address Register):-** It holds the address of the location to be accessed.
2. **MDR – (Memory Data Register):-** It contains the data to be written into or read outof the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction ofthe program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to thememory.

4. After the time required to access the memory elapses, the address word is read out ofthe memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain therequired operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating aread cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle isinitiated.
12. The contents of PC are incremented so that PC points to the next instruction that is tobe executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal. An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue

THE VON NEUMANN ARCHITECTURE

The task of entering and altering programs for the ENIAC was extremely tedious. The programming process can be easy if the program could be represented in a form suitable for storing in memory alongside the data. Then, a computer could get its instructions by reading them from memory, and a program could be set or altered by setting the values of a portion of memory. This idea is known a the stored-program concept. The first publication of the idea

was in a 1945 proposal by von Neumann for a new computer, the EDVAC (Electronic Discrete Variable Computer).

In 1946, von Neumann and his colleagues began the design of a new stored-program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers.

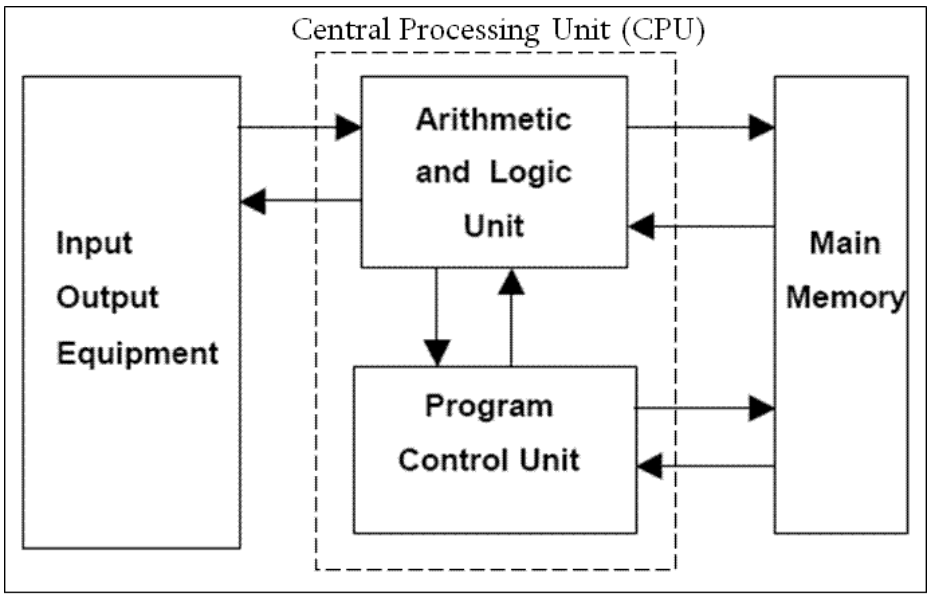


Figure : General structure of Von Neumann Architecture

It consists of

- A main memory, which stores both data and instruction
- An arithmetic and logic unit (ALU) capable of operating on binary data
- A control unit, which interprets the instructions in memory and causes them to be executed
- Input and output (I/O) equipment operated by the control unit

BUS STRUCTURES:

Bus structure and multiple bus structures are types of bus or computing. A bus is basically a subsystem which transfers data between the components of Computer components either within a computer or between two computers. It connects peripheral devices at the same time.

- A multiple Bus Structure has multiple inter connected service integration buses and for each bus the other buses are its foreign buses. A Single bus structure is very simple and consists of a single server.

- A bus cannot span multiple cells. And each cell can have more than one buses. - Published messages are printed on it. There is no messaging engine on Single bus structure

I) In single bus structure all units are connected in the same bus than connecting different buses as multiple bus structure.

II) Multiple bus structure's performance is better than single bus structure. Iii)single bus structure's cost is cheap than multiple bus structure.

Group of lines that serve as connecting path for several devices is called a bus (one bit per line).

Individual parts must communicate over a communication line or path for exchanging data, address and control information as shown in the diagram below. Printer example – processor to printer. A common approach is to use the concept of buffer registers to hold the content during the transfer.

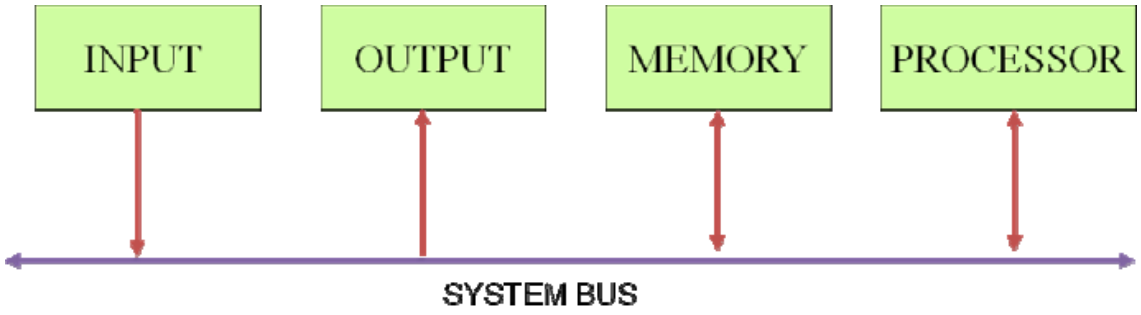


Figure 5: Single bus structure

Buffer registers hold the data during the data transfer temporarily. Ex: printing

Types of Buses:

1. Data Bus:

Data bus is the most common type of bus. It is used to transfer data between different components of computer. The number of lines in data bus affects the speed of data transfer between different components. The data bus consists of 8, 16, 32, or 64 lines. A 64-line data bus can transfer 64 bits of data at one time.

The data bus lines are bi-directional. It means that:

CPU can read data from memory using these lines CPU can write data to memory locations using

these lines

2.Address Bus:

Many components are connected to one another through buses. Each component is assigned

a unique ID.This ID is called the address of that component. It a component wants to communicate

with another component, it uses address bus to specify the address of that component. The address

bus is a unidirectional bus. It can carry information only in one direction. It carries address of

memory location from microprocessor to the main memory.

2.Control Bus:

Control bus is used to transmit different commands or control signals from one component to

another component. Suppose CPU wants to read data from main memory. It will use control is also

used to transmit control signals like ASKS (Acknowledgement signals). A control signal

contains the following:

11.Timing information: It specifies the time for which a device can use data and address bus.

Command Signal: It specifies the type of operation to be performed. Suppose

that CPU gives a command to the main memory to write data. The memory sends acknowledgement

signal to CPU after writing the data successfully. CPU receives the signal and then moves to

perform some other action.

INSTRUCTION FORMATS

The most common fields found in instruction format are:-

(1) An operation code field that specified the operation to be performed

(2) An address field that designates a memory address or a processor registers.

(3) A mode field that specifies the way the operand or the effective address is determined.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address field in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organization.

(1) Single Accumulator organization ADD X AC ® AC + M [×]

(2) General Register Organization ADD R1, R2, R3 R ® R2 + R3

(3) Stack Organization PUSH X

Three address Instruction

Computer with three addresses instruction format can use each address field to specify either processor register are memory operand.

ADD R1, A, B A1 ® M [A] + M [B]

ADD R2, C, D R2 ® M [C] + M [B] X = (A + B) * (C +

A)MUL X, R1, R2 M [X] R1 *

R2

The advantage of the three address formats is that it results in short program when evaluating arithmetic expression. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instruction

Most common in commercial computers. Each address field can specify either a processes register on a memory word.

MOV	R1, A	R1 ® M [A]	
ADD	R1, B	R1 ® R1 + M [B]	
MOV	R2, C	R2 ® M [C]	X = (A + B) * (C + D)
ADD	R2, D	R2 ® R2 + M [D]	
MUL	R1, R2	R1 ® R1 * R2	
MOV	X1 R1	M [X] ® R1	

One Address instruction

It used an implied accumulator (AC) register for all data manipulation. For multiplication/division, there is a need for a second register.

LOAD	A	AC ® M [A]	
ADD	B	AC ® AC + M [B]	
STORE	T	M [T] ® AC	X = (A +B) × (C + A)

All operations are done between the AC register and a memory operand. It's theaddress of a temporary memory location required for storing the intermediate result.

LOAD	C	AC ® M (C)	
ADD	D	AC ® AC + M (D)	
ML	T	AC ® AC + M (T)	
STORE	X	M [×]® AC	

Zero – Address Instruction

A stack organized computer does not use an address field for the instruction ADD and MUL. The PUSH & POP instruction, however, need an address field to specify the operand that communicates with the stack (TOS ® top of the stack)

PUSH	A	TOS ® A	
PUSH	B	TOS ® B	
ADD		TOS ® (A + B)	
PUSH	C	TOS ® C	

PUSH D TOS ® D

ADD TOS ® (C + D)

MUL TOS ® (C + D) * (A +

B)POP X M [X] TOS

Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer register as memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction between the operand is activity referenced. Computer use addressing mode technique for the purpose of accommodating one or both of the following provisions.

(1) To give programming versatility to the uses by providing such facilities as pointer to memory, counters for top control, indexing of data, and program relocation.

(2) To reduce the number of bits in the addressing fields of the instruction.

Addressing Modes: The most common addressing techniques are

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

All computer architectures provide more than one of these addressing modes. The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction

format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of effective address. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

Opcode	Mode	Address

Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:

OPERAND = A

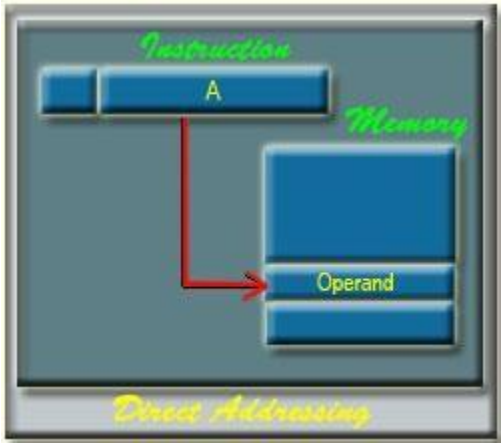
This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

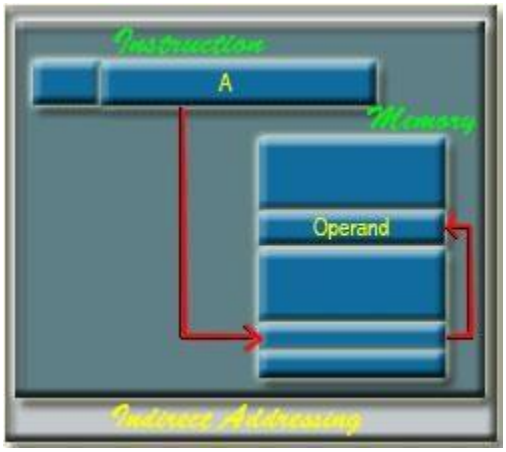
It requires only one memory reference and no special calculation.



Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as indirect addressing:

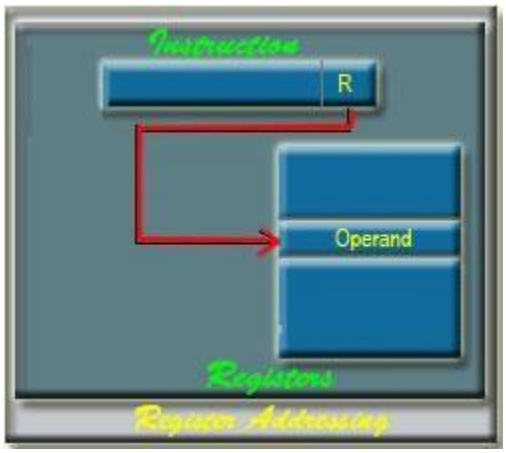
EA = (A)



Register Addressing:

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address: EA = R

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.



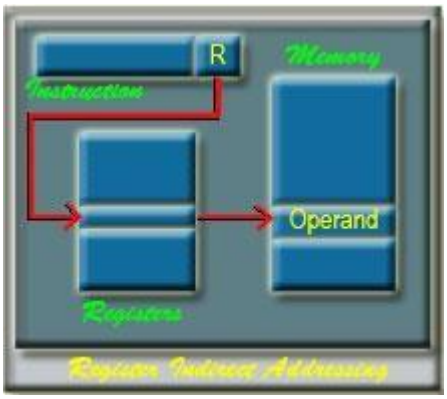
The exact register location of the operand in case of Register Addressing Mode is shown in the Figure 34.4. Here, 'R' indicates a register where the operand is present.

Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.



Displacement Addressing:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:

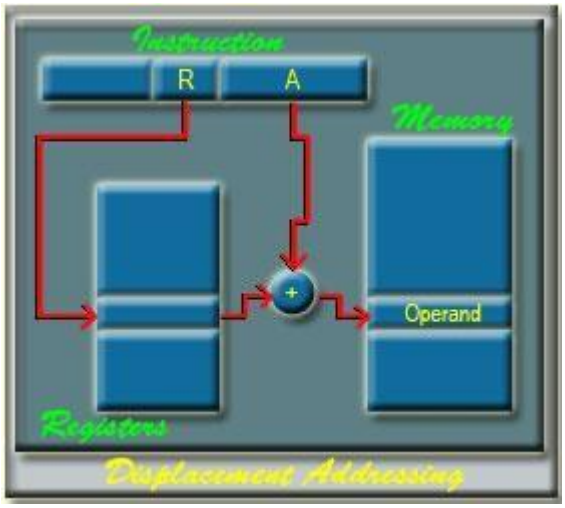
EA = A + (R)

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

The general format of Displacement Addressing is shown in the Figure 4.6.

Three of the most common use of displacement addressing are:

- Relative addressing
- Base-register addressing
- Indexing



Relative Addressing:

For relative addressing, the implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction.

Base-Register Addressing:

The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be explicit or implicit. In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly.

Indexing:

The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit. Generally index register are used for iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some system will automatically do this as part of the same instruction cycle.

This is known as auto-indexing. We may get two types of auto-indexing: -one is auto-incrementing and the other one is -auto-decrementing. If certain registers are devoted exclusively to indexing, then auto-indexing can be invoked implicitly and automatically. If general purpose register are used, the auto index operation may need to be signaled by a bit in the instruction.

Auto-indexing using increment can be depicted as follows:

EA = A +

(R)R = (R) +

1

Auto-indexing using decrement can be depicted as follows:

EA = A +

(R)R = (R) -

1

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: Theindexing is performed either before or after the indirection. If indexing is performed after the indirection, it is termed post indexing

EA = (A) + (R)

First, the contents of the address field are used to access a memory location containing

an address. This address is then indexed by the register value. With pre

indexing, the indexing is performed before the indirection: EA = (A + (R)

An address is calculated, the calculated address contains not the operand, but the address

of the operand.

UNIT – III

MICRO-PROGRAMMED CONTROL: Control memory, address sequencing, micro-program example, design of control unit.

Book: M. Moris Mano (2006), Computer System Architecture, 3rd edition, Pearson/PHI, India:

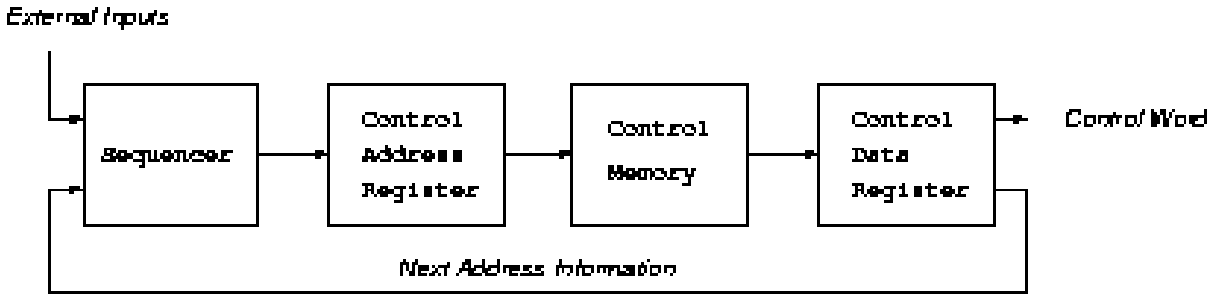
COMPUTER ARITHMETIC: Addition and subtraction, multiplication and division algorithms, floating-point arithmetic operation, decimal arithmetic unit, decimal arithmetic operations.

Book: M. Moris Mano (2006), Computer System Architecture, 3rd edition, Pearson/PHI,India:

Control Memory:

Control memory is a random access memory(RAM) consisting of addressable storage registers. It is primarily used in mini and mainframe computers. It is used as a temporary storage for data. Access to control memory data requires less time than to main memory; this speeds up CPU operation by reducing the number of memory references for data storage and retrieval. Access is performed as part of a control section sequence while the master clock oscillator is running. The control memory addresses are divided into two groups: a task mode and an executive (interrupt) mode.

Addressing words stored in control memory is via the address select logic for each of the register groups. There can be up to five register groups in control memory. These groups select a register for fetching data for programmed CPU operation or for maintenance console or equivalent display or storage of data via maintenance console or equivalent. During programmed CPU operations, these registers are accessed directly by the CPU logic. Data routing circuits are used by control memory to interconnect the registers used in control memory. Some of the registers contained in a control memory that operate in the task and the executive modes include the following: Accumulators Indexes Monitor clock status indicating registers Interrupt data registers



• The control unit in a digital computer initiates sequences of micro operations

• The complexity of the digital system is derived form the number of sequences that are performed

• When the control signals are generated by hardware, it is hardwired

• In a bus-oriented system, the control signals that specify micro operations are groups of bitsthat select the paths in multiplexers, decoders, and ALUs.

• The control unit initiates a series of sequential steps of micro operations

• The control variables can be represented by a string of 1's and 0's called a control word

• A micro programmed control unit is a control unit whose binary control variables are stored in memory

• A sequence of microinstructions constitutes a micro program

• The control memory can be a read-only memory

• Dynamic microprogramming permits a micro program to be loaded and uses a writable control memory

• A computer with a micro programmed control unit will have two separate memories: a main memory and a control memory

• The micro program consists of microinstructions that specify various internal control signals for execution of register micro operations

• These microinstructions generate the micro operations to:

- fetch the instruction from main memory
- evaluate the effective address
- execute the operation
- return control to the fetch phase for the next instruction

• The control memory address register specifies the address of the microinstruction

• The control data register holds the microinstruction read from memory

• The microinstruction contains a control word that specifies one or more micro operations for the data processor

• The location for the next micro instruction may, or may not be the next in sequence

• Some bits of the present micro instruction control the generation of the address of the next micro instruction

• The next address may also be a function of external input conditions

• While the micro operations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next micro instructions

• Typical functions of a sequencer are: o incrementing the CAR by one

- loading into the CAR and address from control memory
- transferring an external address
- loading an initial address to start the control operations

• A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory

• The address value is the input for the ROM and the control work is the output

• No read signal is required for the ROM as in a RAM

• The main advantage of the micro programmed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes

• To establish a different control sequence, specify a different set of microinstructions for control memory

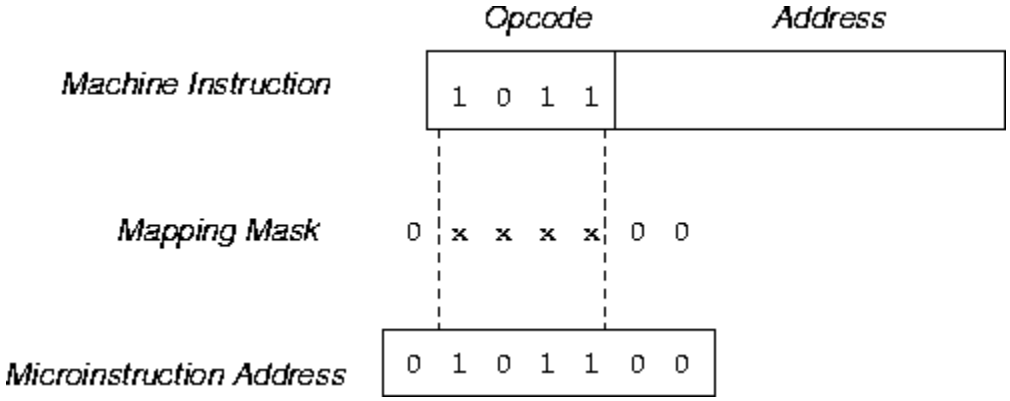
Addressing Sequencing:

Each machine instruction is executed through the application of a sequence of microinstructions. Clearly, we must be able to sequence these; the collection of microinstructions which implements a particular machine instruction is called a *routine*.

The MCU typically determines the address of the first microinstruction which implements a machine instruction based on that instruction's opcode. Upon machine power- up, the CAR should contain the address of the first microinstruction to be executed.

The MCU must be able to execute microinstructions sequentially (e.g., within routines), but must also be able to ``branch" to other microinstructions as required; hence, the need for a sequencer.

The microinstructions executed in sequence can be found sequentially in the CM, or can be found by branching to another location within the CM. Sequential retrieval of microinstructions can be done by simply incrementing the current CAR contents; branching requires determining the desired CW address, and loading that into the CAR.



CAR

Control Address Register

control ROM

control memory (CM); holds CWs

opcode

opcode field from machine instruction

mapping logic

hardware which maps opcode into microinstruction address

branch logic

determines how the next CAR value will be determined from all the various possibilities

multiplexors

implements choice of branch logic for next CAR value

incrementer

generates ***CAR + 1*** as a possible next CAR value

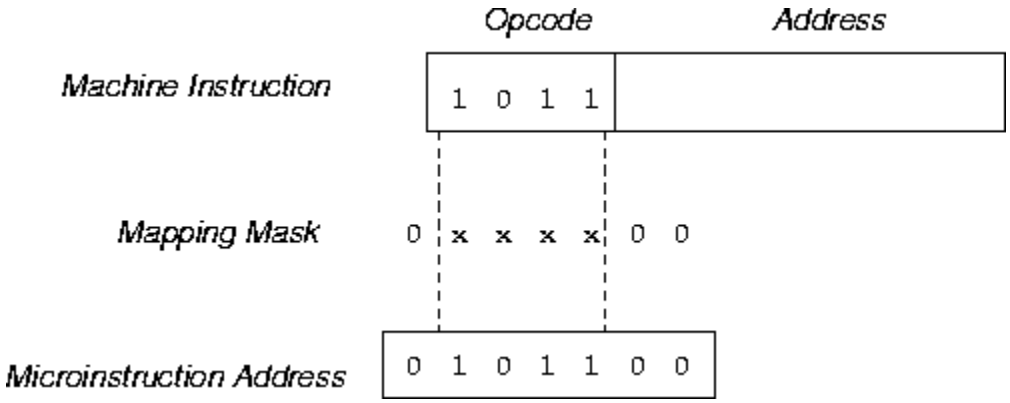
SBR

used to hold return address for subroutine-call branch operations

Conditional branches are necessary in the micro program. We must be able to perform some sequences of micro-ops only when certain situations or conditions exist (e.g., for conditional branching at the machine instruction level); to implement these, we need to be able to conditional execute or avoid certain microinstructions within routines.

Subroutine branches are helpful to have at the micro program level. Many routines contain identical sequences of microinstructions; putting them into subroutines allows those routines to be shorter, thus saving memory. Mapping of opcodes to microinstruction addresses can be done very simply. When the CM is designed, a ``required" length is determine for the machine instruction routines (i.e., the length of the longest one). This is rounded up to the next power of 2, yielding a value ***k*** such that ***2 k*** microinstructions will be sufficient to implement any routine.

The first instruction of each routine will be located in the CM at multiples of this ``required" length. Say this is ***N***. The first routine is at 0; the next, at ***N***; the next, at ***2*N***; etc. This can be accomplished very easily. For instance, with a four-bit opcode and routine length of four microinstructions, ***k*** is two; generate the microinstruction address by appending two zero bits to the opcode:



Alternately, the *n*-bit opcode value can be used as the ``address'' input of a ***2n x M*** ROM; the contents of the selected ``word'' in the ROM will be the desired *M*-bit CAR address for the beginning of the routine implementing that instruction. (This technique allows for variable- length routines in the CM.) >pp We choose between all the possible ways of generating CAR values by feeding them all into a multiplexor bank, and implementing special branch logic which will determine how the muxes will pass on the next address to the CAR.

As there are four possible ways of determining the next address, the multiplexor bank is made up of ***N*** 4x1 muxes, where ***N*** is the number of bits in the address of a CW. The branch logic is used to determine which of the four possible ``next address'' values is to be passed on to the CAR; its two output lines are the select inputs for the muxes.

