DIGITAL LOGIC AND COMPUTER ORGANIZATION

II B.TECH I SEMESTER - CSE (AR 23)

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

LENDI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institute, Approved by AICTE & Permanently Affiliated to JNTU-GV, Vizianagaram)
(Accredited By NAAC with A Grade and Accredited by NBA)
Jonnada (Village), Denkada (Mandal), Vizianagaram District – 535 005
Phone No. 08922-241111, 241112
E-Mail: lendi_2008@yahoo.com website: www.lendi.org

UNIT-IV

MACHINE INSTRUCTION AND I/O ORGANISATION: Component of Instructions: Logic Instructions, shift and Rotate Instructions, Type of Instructions: Arithmetic and Logic Instructions, Branch Instructions, Addressing Modes, Input/output Operations, Interrupts: Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Direct Memory Access.

INDEX

UNIT-IV

MACHINE INSTRUCTION AND I/O ORGANISATION: Component of Instructions: Logic Instructions, shift and Rotate Instructions Type of Instructions: Arithmetic and Logic Instructions, Branch Instructions, Addressing Modes, Input/output Operations, Interrupts: Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Direct Memory Access.

COMPONENT OF INSTRUCTIONS: LOGIC INSTRUCTIONS, SHIFT AND ROTATE INSTRUCTIONS

LOGIC INSTRUCTIONS:

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits. It is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

Not dst

2’s complement of a number can be obtained by adding 1 to the 1’s complement of the number. Consider the number to which we need to perform the 2’s complement is in register R0 and 2’s complement can be achieved by following instructions

```
Not   R0
Add   #1, R0
```

and in many computers have a single instruction that accomplish the same thing.

Negate R0

Now consider an application for the logic instruction AND, which performs the bit-wise AND operation on the source and destination operands. Suppose that four ASCII characters are contained in the 32-bit register R0. In some task, we wish to determine if the leftmost character is A. If it is, then a conditional branch to FOUNDA is to be made. ASCII code for A is 01000001, which is expressed in hexadecimal notation as 41H. The three-instruction sequence

```
And          #$FF000000, R0
Compare      #$41000000, R0
Branch =0    FOUNDA
```

implements the desired action. The And instruction clears all bits in the rightmost three character positions of R0 to zero, leaving the leftmost character unchanged. This is the result of using an immediate operand that has eight 1s at its right end, and 0s in the 24 bits to the right. The compare instruction compares the remaining character at the left end of R0 with binary representation for the character A. The Branch instruction causes a branch to FOUNDA if there is a match. The And instruction is used where all the bits of an operand except for some specified fields are to be cleared to 0.

SHIFT INSTRUCTIONS:

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a number, we use an arithmetic shift, which preserves the sign of the number.

Logical shifts:

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a logical left shift instruction is

LShiftL count, dst

The count operand may be given as an immediate operand, or it may be contained in a processor register. To complete the description of the shift left operation, we need to specify the bit values brought into the vacated positions at the right end of the destination operand, and to determine what happens to the bits shifted out of the left end. **Vacated positions are filled with zeros**. In computers that do not use condition code flags, the bits shifted out are simply dropped. In computers that use condition code flags, these bits are passed through the Carry flag, C, and then dropped. Involving the C flag in shifts is useful in performing arithmetic operations on large numbers that occupy more than one word.

The below figure shows an example of shifting the contents of register R0 left by two bit positions. The Logical-shift-right instruction, LShiftR, works in the same manner except that it shifts to the right.

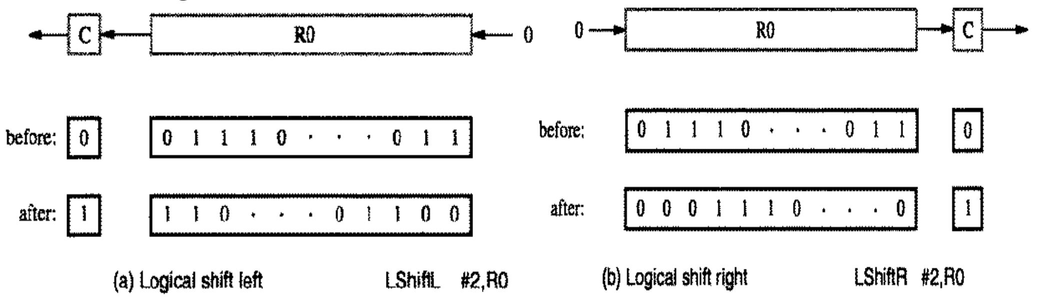


Figure: Logical shift instructions

Arithmetic Shifts

In an **arithmetic shift**, the bit pattern being shifted is **interpreted as a signed number**. Shifting a number one bit position to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shifting left, and the remainder is lost when shifting right. Another important note is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers. This requirement when shifting right distinguishes arithmetic shifts from logical shifts in which the fill-in bit is always 0. Otherwise, the two types of shifts are the same. An example of an Arithmetic shift- right instruction, AShiftR, is shown below. The Arithmetic-shift-left is exactly the same as the Logical-shift-left.

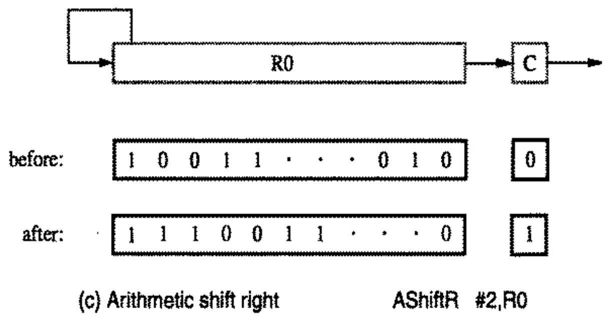


Figure: Arithmetic shift instructions

ROTATE INSTRUCTIONS:

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. To preserve all bits, a set of rotate instructions can be used. They move the bits that are shifted out of one end of the operand back into the other end. Two versions of both the left and right rotate instructions are usually provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. The OP codes RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations.

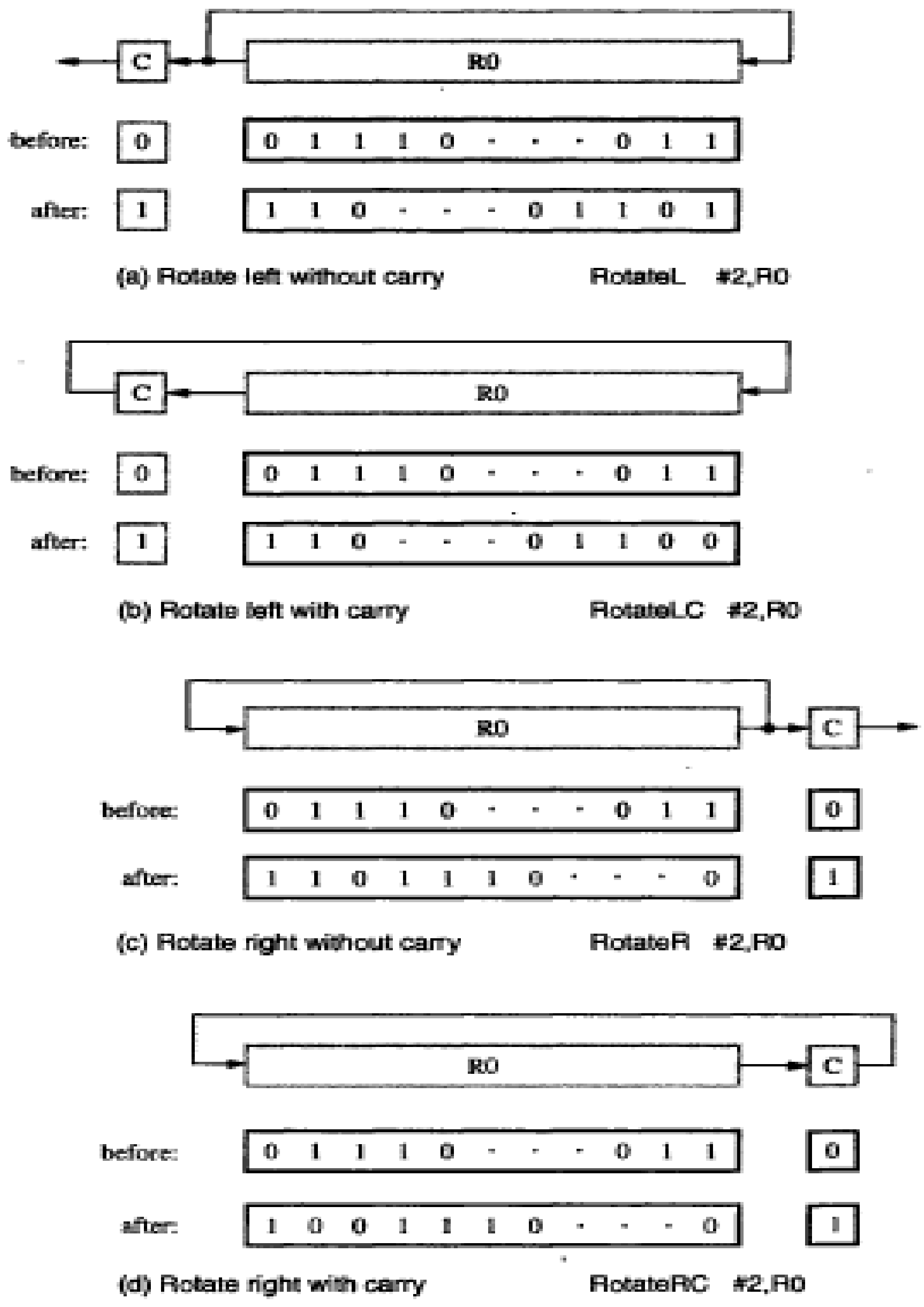

C
R0
before: 0 | 0 1 1 1 0 - - - 0 1 1
after: 1 | 1 1 0 - - - 0 1 1 0 1
(a) Rotate left without carry RotateL #2,R0
C
R0
before: 0 | 0 1 1 1 0 - - - 0 1 1
after: 1 | 1 1 0 - - - 0 1 1 0 0
(b) Rotate left with carry RotateLC #2,R0
R0
C
before: 0 1 1 1 0 - - - 0 1 1 | 0
after: 1 1 0 1 1 1 0 - - - 0 | 1
(c) Rotate right without carry RotateR #2,R0
R0
C
before: 0 1 1 1 0 - - - 0 1 1 | 0
after: 1 0 0 1 1 1 0 - - - 0 | 1
(d) Rotate right with carry RotateRC #2,R0

Figure: Rotate instructions

TYPE OF INSTRUCTIONS: ARITHMETIC AND LOGIC INSTRUCTIONS, BRANCH INSTRUCTIONS,

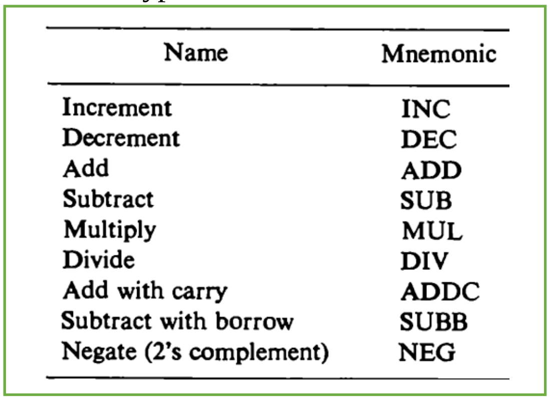
ARITHMETIC INSTRUCTIONS

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by means of software subroutines. The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of numerical analysis methods.