Eight Conditions for Signed-Magnitude Addition/Subtraction

	Operation	ADD Magnitudes	SUBTRACT Magnitudes		
			A > B	A < B	A = B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+ (A – B)	- (B – A)	+ (A – B)
3	(-A) + (+B)		- (A – B)	+ (B – A)	+ (A – B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+ (A – B)	- (B – A)	+ (A – B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		- (A – B)	+ (B – A)	+ (A – B)

Addition and Subtraction

Four basic computer arithmetic operations are addition, subtraction, division and multiplication. The arithmetic operation in the digital computer manipulate data to produce results. It is necessary to design arithmetic procedures and circuits to program arithmetic operations using algorithm. The algorithm is a solution to any problem and it is stated by a finite number of well-defined procedural steps. The algorithms can be developed for the following types of data.

1. Fixed point binary data in signed magnitude representation
2. Fixed point binary data in signed 2's complement representation.
3. Floating point representation
4. Binary Coded Decimal (BCD) data

Addition and Subtraction with signed magnitude

Consider two numbers having magnitude A and B. When the signed numbers are added or subtracted, there can be 8 different conditions depending on the sign and the operationperformed as shown in the table below:

Operation	Add 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)

From the table, we can derive an algorithm for addition and subtraction as follows:

Addition (Subtraction) Algorithm:

- When the signs of A & B are identical, add the two magnitudes and attach the sign of A to the result.
- When the sign of A & B are different, compare the magnitude and subtract the smaller number from the large number. Choose the sign of the result to be same as A if A > B, or thecomplement of the sign of A if A < B. If the two numbers are equal, subtract B from A and make the sign of the result positive.

Hardware Implementation

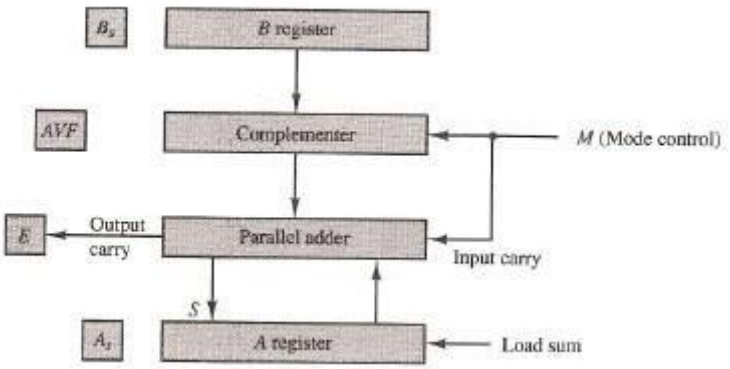


fig: Hardware for signed magnitude addition and subtraction

The hardware consists of two registers A and B to store the magnitudes, and two flip- flops As and Bs to store the corresponding signs. The results can be stored in the register A and As which acts as an accumulator. The subtraction is performed by adding A to the 2's complement of B. The output carry is transferred to the flip-flop E. The overflow may occur during the add operation which is stored in the flip-flop A Ë... F. When m = 0, the output of E is transferred to the adder without any change along with the input carry of '0".

The output of the parallel adder is equal to A + B which is an add operation. When m = 1, the content of register B is complemented and transferred to parallel adder along with the input carry of 1. Therefore, the output of parallel is equal to A + B' + 1 = A – B which is a subtract operation.

Hardware Algorithm

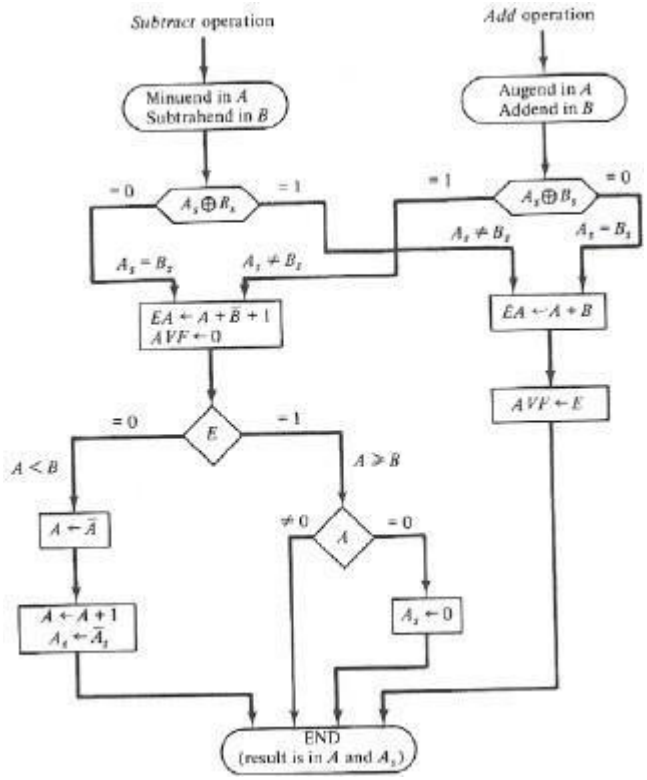


fig: flowchart for add and subtract operations

As and Bs are compared by an exclusive-OR gate. If output=0, signs are identical, if 1 signs are different.

- For Add operation, identical signs dictate addition of magnitudes and for operation identical signs dictate addition of magnitudes and for *subtraction*, different magnitudes dictate magnitudes be added. Magnitudes are added with a micro operation EA
- Two magnitudes are subtracted if signs are different for add operation and identical for subtract operation. Magnitudes are subtracted with a micro operation EA = B and number (this number is checked again for 0 to make positive 0 [As=0]) in A is correct result. E = 0 indicates A < B, so we take 2's complement of A.

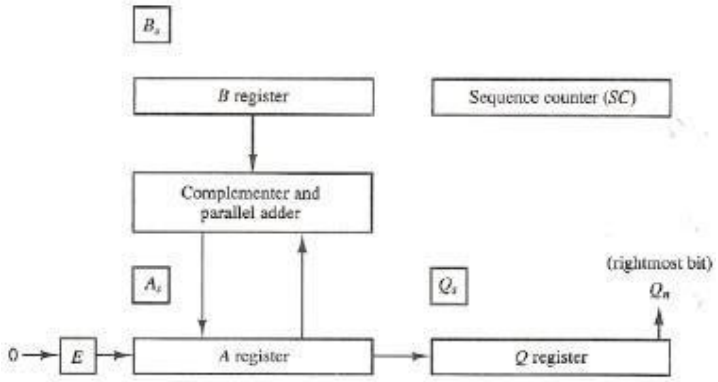
Multiplication

Hardware Implementation and Algorithm

Generally, the multiplication of two final point binary number in signed magnitude representation is performed by a process of successive shift and ADD operation. The process consists of looking at the successive bits of the multiplier (least significant bit first). If the multiplier is 1, then the multiplicand is copied down otherwise, 0's are copied. The numbers

copied down in successive lines are shifted one position to the left and finally, all the numbers are added to get the product.

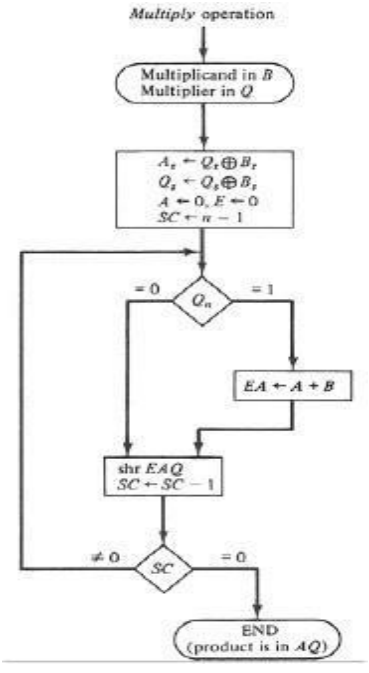
But, in digital computers, an adder for the summation (∑) of only two binary numbers are used and the partial product is accumulated in register. Similarly, instead of shifting the multiplicand to the left, the partial product is shifted to the right. The hardware for the multiplication of signed magnitude data is shown in the figure below.



Hardware for multiply operation

Initially, the multiplier is stored q register and the multiplicand in the B register. A register is used to store the partial product and the sequence counter (SC) is set to a number equal to the number of bits in the multiplier. The sum of A and B form the partial product and both shifted to the right using a statement "Shr EAQ" as shown in the hardware algorithm. The flip flops As, Bs & Qs store the sign of A, B & Q respectively. A binary '0" inserted into the flip-flop E during the shift right.

Hardware Algorithm



flowchart for multiply algorithm

Example: Multiply 23 by 19 using multiply algorithm.

multiplicand	E	A	Q	SC
Initially,	0	00000	10011	101(5)
Iteration1(Qn=1), add B first partial product shrEAQ,	0	00000 +10111 10111		
	0	01011	11001	100(4)
Iteration2(Qn=1) Add B Second partial product shrEAQ,	1	01011 +10111 00010	11001	
	0	10001	01100	011(3)
Iteration3(Qn=0) shrEAQ,	0	01000	10110	010(2)
Iteration4(Qn=0) shrEAQ,	0	00100	01011	001(1)
Iteration5(Qn=1 Add B Fifth partial product shrEAQ,	0	00100 +10111 11011	01011	
	0	01101	10101	000
FinalProductinAQ	0110110101			

The final product is in register A & Q. therefore, the product is 0110110101.

Booth Algorithm

The algorithm that is used to multiply binary integers in signed 2's complement form is called booth multiplication algorithm. It works on the principle that the string 0's in the multiplier doesn't need addition but just the shifting and the sting of 1's from bit weight 2^k to 2^m can be treated as $2^{k+1} - 2^m$ (Example, +14 = 001110 = $2^{3=1} - 2^1 = 14$). The product can be obtained by shifting the binary multiplication to the left and subtraction the multiplier shifted left once.

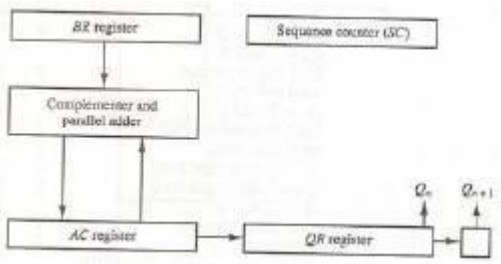
According to booth algorithm, the rule for multiplication of binary integers in signed 2's complement form are:

- The multiplicand is subtracted from the partial product of the first least significant bit is 1in a string of 1's in the multiplicand.

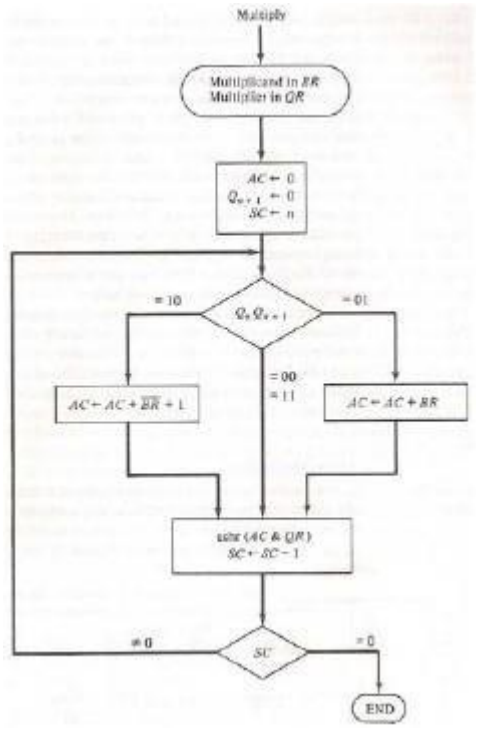
- The multiplicand is added to the partial product if the first least significant bit is 0(provided that there was a previous 1) in a string of 0's in the multiplier.
- The partial product doesn't change when the multiplier bit is identical to the previous multiplier bit.

This algorithm is used for both the positive and negative numbers in signed 2's complement form.