A list of typical arithmetic instructions is given below. The increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1's when incremented produces a result of all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces a number with all 1's.

Add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed ·to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

Table: Typical arithmetic instructions



Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

It is not uncommon to find computers with three or more add instructions: one for binary integers, one for floating-point operands, and one for decimal operands. The mnemonics for three add instructions that specify different data types are shown below.

ADDI	Add two binary integer numbers
ADDF	Add two floating- point numbers
ADDD	Add two decimal numbers in BCD

The number of bits in any register is of finite length and therefore the results of arithmetic operations are of finite precision. Some computers provide hardware double-precision operations where the length of each operand is taken to be the length of two memory words. Most small computers provide special instructions to facilitate double-precision arithmetic. A special carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed-2's complement form.

Multiplication and division instructions

Two signed integers can be multiplied or divided by machine instructions with the same format as like for an Add instruction. The instruction

Multiply Ri, Rj

performs the operation

$$Rj \leftarrow [Ri] \times [Rj]$$

The product of two n-bit numbers can be as large as 2n bits. Therefore, the answer will not necessarily fit into register Rj. A number of instruction sets have a multiply instruction that computes the low-order n bits of the product and places it in register Rj, as indicated. This is sufficient if it is known that all products in some particular application task will fit into n bits. To accommodate the general 2n-bit product case, some processors produce the product in two registers, usually adjacent registers Rj and R(j+ 1), with the high-order half being placed in register R(j + 1).

An instruction set may also provide a signed integer Divide instruction

Divide Ri, Rj

which performs the operation

$$Rj \leftarrow [Rj]/[Ri]$$

placing the quotient in Rj. The remainder may be placed in R(j + 1), or it may be lost.

LOGIC INSTRUCTIONS

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits. It is also useful to be able to perform logic operations in software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

And R4, R2, R3

computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4.

An immediate form of this instruction may be

And R4, R2, #Value

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16 most-significant bit positions.

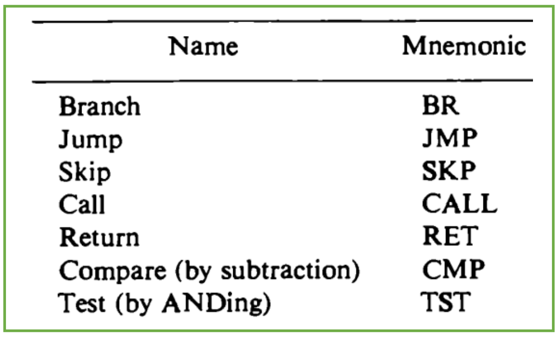
Consider the following application for this logic instruction. Suppose that four ASCII characters are contained in the 32-bit register R2. In some task, we wish to determine if the rightmost character is Z. If it is, then a conditional branch to FOUNDZ is to be made. We find that the ASCII code for Z is 01011010, which is expressed in hexadecimal notation as 5A. The three-instruction sequence

And R2, R2, #0xFF
Move R3, #0x5A

BRANCH or PROGRAM CONTROL INSTRUCTIONS

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence. After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence. On the other hand, **a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered.** In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations. The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments. Some typical program control instructions are listed in below table.

Table: Typical Program control instructions.



Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

The skip instruction does not need an address field and is therefore a zero-address instruction. **A conditional skip instruction will skip the next instruction if the condition is met.** This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a **skip-branch** pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

The **call and return** instructions are used in conjunction with subroutines. The **compare and test** instructions do not change the program sequence directly because of their application in setting conditions for subsequent conditional branch instructions. The **compare** instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. Similarly, the **test** instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition.

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. The below figure shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C. S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

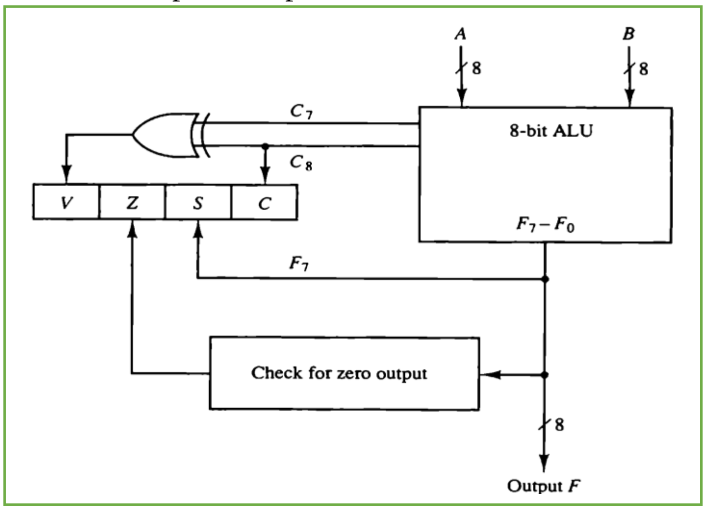


Figure: Status register bits

1. Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.

2. Bit S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.

3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.

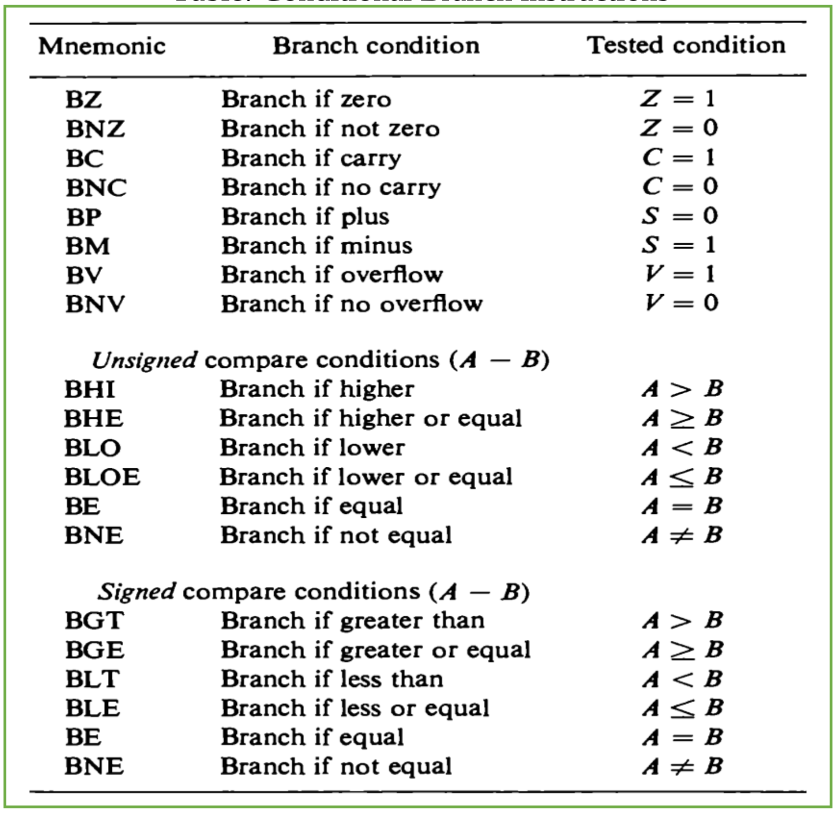
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A = B. This is so because $x \oplus x = 0$, and the exclusive-OR of two equal operands gives an all-0' s result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let A = 101x1100, where x is the bit to be checked. The AND operation of A with B = 00010000 produces a result 000x0000. If x = 0, the Z status bit is set, but if x = 1, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction if the original content of A must be preserved.

Conditional Branch Instructions

List of the most common branch instructions is given in the below table. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

Table: Conditional Branch instructions



Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
Unsigned compare conditions ($A - B$)		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

The zero status bit is used for testing if the result of an ALU operation is equal to zero or not. The carry bit is used to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the most significant bit of the output from the ALU. S=0 denotes a positive sign and S=1, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It must be realized, however, that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's complement representation.

The compare instruction performs a subtraction of two operands, say A - B. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides information about the relative magnitude of A and B. Some computers provide conditional branch instructions that can be applied right after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers A and B are considered to be unsigned or signed numbers. Note that we use the words higher and lower to denote the relations between unsigned numbers, and greater and less than for signed numbers.

ADDRESSING MODES

Addressing mode:

The different ways in which the location of an operand is specified in an instruction is called addressing mode. The term addressing mode refers to the way in which the operand of an instruction is specified.

The different addressing modes are listed in the below table:

NAME	ASSEMBLER SYNTAX	ADDRESSING FUNCTION
Immediate	#value	Operand = value
Register	R_i	EA = R_i
Absolute (Direct)	LOC	EA = LOC

Indirect	(R_i)
	(LOC)
		EA = [R_i]
		EA = [LOC]

Index	X(R_i)	EA = [R_i] + X
Base with index	(R_i, R_j)	EA = [R_i] + [R_j]
Base with index and offset	X(R_i, R_j)	EA = [R_i] + [R_j] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R_i)+	EA = [R_i]; increment R_i
Autodecrement	-(R_i)	Decrement R_i; EA = [R_i]

Variables and constants are the simplest data types used in computer programs. In assembly language, a variable is represented by allocating a register or memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

1. Register mode: The operand is the contents of a processor register; the name of register is given in the instruction

E.g. Move R1, R2; The contents of R1 is transferred to R2

2. Absolute mode (Direct mode): The operand is in a memory location; the address of this memory location is explicitly given in the instruction.

E.g. Add A, B; The contents of the memory location A is added to the contents of the memory location B. The addresses of A and B are given in the instruction itself.

3. Immediate mode: The operand is given explicitly in the instruction. This mode is used to specify the value of a source operand. # indicate that the value is to be used as an immediate operand.

E.g. Move #200, R0; This instruction places the value 200 in register R0.

Example: Write an assembly language program to perform A = B + 6

Program:
```
Move B, R1
Add #6, R1
Move R1, A
```

4. Indirect mode: Here the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand is determined.

The effective address of the operand is the contents of a register or main memory location, whose address appears in the instruction. We denote indirection by placing the name of the register or the memory address given in the instruction in parenthesis.

Let us consider two cases:

a) Add (R1), R0 – indirect addressing through a general purpose register
b) Add (A), R0 – indirect addressing through a memory location

In the first case, to execute the add instruction, the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds the contents of register R0.