The hardware implementation of this algorithm is in figure below:



The flowchart for booth multiplication algorithm is given below:



flowchart for booth multiplication algorithm

Numerical Example: Booth algorithm

BR=10111(**Multiplicand**)

QR=10011(**Multiplier**)

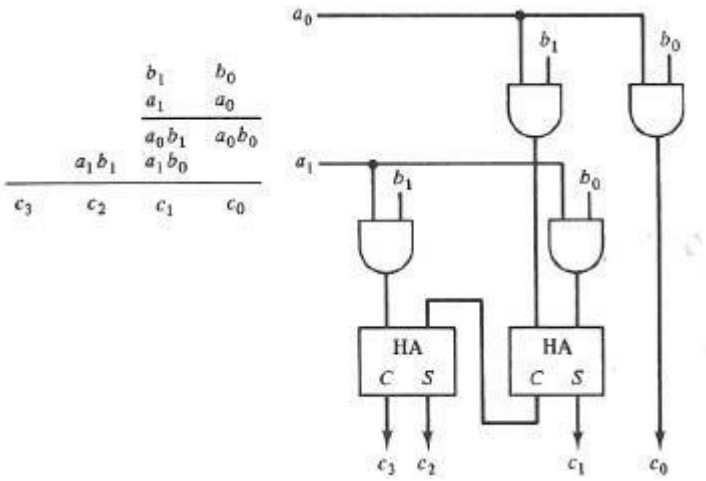
Array Multiplier

The multiplication algorithm first check the bits of the multiplier one at time and form partial product. This is a sequential process that requires a sequence of add and shift micro operation. This method is complicated and time consuming. The multiplication of 2 binary

numbers can also be done with one micro operation by using combinational circuit that provides the product all at once.

Example.

Consider that the multiplicand bits are b1 and b0 and the multiplier bits are a1 and a0. The partial product is c3c2c1c0. The multiplication two bits a0 and a1 produces a binary 1 if both the bits are 1, otherwise it produces a binary 0. This is identical to the AND operation and can be implemented with the AND gates as shown in figure.



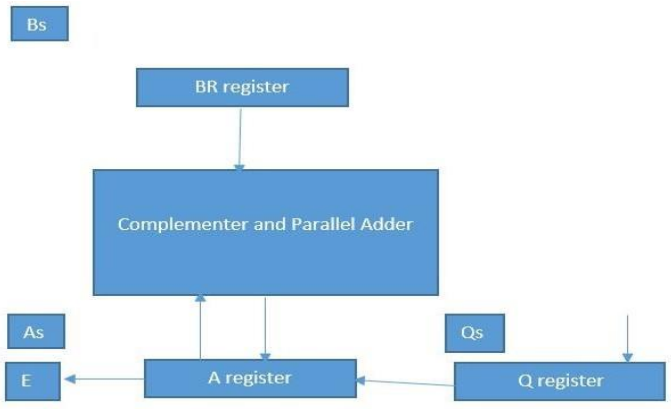
2-bit by 2-bit array multiplier

Division Algorithm

The division of two fixed point signed numbers can be done by a process of successive compare shift and subtraction. When it is implemented in digital computers, instead of shifting the divisor to the right, the dividend or the partial remainder is shifted to the left. The subtraction can be obtained by adding the number A to the 2's complement of number B. The information about the relative magnitudes of the information about the relative magnitudesof numbers can be obtained from the end carry,

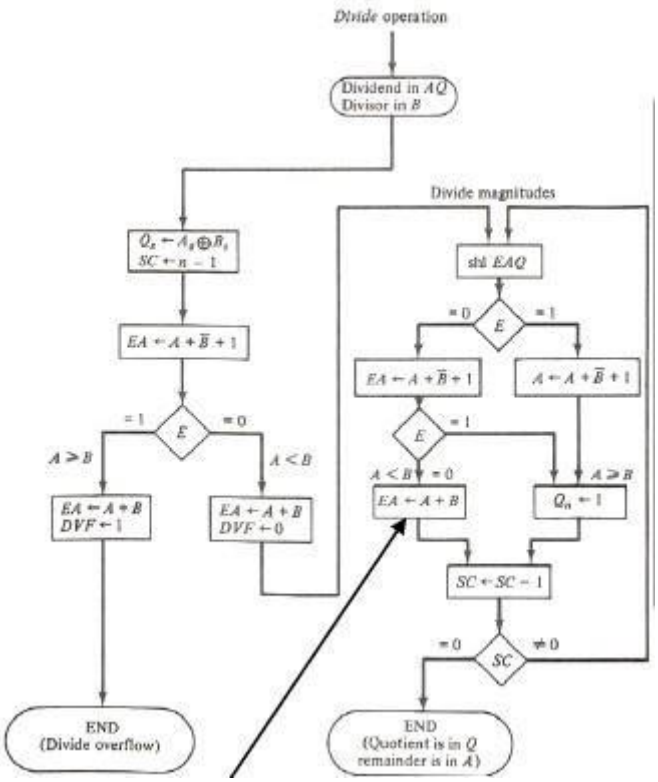
Hardware Implementation

The hardware implementation for the division signed numbers is shown id the figure.



Division Algorithm

The divisor is stored in register B and a double length dividend is stored in register A and Q. the dividend is shifted to the left and the divider is subtracted by adding twice complement of the value. If E = 1, then A >= B. In this case, a quotient bit 1 is inserted into Qn and the partial remainder is shifted to the left to repeat the process. If E = 0, then A > B. In this case, the quotient bit Qn remains zero and the value of B is added to restore the partial remainder in A to the previous value. The partial remainder is shifted to the left and approaches continues until the sequence counter reaches to 0. The registers E, A & Q are shifted to the left with 0 inserted into Qn and the previous value of E is lost as shown in the flow chart for division algorithm.



flowchart for division algorithm

This algorithm can be explained with the help of an example.

Consider that the divisor is 10001 and the dividend is 01110.

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		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
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	

binary division with digital hardware

Restoring method

Method described above is restoring **method** in which partial remainder is restored by adding the divisor to the negative result. Other methods:

Comparison method: A and B are compared prior to subtraction. Then if A >= B, B is subtracted from A. if A < B nothing is done. The partial remainder is then shifted left and numbers are compared again. Comparison inspects end carry out of the parallel adder before transferring to E.

Non-restoring method: In contrast to restoring method, when A -B is negative, B is not added to restore A but instead, negative difference is shifted left and then B is added. How is it possible? Let's argue:

- In flowchart for restoring method, when A < B, we restore A by operation A - B + B. Next time in a loop,

this number is shifted left (multiplied by 2) and B subtracted again, which gives: 2 (A - B + B) – B = 2 A - B.

- In Non-restoring method, we leave A - B as it is. Next time around the loop, the number is shifted left and B is added: 2 (A - B) + B = 2 A - B (same as above).

Divide Overflow

The division algorithm may produce a quotient overflow called dividend overflow. The overflow can occur of the number of bits in the quotient are more than the storage capacity of the register. The overflow flip-flop DVF is set to 1 if the overflow occurs.

The division overflow can occur if the value of the half most significant bits of the dividend is equal to or greater than the value of the divisor. Similarly, the overflow can occue=r if the dividend is divided by a 0. The overflow may cause an error in the result or sometimes it may stop the operation. When the overflow stops the operation of the system, then it is called divide stop.

Arithmetic Operations on Floating-Point Numbers

The rules apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations. Intermediate results for both mantissas and exponents might require more than 24 and 8 bits,respectively & overflow or an underflow may occur. These and other aspects of the operations must be carefully considered in] designing an arithmetic unit that meets the standard. If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add 2.9400 x 10^2 to 4.3100 x 10^4. We rewrite 2.9400 x 10^2 as 0.0294 x 10^4 and then perform addition of the mantissas to get 4.3394 x 10^4 . The rule for addition and subtraction can be stated as follows:

UNIT – IV **(10 Lectures)**

THE MEMORY SYSTEM: Basic concepts, semiconductor RAM types of read - only memory (ROM), cache memory, performance considerations, virtual memory, secondary storage, raid, direct memory access (DMA).

Book: Carl Hamacher, Zvonks Vranesic, SafeaZaky (2002), Computer Organization, 5th edition,

BASIC CONCEPTS OF MEMORY SYSTEM

The maximum size of the Main Memory (MM) that can be used in any computer is determined by its addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing upto 216 =64K memory locations. If a machine generates 32-bit addresses, it can access upto 2^{32} = 4G memory locations. This number represents the size of address space of the computer.

If the smallest addressable unit of information is a memory word, the machine is called word-addressable. If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable. Most of the commercial machines are byte addressable. For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes. A possible word-address assignment would be:

Word Address	Byte Address
0	0 1 2 3
4	4 5 6 7
8	8 9 10 11
.	….. . ….. . …..

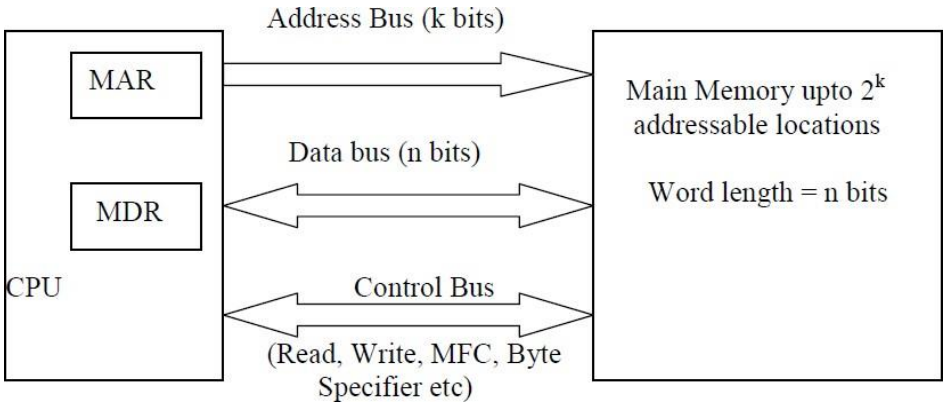
With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte. In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered. In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

CPU-Main Memory Connection – A block schematic: -

From the system standpoint, the Main Memory (MM) unit can be viewed as a “block box”. Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register). If MAR is K bits long and MDR is ‘n’ bits long, then the MM unit may contain upto 2^k addressable locations and

each location will be ‘n’ bits wide, while the word length is equal to ‘n’ bits. During a “memory cycle”, n bits of data may be transferred between the MM and CPU.

This transfer takes place over the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus). For a read operation, the CPU loads the address into MAR, set READ to 1 and sets other control signals if required. The data from the MM is loaded into MDR and MFC is set to 1. For a write operation, MAR, MDR are suitably loaded by the CPU, write is set to 1 and other control signals are set suitably. The MM control circuitry loads the data into appropriate locations and sets MFC to 1. This organization is shown in the following block schematic.



Address Bus (k bits) Main Memory upto 2k addressable locations Word length = n bits Data bus (n bits) Control Bus (Read, Write, MFC, Byte Specifier etc) MAR MDR CPU

Some Basic Concepts

Memory Access Times: - It is a useful measure of the speed of the memory unit. It is the time that elapses between the initiation of an operation and the completion of that operation (for example, the time between READ and MFC).

Memory Cycle Time :- It is an important measure of the memory system. It is the minimum time delay required between the initiations of two successive memory operations (for example, the time between two successive READ operations). The cycle time is usually slightly longer than the access time.

Random Access Memory (RAM):

A memory unit is called a Random Access Memory if any location can be accessed for a READ or WRITE operation in some fixed amount of time that is independent of the location's address. Main memory units are of this type. This distinguishes them from serial or partly serial access storage devices such as magnetic tapes and disks which are used as the secondary storage device.

Cache Memory:-

The CPU of a computer can usually process instructions and data faster than they can be fetched from compatibly priced main memory unit. Thus the memory cycle time become the bottleneck in the system. One way to reduce the memory access time is to use cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. This holds the currently active segments of a program and its data. Because of the locality of address references, the CPU can, most of the time, find the relevant information in the cache memory itself (cache hit) and infrequently needs access to the main memory (cache miss) with suitable size of the cache memory, cache hit rates of over 90% are possible leading to a cost-effective increase in the performance of the system.

Memory Interleaving: -

This technique divides the memory system into a number of memory modules and arranges addressing so that successive words in the address space are placed in different modules. When requests for memory access involve consecutive addresses, the access will be to different modules. Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Virtual Memory: -

In a virtual memory System, the address generated by the CPU is referred to as a virtual or logical address. The corresponding physical address can be different and the required mapping is implemented by a special memory control unit, often called the memory management unit. The mapping function itself may be changed during program execution according to system requirements.

Because of the distinction made between the logical (virtual) address space and the physical address space; while the former can be as large as the addressing capability of the CPU, the actual physical memory can be much smaller. Only the active portion of the virtual address space is mapped onto the physical memory and the rest of the virtual address space

is mapped onto the bulk storage device used. If the addressed information is in the Main Memory (MM), it is accessed and execution proceeds.

Otherwise, an exception is generated, in response to which the memory management unit transfers a contiguous block of words containing the desired word from the bulk storage unit to the MM, displacing some block that is currently inactive. If the memory is managed in such a way that, such transfers are required relatively infrequency (ie the CPU will generally find the required information in the MM), the virtual memory system can provide a reasonably good performance and succeed in creating an illusion of a large memory with a small, in expensive MM.

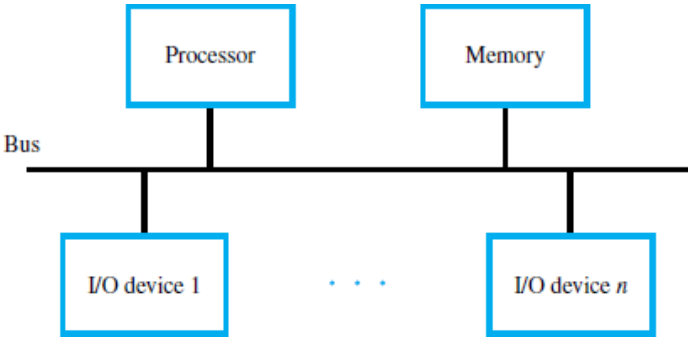
UNIT – V

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 electromechnical 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.

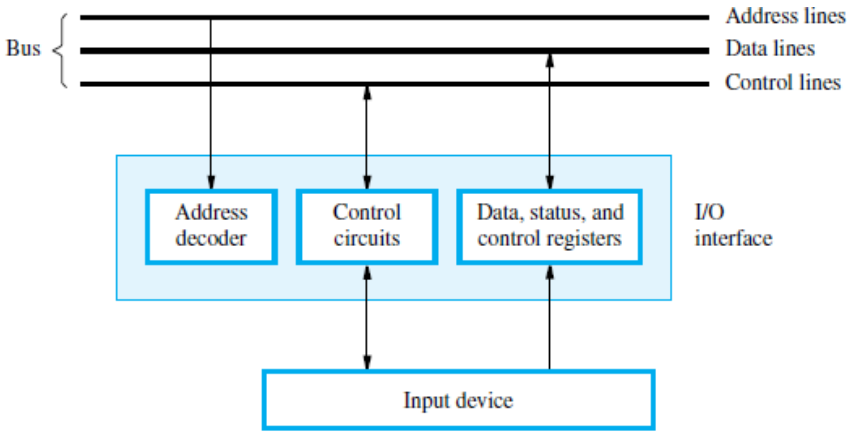


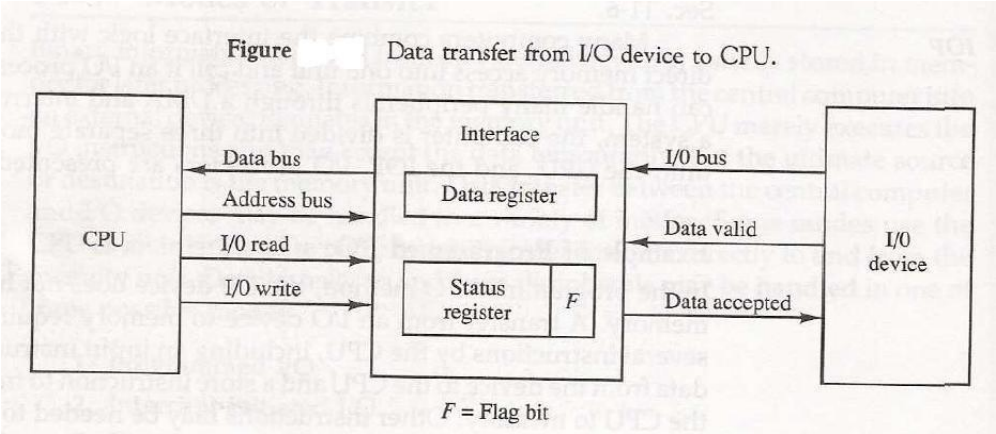
Figure 7.2 I/O interface for an input device.

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.



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.

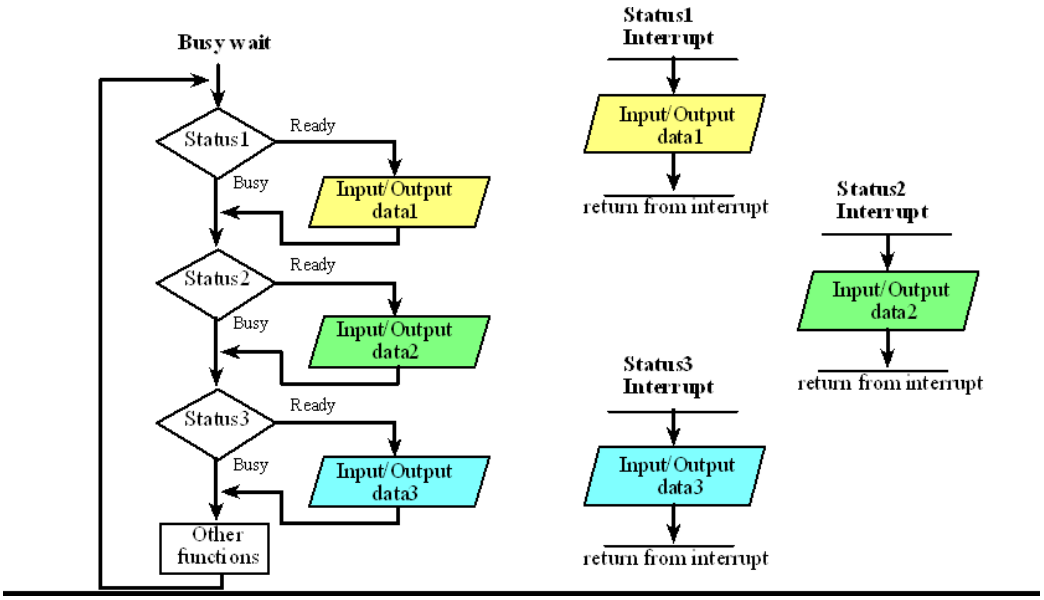
Programs and processes-Role of interrupts in process state transitions

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.



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

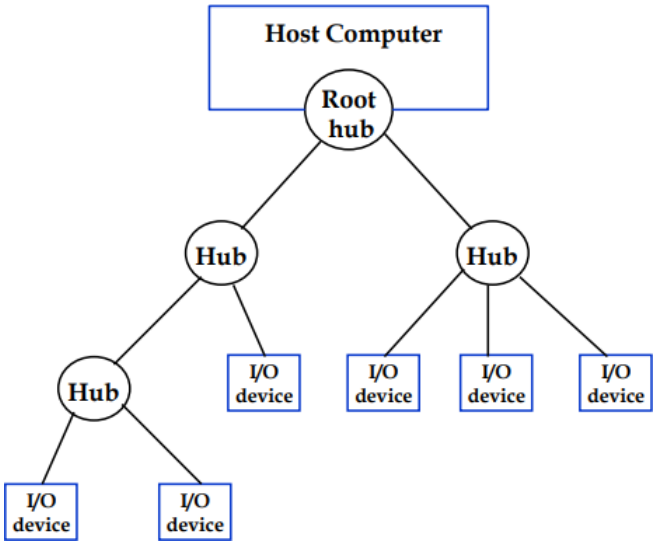
The USB has been designed to meet several key objectives:

- 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:

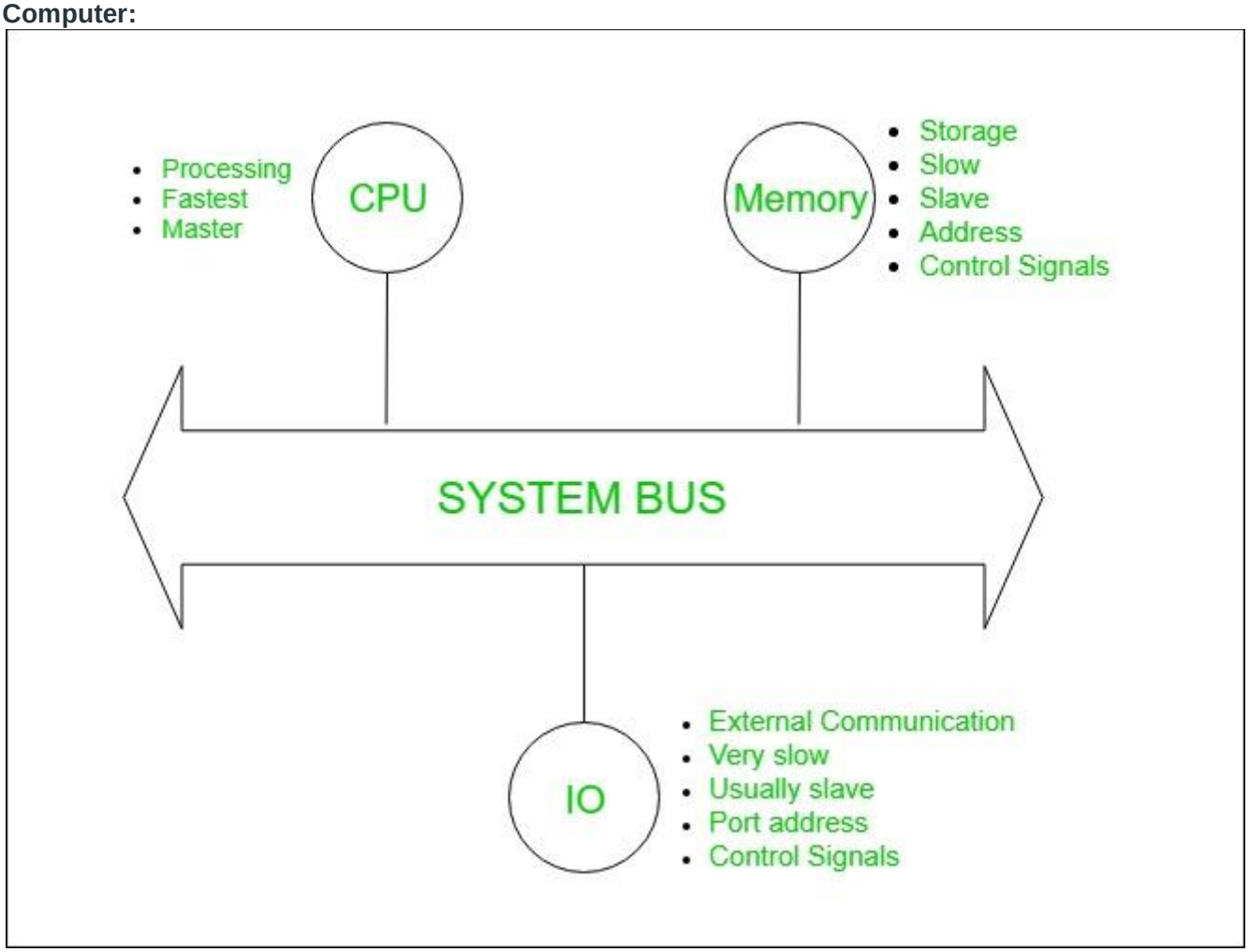
- All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information.
- The information transferred on the USB can be divided into two broad categories: control and data. ‹ Control packets perform such tasks as addressing a device to initiate 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

Bus and System Bus

Definition:
The electrically conducting path along which data is transmitted inside any digital electronic device. A Computer bus consists of a set of parallel conductors, which may be conventional wires, copper tracks on a PRINTED CIRCUIT BOARD, or microscopic aluminum trails on the surface of a silicon chip. Each wire carries just one bit, so the number of wires determines the most significant data WORD the bus can transmit: a bus with eight wires can carry only 8-bit data words and hence defines the device as an 8-bit device.

- The bus is a communication channel.
- The characteristic of the bus is shared transmission media.
- The limitation of a bus is only one transmission at a time.
- A bus used to communicate between the major components of a computer is called a **System bus**.