In the second case, when the instruction is executed, CPU starts by fetching the contents of location A in the main memory. Since indirect addressing is used, the value B stored in A is not the operand, but the address of the operand. Hence CPU requests another read operation from the main memory and this is to read the operands (contents of location B). The CPU then adds the operand to the contents of R0

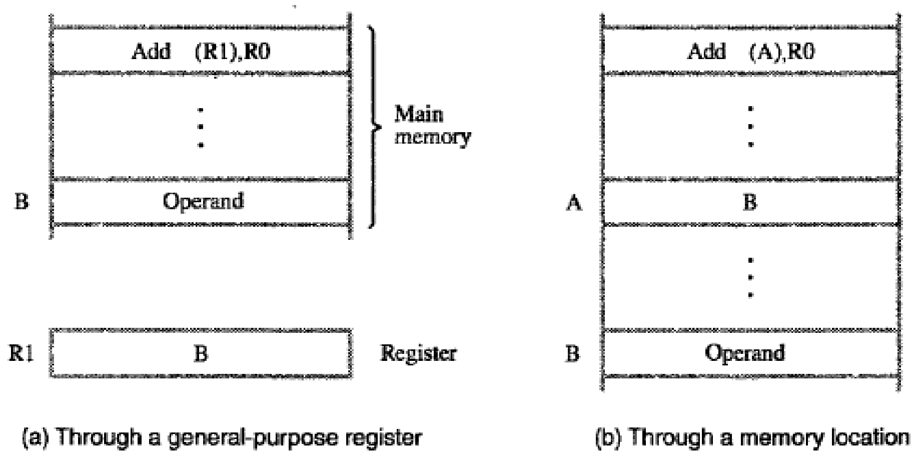

Add (R1),R0
Main memory
B
Operand
R1
B
Register
(a) Through a general-purpose register
Add (A),R0
A
B
B
Operand
(b) Through a memory location

Figure: Indirect addressing

The register or memory location that contains the address of the operand is called a pointer. Indirection is a powerful concept in programming.

5. Index mode: It is useful in dealing with lists and arrays. In this mode, the effective address of the operand is generated by adding a constant value to the contents of a register. The register used may be a special register provided for this purpose or may be any one of the general purpose register – referred to as an Index Register.

Index mode is indicated symbolically as **X(Ri**), where **X** denotes a constant value contained in the instruction and **Ri** is the name of the register involved. The effective address of the operand is given by **EA = X + [Ri].** The contents of the index register are not changed in the process of generating the effective address.

In assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value. There are two ways of using the index mode.

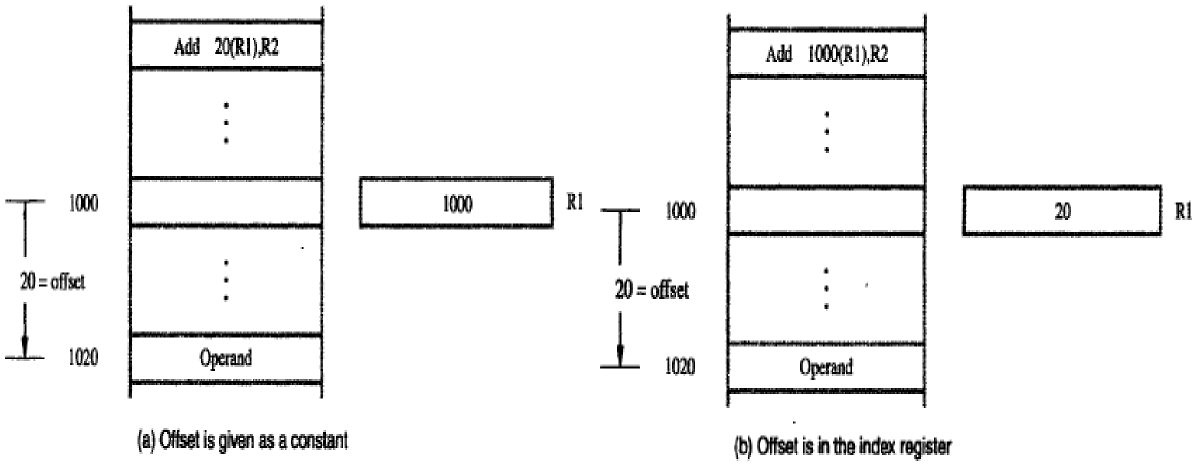
a. Offset is given as a constant: Here the index register R1 contains the address of a memory location and the value X defines an offset (displacement) from this address to the location where the operand is found.

Add 20(R1), R2

b. **Offset is in the index register**: Here the constant X corresponds to a memory address and the contents of the index register define the offset to the operand.

Add 1000(R1), R2

In either case, the effective address is the sum of two values; one is given explicitly in the instruction and the other is stored in a register.



Several variations are there in indexed addressing form provide very efficient access to memory operands in practical programming situation. For example, a second register may be used to contain the offset X, and can write the index mode as **(Ri, Rj)**. The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the **base register**. This form of indexed addressing provides more flexibility in accessing operands, because both the components of the effective address can be changed.

Another version of index mode uses two registers plus a constant, which can be denoted as **X(Ri, Rj).** In this case, the effective address is the sum of the constant X and the contents of the registers Ri and Rj. This mode implements a three dimensional array.

6. Relative mode: In index mode we use general purpose registers to obtain the effective address whereas program counter (PC) is used instead of the general purpose register Ri. **X(PC)** can be used to address a memory location that is X bytes away from the location presently pointed by the program counter. Since the addressed location is identified "relative" to the program counter, which always identifies the current execution point in a program, this mode is called as Relative mode. This mode is used to access data operands. It is commonly used to specify the target address in branch instructions.

Example: Instruction **Branch > 0 LOOP**

causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset address from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

7. Autoincrement mode: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of the register is automatically incremented to point to the next item in a list. We denote the autoincrement mode by putting the specified register in parenthesis to show that the contents of register is used as the effective address, followed by a plus (+) sign to indicate that these contents are to be incremented after the operand is accessed. Thus the autoincrement mode is written as **(Ri) +.** Implicitly, the increment amount is 1 when the mode is given in this form (increment is 1 for byte- sized operands, 2 for 16-bit operands, and 4 for 32-bit operands). If we use autoincrement mode, it is possible to eliminate the increment instruction

8. Autodecrement mode: The contents of a register specified in the instruction are first automatically decremented and then these contents used as the effective address of the operand. We denote the autodecrement mode by putting the specified register in the parenthesis, preceded by a minus (-) sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus we write **− (Ri).** This mode allows accessing of operands in descending address order.

The autoincremented and autodecremented modes are useful for accessing data items in successive locations in the memory.

BASIC INPUT/OUTPUT OPERATIONS

To transfer the data between the memory of a computer and the outside world, Input/output (I/O) operations are essential, and the way they are performed can have a significant effect on the performance of the computer.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as **program-controlled I/O**. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute many millions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

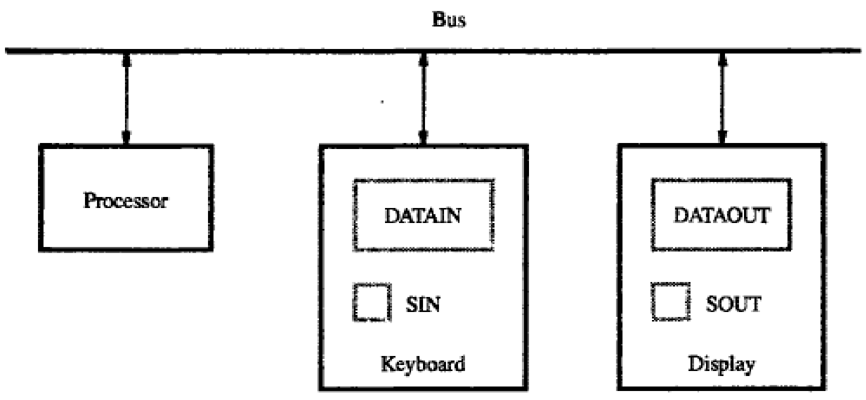


Figure: Bus connection for processor, keyboard, and display

A solution to this problem is, on output, the processor sends the first character and then waits for signal from the display that the character has been received. It then sends the second character, and so on. Input is sent from key board, the processor waits for the signal indicating that a character key has been struck and that its code is available in some buffer register associated with the key board. Then the processor proceeds to read that code.

The keyboard and the display are separate devices as shown in above figure. The action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Consider the problem of moving a character code from the keyboard to the processor. Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register DATAIN. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1. A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1, and the processor repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer. When SOUT equals 1, the display is ready to receive a character. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT. The transfer of a character to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1. The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part to circuitry commonly known as **device interface.**

Machine instructions required to perform basic I/O operations:

In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. These instructions are similar in format to those used for moving data between the processor and the memory. For example, the processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the following sequence of operations.

```
READWAIT   Branch to READWAIT if SIN = 0
            Input data from DATAIN to R1
```

The branch operation is usually implemented by two machine instructions. The first instruction tests the status flag and second performs the branch. Although the details vary from computer to computer, the main idea is that processor monitors the status flag by executing a short while loop and proceeds to transfer the input data when SIN is set to 1 as a result of a key being struck. The input operation resets SIN to 0.

An analogous process takes place when characters are transferred from the processor to the display.

```
WRITEWAIT    Branch to WRITEWAIT if SOUT = 0
             Output from R1 to DATAOUT
```

Again, the Branch operation is normally implemented by two machine instructions. The wait loop is executed repeatedly until the status flag SOUT is set to 1 by the display when it is free to receive a character. The output operation transfers a character from R1 to DATAOUT to be displayed, and it clears SOUT to 0.

In general, initial state of SIN is 0 and the initial state of SOUT is 1. This initialization is normally performed by the device control circuits when the devices are placed under computer control before program execution begins.

So far we assumed that the addresses issued by the processor to access instructions and operands always refer to memory locations. Many computers use an arrangement called **memory mapped I/O** in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT. Thus, no special instructions are needed to access the contents of these registers and the processor using instructions such as Move, Load, or Store.

For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

MoveByte DATAIN, R1

Similarly, the contents of register R1 can be transferred to DATAOUT by the instruction

MoveByte R1, DATAOUT

The status flags SIN and SOUT are automatically cleared when the buffer registers DATAIN and DATAOUT are referenced, respectively. The MoveByte operation code signifies that the operand size is a byte and Move has been used for word operands.

INTERRUPTS

The routine executed in response to an interrupt request is called the interrupt service routine, which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction ‘i’ in below figure.

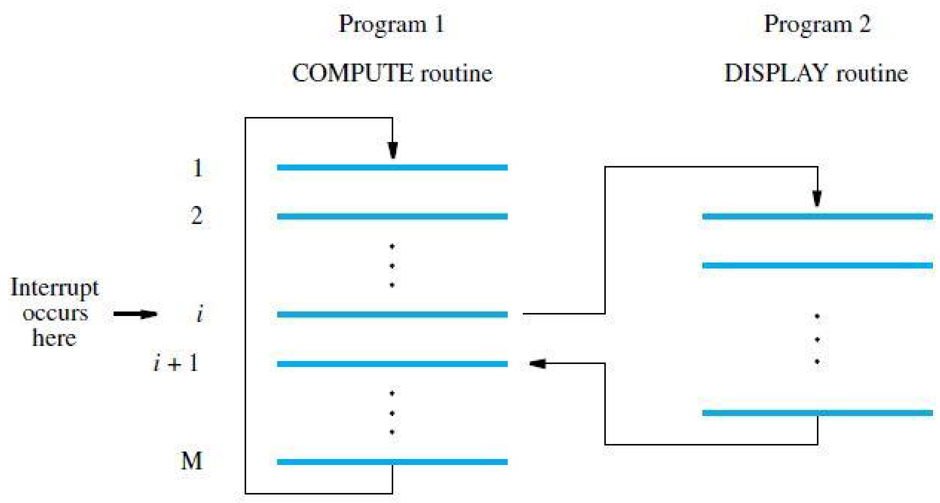


Fig: Transfer of control through the use of interrupts

The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction i +1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i+1, must be put in temporary storage in a known location. A Return-from interrupt instruction at the end of the interrupt-service routine reloads the PC from the temporary storage location, causing execution to resume at instruction i +1. In many processors, the return address is saved on the processor stack.