Computer:



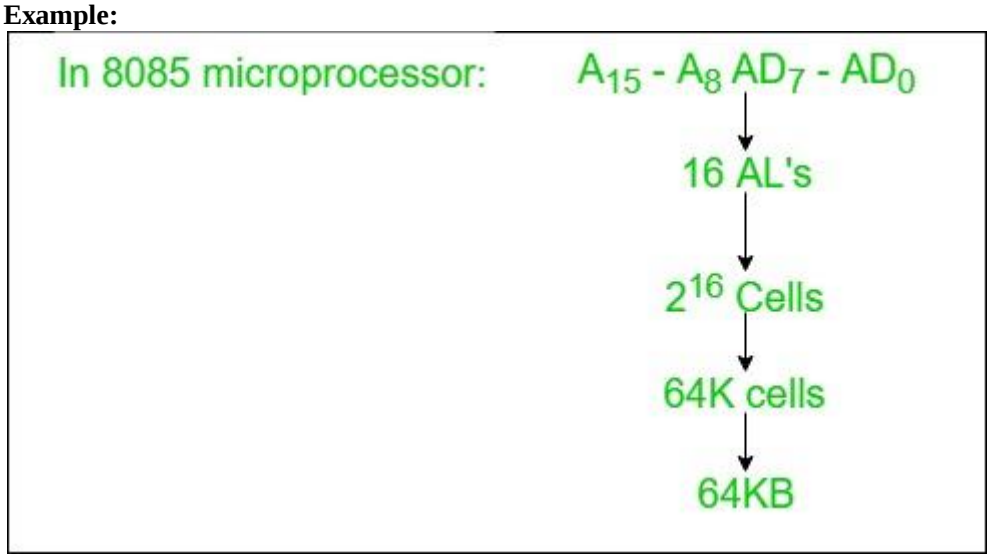
System bus contains 3 categories of lines used to provide the communication between the CPU, memory and IO named as:

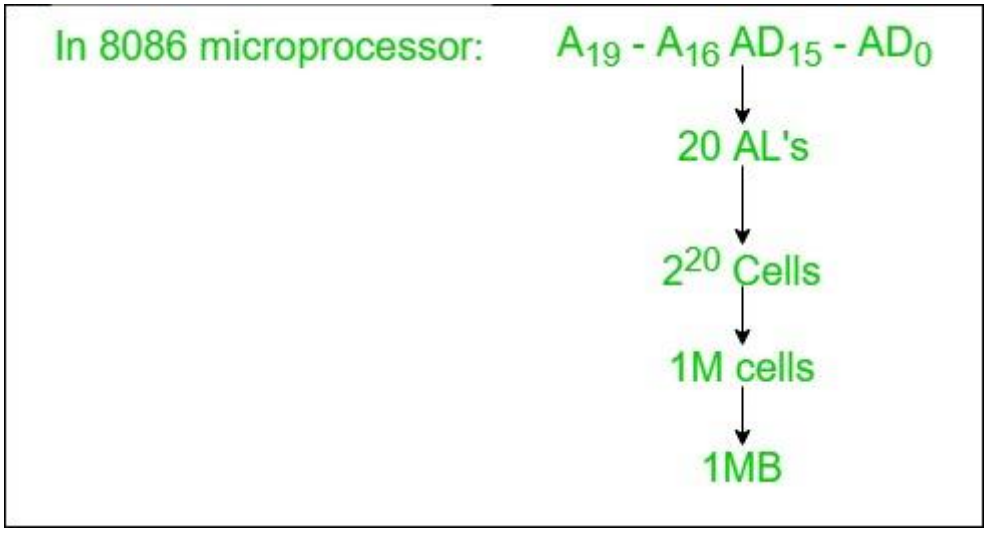
1. Address lines (AL)
2. Data lines (DL)
3. Control lines (CL)

1. Address Lines:

- Used to carry the address to memory and IO.
- Unidirectional.
- Based on the width of an address bus we can determine the capacity of a main memory

Example:

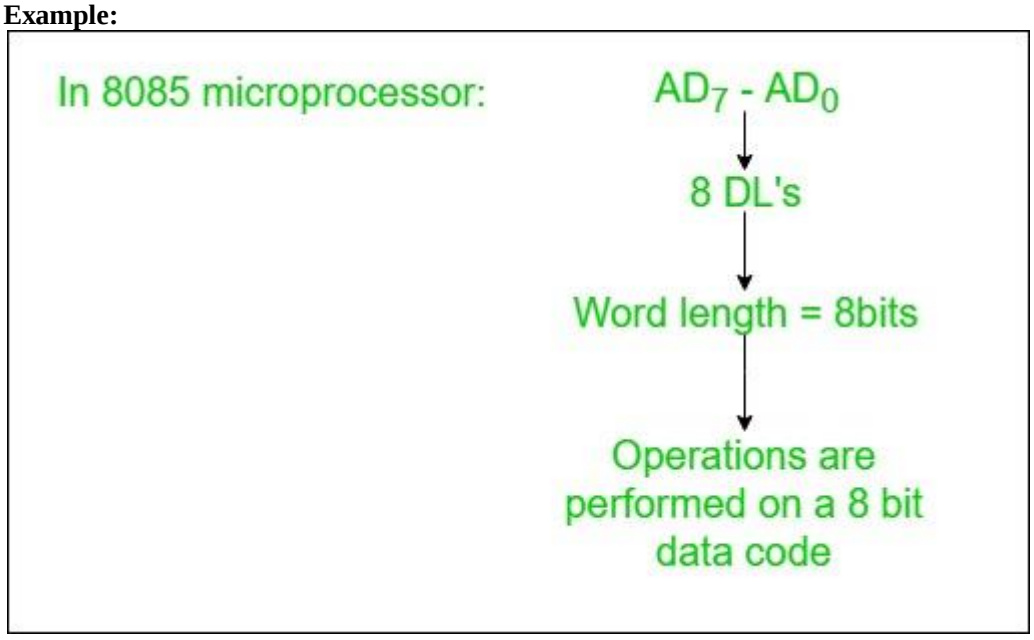


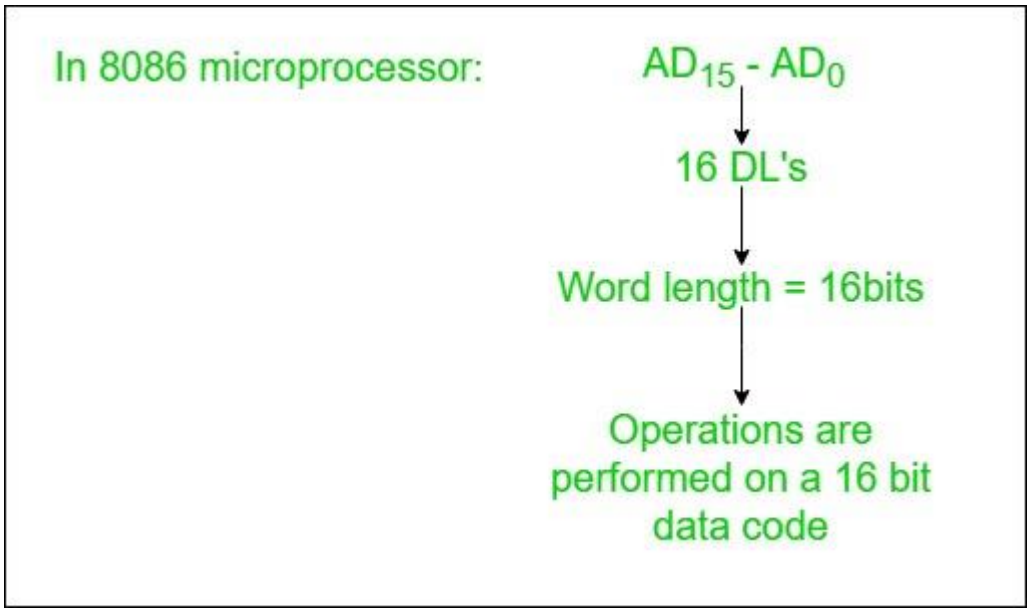


2. Data Lines:

- Used to carry the binary data between the CPU, memory and IO.
- Bidirectional.
- Based on the width of a data bus we can determine the word length of a CPU.
- Based on the word length we can determine the performance of a CPU.

Example:





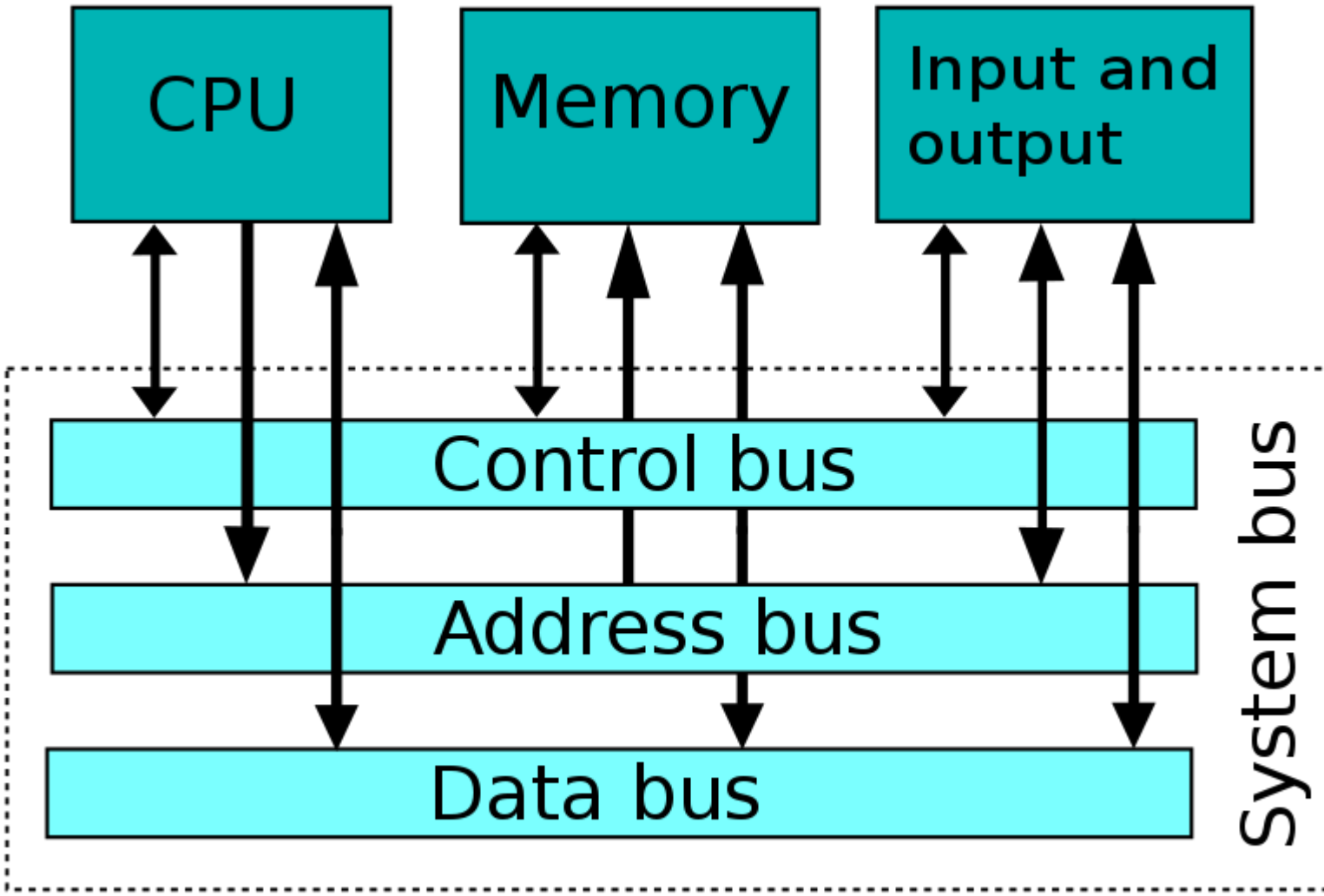
3. Control Lines:

- Used to carry the control signals and timing signals
- Control signals indicate the type of operation.
- Timing Signals are used to synchronize the memory and IO operations with a CPU clock.
- Typical Control Lines may include Memory Read/Write, IO Read/Write, Bus Request/Grant, etc.

Last Updated : 02 Apr, 2023

system bus explain its types with a diagram

is a bus that is a communication system that transfers data between components inside a computer or between computers. This expression covers all related hardware components and **software** including communication
protocols.



The bus consists of three main parts

1. control Bus: The control bus carries that control signal. The control signal is used for controlling and coordinating the various activities across the computer. It is generated from the control unit within the CPU. The control unit generates a specific **Control Signal** for every operation, such as a memory rod or inputs Joulut operation. This signal is also used to identify a device type, with which the microprocessor communicates.