We should note that as **part of handling interrupts**, the processor must inform the device that its **request has been recognized** so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus. **An interrupt-acknowledge signal**. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs that device that its interrupt request has been recognized.

An interrupt-service routine is very similar to that of a subroutine. An important departure from this similarity should be noted. **A subroutine performs a function required by the program from which it is called**. However, the interrupt-service routine may not have anything in common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users. Therefore, before starting execution of the interrupt-service routine, any information that may be altered during the execution of that routine must be saved. This information must be restored before execution of the interrupt program is resumed. In this way, the original program can continue execution without being affected in any way by the interruption, except for the time delay. The information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine.

The task of saving and restoring information can be done automatically by the processor or by program instructions. Most modern processors save only the minimum amount of information needed to maintain the registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increase the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called **interrupt latency.**

INTERRUPT HARDWARE

An I/O device requests an interrupt by activating a bus line called **interrupt-request**. Most computers are likely to have several I/O devices that can request an interrupt. A single interrupt request line may be used to serve ‘n’ devices as shown in below figure. All the devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt- request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt request line will be equal to V_{dd}. This is the inactive state of the line. When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the interrupt request signal, INTR, received by the processor to go to 1. Since the closing of one or more switches will cause the line voltage to drop to 0, the value of INTR is the logical OR of the requests from individual devices, that is

$$\mathbf{INTR = INTR_1 + \ldots\ldots + INTR_n.}$$

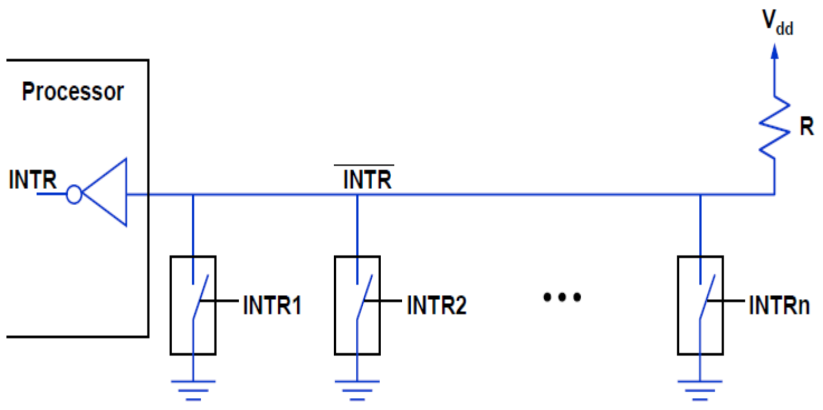


Fig: An equivalent circuit for open drain bus used to implement a common interrupt request line

It is customary to use the complemented form, $\overline{INTR}$, to name the interrupt-request signal on the common line, because this signal is active when in the low-voltage state.

In the electronic implementation of the circuit, a special gate known as open-collector (for bipolar circuits) or open-drain (for MOS circuits) are used to drive the $\overline{INTR}$ line. The output of the open-collector or an open-drain is equivalent to a switch to ground that is open when the gate's input is in the 0 state and closed when it is in the 1 state. The voltage level, hence the logic state, at the output of the gate is determined by the data applied to all the gates connected to the bus, according to equation given above. Resistor R is called a pull-up resistor because it pulls the line voltage up to the high-voltage when the switches are open.

ENABLING AND DISABLING INTERRUPTS

The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another. Because interrupts can arrive at any time, they may alter the sequence of events from that envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled. A fundamental facility found in all computers is the ability to enable and disable such interruptions as desired.

When a device activates interrupt signal line and waits with this signal activated until processors attends. The interrupt signal line is active during execution of ISR and till the device caused interrupt is serviced. It is essential to ensure that the active request signal does not lead to successive interruptions **(level -triggered input)** causing the system to fall in infinite loop from which it cannot recover.

Three methods of Controlling Interrupts (single device) are

1) Ignoring interrupt
2) Disabling interrupts
3) Special Interrupt request line

Ignoring Interrupts – Processor hardware ignores the interrupt request line until the execution of the first instruction of the ISR has been completed. Then by using an interrupt disable instruction after the first instruction of the ISR, the programmer can ensure that no further interrupts will occur until an interrupt enable instruction is executed. The processor must guarantee that execution of the return from interrupt instruction is completed before further interruptions can occur.

Disabling Interrupts – Processor automatically disables interrupts before starting the execution of the ISR. The processor saves the contents of PC and PS (processor status register) before performing interrupt disabling. The interrupt-enable is set to 0 then no further interrupts allowed. When return from interrupt instruction is executed the contents of the PS are restored from the stack, and the interrupt enable is set to 1.

Special Interrupt request line – Special interrupt request line for which the interrupt handling circuit responds only to the leading edge of the signal. Such a line is said to be **edge triggered**. Processor receives only one request, regardless of how long the line is activated. Hence, there is no danger of multiple interruptions and no need of explicitly disable interrupt requests from this line.

The sequence of events involved in handling an interrupt request from a single device. Assuming that interrupts are enabled, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge triggered interrupts) executed.
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt - request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request in interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions

1. How can the processor recognize the device requesting an interrupts?
2. Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

The means by which these problems are resolved vary from one computer to another, and the approach taken is an important consideration in determining the computer’s suitability for a given application.

When a request is received over the common interrupt-request line, additional information is needed to identify the particular device that activated the line.