2. **Address Bus:** The address bus carries the memory of the address within the device and allows the CPU to reference memory locations within the device. It connects the CPU and another peripheral and carries only the memory address. The address bus contains the connections between the processor and memory that carry the signals relating to the addresses which the **CPU** is processing at that time, such as the locations that the CPU is reading from or writing to. It addresses but could carry 8 bit at a time, the CPU could address only. (2^8) 256 bytes of RAM.

3. Data Bus: Data bus transfer data from one location to another across the computer. The meaningful data which is to be sent from a device is placed on these lines. The CPU uses a data bus to transfer data. It May be a 16 or 32-bit data bus. It is an **electrical** connects the CPU, memory, and other hardware devices on the motherboard. These lines are bidirectional data flow in both directions between the **processor** and memory and peripheral devices.

Basic Parameters of Bus design

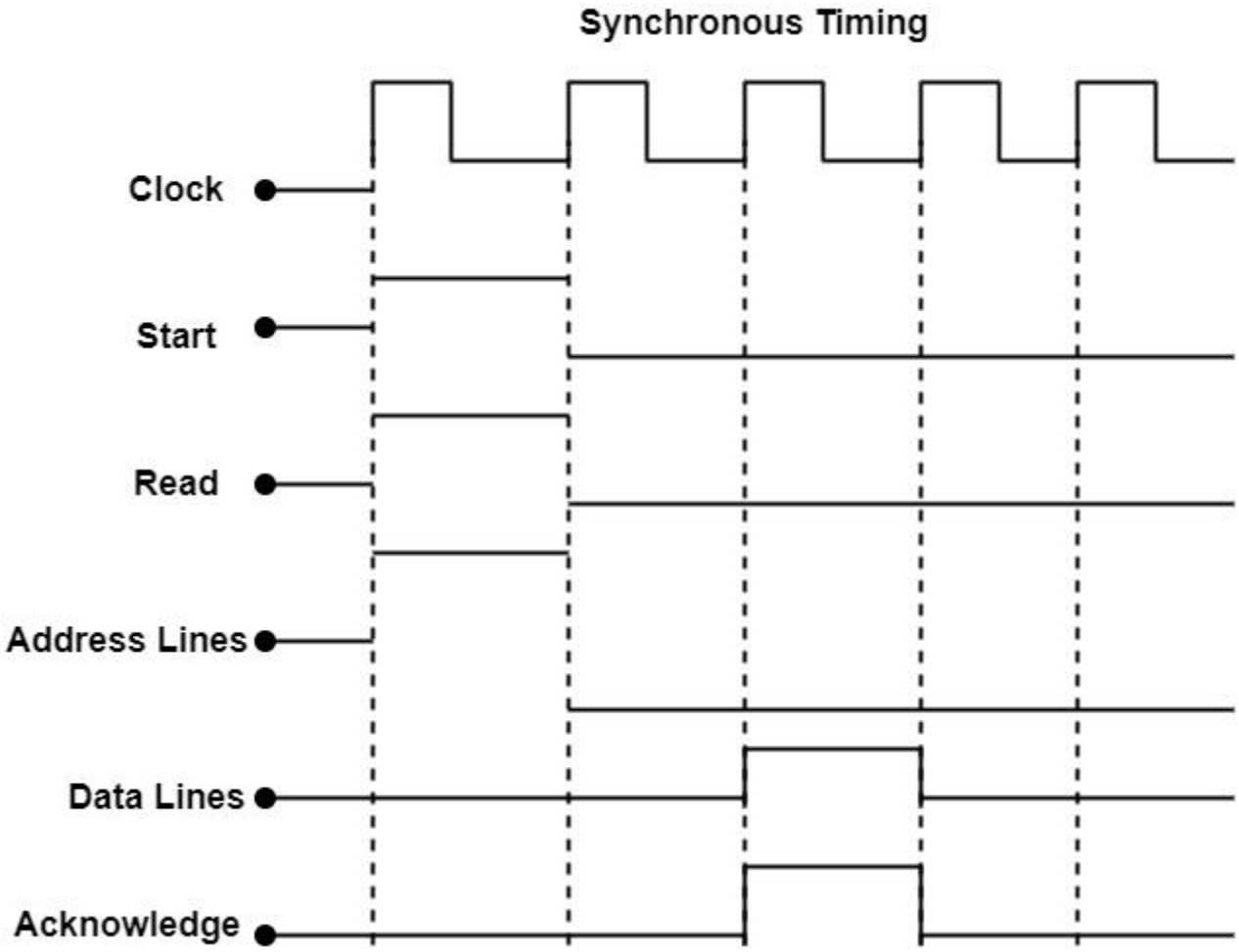
Method of Arbitration

In all but the simplest systems, more than one module can require control of the bus. Therefore only one unit at a time can strongly transfer over the bus, some method of arbitration is required. Various methods can be classified as centralized or distributed.

In a centralized scheme, a single hardware device defined as a bus controller or arbiter is important for assigning time on the bus. In a distributed scheme, there is no central controller. Each module includes access control logic and the modules help together to transfer the bus.

Timing

Timing defines how events are integrated on the bus. With synchronous timing, the circumstances of events on the bus are persistent by a clock. The figure shows the timing diagram for a synchronous read operation.



With asynchronous timing, the circumstances of one event on a bus follows and are based on the circumstances of a previous event. In this example, the CPU places address and read signals on the bus.

After pausing for these signals to maintain, it declares an MSYN (master sync) signal. It indicating the presence of valid current address and control signals. The memory module responds with data and an SSYN (slave sync) signal, indicating the response.

SCSI

Small Computer System Interface (**SCSI**, /ˈskʌzi/ *SKUZ-ee*)[1] is a set of standards for physically connecting and transferring data between computers and peripheral devices. The SCSI standards define commands, protocols, electrical, optical and logical interfaces. The SCSI standard defines command sets for specific peripheral device types; the presence of "unknown" as one of these types means that in theory it can be used as an interface to almost any device, but the standard is highly pragmatic and addressed toward commercial requirements. The initial Parallel SCSI was most commonly used for hard disk drives and tape drives, but it can connect a wide range of other devices, including scanners and CD drives, although not all controllers can handle all devices.

The ancestral SCSI standard, X3.131-1986, generally referred to as SCSI-1, was published by the X3T9 technical committee of the American National Standards Institute (ANSI) in 1986. SCSI-2 was published in August 1990 as X3.T9.2/86-109, with further revisions in 1994 and subsequent adoption of a multitude of interfaces. Further refinements have resulted in improvements in performance and support for ever-increasing data storage capacity.[2]

History[edit]

Parallel interface[edit]


Adaptec ACB-4000A SASI card from 1985

SCSI is derived from "SASI", the "Shugart Associates System Interface", developed beginning 1979[3] and publicly disclosed in 1981.[4] Larry Boucher is considered to be the "father" of SASI and ultimately SCSI due to his pioneering work first at Shugart Associates and then at Adaptec.[5]

A SASI controller provided a bridge between a hard disk drive's low-level interface and a host computer, which needed to read blocks of data. SASI controller boards were typically the size of a hard disk drive and were usually physically mounted to the drive's chassis. SASI, which was used in mini- and early microcomputers, defined the interface as using a 50-pin flat ribbon connector which was adopted as the SCSI-1 connector. SASI is a fully compliant subset of SCSI-1 so that many, if not all, of the then-existing SASI controllers were SCSI-1 compatible.[6]

Until at least February 1982, ANSI developed the specification as "SASI" and "Shugart Associates System Interface"[7] however, the committee documenting the standard would not allow it to be named after a company. Almost a full day was devoted to agreeing to name the standard "Small Computer System Interface", which Boucher intended to be pronounced "sexy", but ENDL's[8] Dal Allan pronounced the new acronym as "scuzzy" and that stuck.[5]

A number of companies such as NCR Corporation, Adaptec and Optimem were early supporters of SCSI.[7] The NCR facility in Wichita, Kansas is widely thought to have developed the industry's first SCSI controller chip; it worked the first time.[9]

The "small" reference in "small computer system interface" is historical; since the mid-1990s, SCSI has been available on even the largest of computer systems.

Since its standardization in 1986, SCSI has been commonly used in the Amiga, Atari, Apple Macintosh and Sun Microsystems computer lines and PC server systems. Apple started using the less-expensive parallel ATA (PATA, also known as *IDE*) for its low-end machines with the Macintosh Quadra 630 in 1994, and added it to its high-end desktops starting with the Power Macintosh G3 in 1997. Apple dropped on-board SCSI completely in favor of IDE and FireWire with the (Blue & White) Power Mac G3 in 1999, while still offering a PCI SCSI host adapter as an option on up to the Power Macintosh G4 (AGP Graphics) models.[10] Sun switched its lower-end range to Serial ATA (SATA). Commodore included SCSI on the Amiga 3000/3000T systems and it was an add-on to previous Amiga 500/2000 models. Starting with the Amiga 600/1200/4000 systems Commodore switched to the IDE interface. Atari included SCSI as standard in its Atari MEGA STE, Atari TT and Atari Falcon computer models. SCSI has never been popular in the low-priced IBM PC world, owing to the lower cost and adequate performance of ATA hard disk standard. However, SCSI drives and even SCSI RAIDs became common in PC workstations for video or audio production.

Modern SCSI[edit]

Recent physical versions of SCSI—Serial Attached SCSI (SAS), SCSI-over-Fibre Channel Protocol (FCP), and USB Attached SCSI (UAS)—break from the traditional parallel SCSI bus and perform data transfer via serial communications using point-to-point links. Although much of the SCSI documentation talks about the parallel interface, all modern development efforts use serial interfaces. Serial interfaces have a number of advantages over parallel SCSI, including higher data rates, simplified cabling, longer reach, improved fault isolation and full-duplex capability. The primary reason for the shift to serial interfaces is the clock skew issue of high speed parallel interfaces, which makes the faster variants of parallel SCSI susceptible to problems caused by cabling and termination.[11]

The non-physical iSCSI preserves the basic SCSI paradigm, especially the command set, almost unchanged, through embedding of SCSI-3 over TCP/IP. Therefore, iSCSI uses *logical connections* instead of physical links and can run on top of any network supporting IP. The actual physical links are realized on lower network layers, independently from iSCSI. Predominantly, Ethernet is used which is also of serial nature.

SCSI is popular on high-performance workstations, servers, and storage appliances. Almost all RAID subsystems on servers have used some kind of SCSI hard disk drives for decades (initially Parallel SCSI, interim Fibre Channel, recently SAS), though a number of manufacturers offer SATA-based RAID subsystems as a cheaper option. Moreover, SAS offers compatibility with SATA devices, creating a much broader range of options for RAID subsystems together with the existence of nearline SAS (NL-SAS) drives. Instead of SCSI, modern desktop computers and notebooks typically use SATA interfaces for internal hard disk drives, with NVMe over PCIe gaining popularity as SATA can bottleneck modern solid-state drives.

Interfaces[edit]

Main article: SCSI connector

SCSI is available in a variety of interfaces. The first was parallel SCSI (also called SCSI Parallel Interface or SPI), which uses a parallel bus design. Since 2005, SPI was gradually replaced by Serial Attached SCSI (SAS), which uses a serial design but retains other aspects of the technology. Many other interfaces which do not rely on complete SCSI standards still implement the SCSI command protocol; others drop physical implementation entirely while retaining the SCSI architectural model. iSCSI, for example, uses TCP/IP as a transport mechanism, which is most often transported over Gigabit Ethernet or faster network links.

SCSI interfaces have often been included on computers from various manufacturers for use under Microsoft Windows, classic Mac OS, Unix, Amiga and Linux operating systems, either implemented on the motherboard or by the means of plug-in adaptors. With the advent of SAS and SATA drives, provision for parallel SCSI on motherboards was discontinued.[12]

Parallel SCSI[edit]


Assorted Parallel SCSI connectors

Main article: Parallel SCSI

Initially, the *SCSI Parallel Interface* (SPI) was the only interface using the SCSI protocol. Its standardization started as a single-ended 8-bit bus in 1986, transferring up to 5 MB/s, and evolved into a low-voltage differential 16-bit bus capable of up to 320 MB/s. The last SPI-5 standard from 2003 also defined a 640 MB/s speed which failed to be realized.

Parallel SCSI specifications include several synchronous transfer modes for the parallel cable, and an asynchronous mode. The asynchronous mode is a classic request/acknowledge protocol, which allows systems with a slow bus or simple systems to also use SCSI devices. Faster synchronous modes are used more frequently.

USB

A USB is intended to enhance plug-and-play and allow hot swapping. Plug-and-play enables the operating system (OS) to spontaneously configure and discover a new peripheral device without having to restart the computer. As well, hot swapping allows removal and replacement of a new peripheral without having to reboot.

There are several types of USB connectors. In the past the majority of USB cables were one of two types, type A and type B. The USB 2.0 standard is type A; it has a flat rectangle interface that inserts into a hub or USB host which transmits data and supplies power. A keyboard or mouse are common examples of a type A USB connector. A type B USB connector is square with slanted exterior corners. It is connected to an upstream port that uses a removable cable such as a printer. The type B connector also transmits data and supplies power. Some type B connectors do not have a data connection and are used only as a power connection.

Today, newer connectors have replaced old ones, such as the Mini-USB (or Mini-B), that has been abandoned in favor of the Micro-USB and USB-C cables. Micro-USB cables are usually used for charging and data transfer between smartphones, video game controllers, and some computer peripherals. Micro-USB are being slowly replaced by type-C connectors, which are becoming the new standard for Android smartphones and tablets.

Techopedia Explains Universal Serial Bus

A Universal Serial Bus (USB) is basically a newer port that is used as a common interface to connect several different types of devices such as:

- Keyboards.
- Printers.
- Media devices.
- Cameras.
- Scanners.
- Mice.

It is designed for easy installation, faster transfer rates, higher quality cabling and hot-swapping. It has conclusively replaced the bulkier and slower serial and parallel ports.

The USB was co-invented and established by Ajay Bhatt, a computer architect who had been working for Intel. In 1994 seven companies that included Intel, Compaq, Microsoft, IBM, Digital Equipment Corporation (DEC), Nortel and NEC Corporation started the development of the USB.

Their objective was to make it easier to connect peripheral devices to a PC and eliminate the excessive amount of connectors. Factors involved included: creating larger bandwidths, streamlining software configurations and solving utilization problems for current interfaces.

The USB design is standardized by the USB Implementers Forum (USBIF) that is comprised of a group of companies supporting and promoting the USB. The USBIF not only markets the USB but maintains the specifications and upholds the compliance program. Specifications for the USB were created in 2005 with the 2.0 version. The standards were introduced by the USBIF in 2001; these included the older versions of 0.9, 1.0 and 1.1, which are backward compatible.

One of the greatest features of the USB is hot swapping. This feature allows a device to be removed or replaced without the past prerequisite of rebooting and interrupting the system. Older ports required that a PC be restarted when adding or removing a new device.

Rebooting allowed the device to be reconfigured and prevented electrostatic discharge (ESD), an unwanted electrical current capable of causing serious damage to sensitive electronic equipment such as integrated circuits.

Hot swapping is fault-tolerant, i.e. able to continue operating despite a hardware failure. However, care should be taken when hot-swapping certain devices such as a camera; damage can occur to the port, camera or other devices if a single pin is accidentally shorted.

Another USB feature is the use of direct current (DC). In fact, several devices use a USB power line to connect to DC current and do not transfer data. Example devices using a USB connector only for DC current include a set of speakers, an audio jack and power devices like a miniature refrigerator, coffee cup warmer or keyboard lamp.

USB Version 1 allowed for two speeds: 1.5 Mb/s (megabits per second) and 12 Mb/s, which work well for slow I/O devices. USB Version 2 allows up to 480 Mb/s and is backward compatible with slower USB devices. The first USB version 3 (USB 3.0 or SuperSpeed USB) was released in 2008, and allowed for a speed of 500 Mb/s. In 2013 and 2017, two new USB version 3 were released: USB 3.1 and USB 3.2, which allowed for 1.21 Gb/s and 2.42 Gb/s, respectively.

UNIT – VII

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

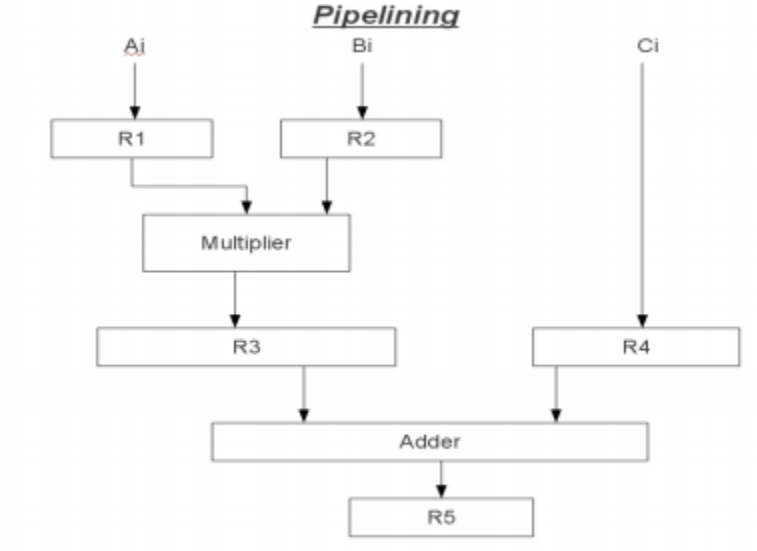
Basic concepts of pipelining:

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)



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

$$S = T_{seq} / T_{pipe} = n*m / (m+n-1)$$

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

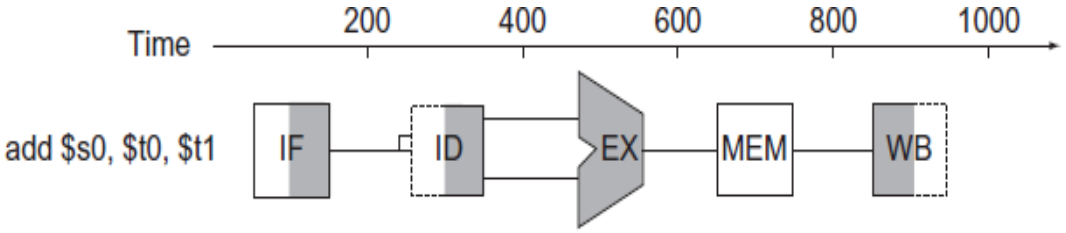


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to

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.

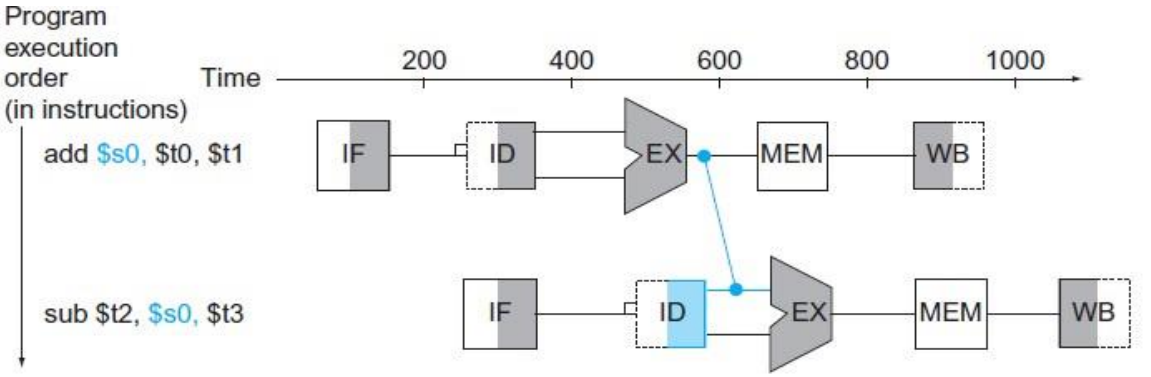


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path

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.

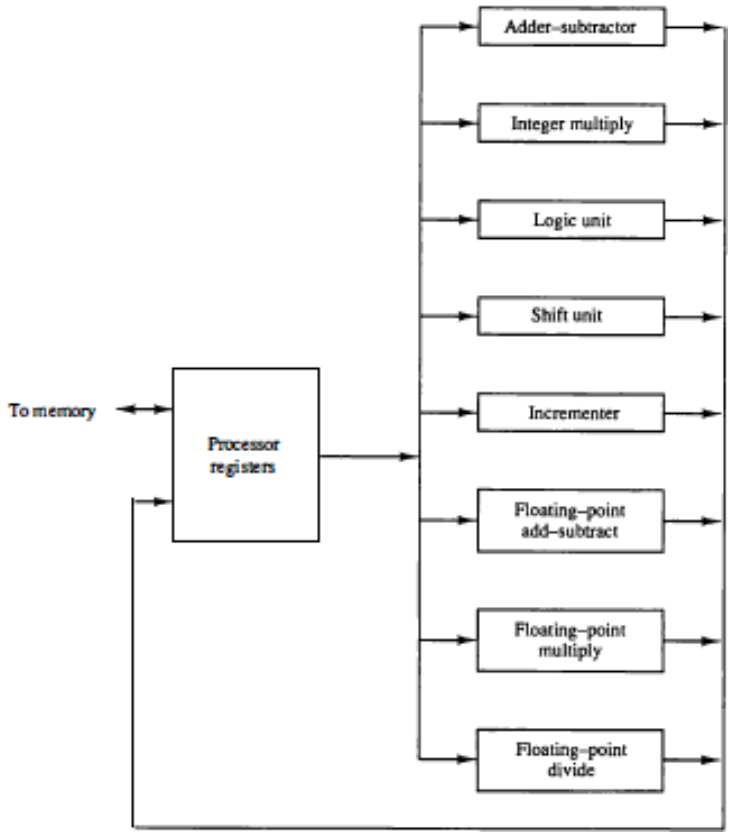
The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. 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.

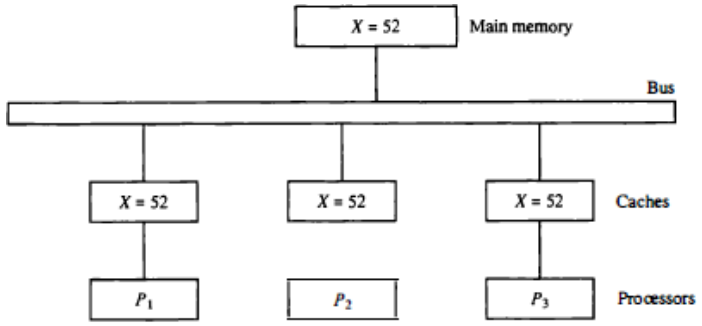
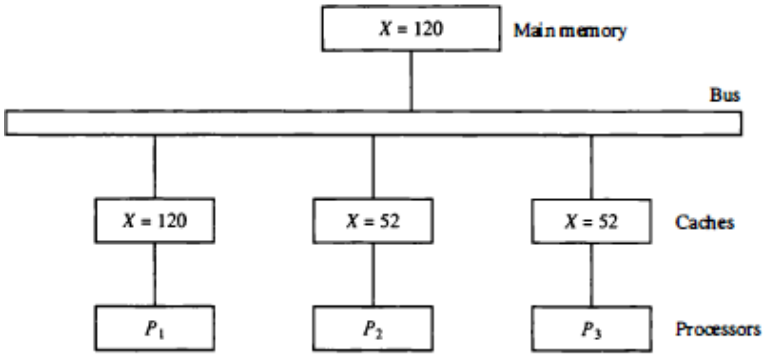


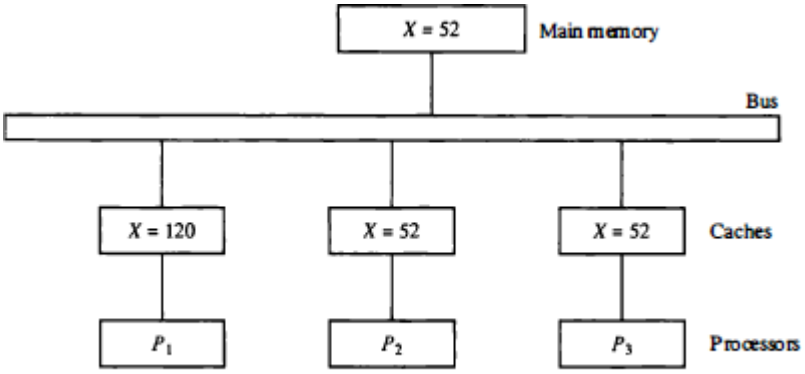
Figure 13-12 Cache configuration after a load on X.

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 P_1.



(a) With write-through cache policy



(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.