The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it sets to 1 one of the bits in its status register, which we will call the IRQ bit. For example, bits KIRQ and DIRQ are the interrupt request bits for the keyboard and the display, respectively. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The first device encountered with its IRQ bit set is the device that should be serviced. An appropriate subroutine is called to provide the requested service.

The **polling scheme is easy to implement**. Its main disadvantage is the time spent interrogating the IRQ bits of all the devices that may not be requesting any service. An alternative approach is to use vectored interrupts, which we describe next.

Vectored Interrupts:

To reduce the time involved in the polling process, **a device requesting an interrupt may identify itself directly to the processor**. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term vectored interrupts refers to all interrupt-handling schemes based on this approach.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the starting address of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine. In many computers, this is done automatically by interrupt handling mechanism. The location pointed to by the interrupting device is used to store the starting address of the interrupt service routine. The processor reads this address, called **interrupt vector**, and loads it into the PC. The interrupt vector may also include a new value for the processor status register.

In most computers, I/O devices send the interrupt vector code over the data bus, using the bus control signals to ensure that devices do not interfere with each other. When a device sends an interrupt request, the processor may not be ready to receive the interrupt vector code immediately. For example, it must first complete the execution of the current instruction, which may require the use of the bus. There may be further delays if interrupts happen to be disabled at the time the request is raised. The interrupting device must wait to put data on the bus only when the processor is ready to receive it. When the processor is ready to receive the interrupt vector code, it activates the **interrupt acknowledge line**, INTA. The I/O device responds by sending its interrupt vector code and turning off the INTR signal.

Interrupt Nesting:

Interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, once started, always continues to completion before the processor accepts an interrupt request from a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices. For some devices, however, **a long delay in responding to an interrupt request may lead to erroneous operation.**

Consider, for example, a computer that keeps track of the time of day using a real-time clock. This is a device that sends interrupt requests to the processor at regular intervals. For each of these requests, the processor executes a short interrupt-service routine to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on. Proper operation requires that the delay in responding to an interrupt request from the real-time clock be small in comparison with the interval between two successive requests. To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept an interrupt request from the clock during the execution of an interrupt-service routine for another device. This example suggests that I/O devices should be organized in a priority structure.

An interrupt request from a high-priority device should be accepted while the processor is servicing another request from a lower-priority device. A **multiple-level priority organization** means that during execution of an interrupt-service routine, interrupt requests will be accepted from some devices but not from others, depending upon the **device's priority**. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own.

The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write into the PS. These are **privileged instructions**, which can be executed only while the processor is running in the **supervisor mode**. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. **An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a privileged exception**.

A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device, as shown in below figure. Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

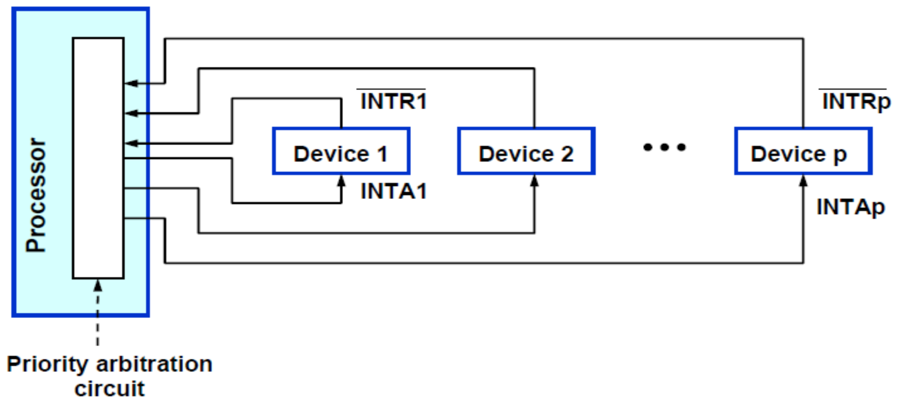

Processor
$\overline{\text{INTR1}}$
$\overline{\text{INTRp}}$
Device 1
Device 2
Device p
INTA1
INTAp
Priority arbitration circuit

Fig (a): Implementation of interrupt priority using individual interrupt request and acknowledge lines

Simultaneous Requests:

Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which requests to service first. Using a priority scheme the solution is straightforward. The processor simply accepts the requests having the highest priority.

Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a **daisy chain**, as shown below. The interrupt-request line INTR is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices raise an interrupt request and the INTR line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, **in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority.** The second device along the chain has second highest priority, and so on.

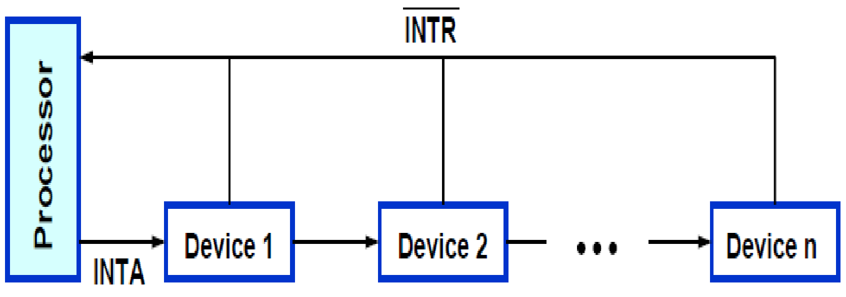


Fig (b): Daisy chain

The scheme in figure (b) requires considerably fewer wires than the individual connections in figure (a). The main advantage of the scheme in the figure (a) is that it allows the processor to accept interrupt requests from some devices but not from others, depending upon their priorities. The two schemes may be combined to produce the more general structure in figure (c). **Devices are organized in groups, and each group is connected at a different priority level. Within a group, devices are connected in a daisy chain.** This organization is used in many computer systems.

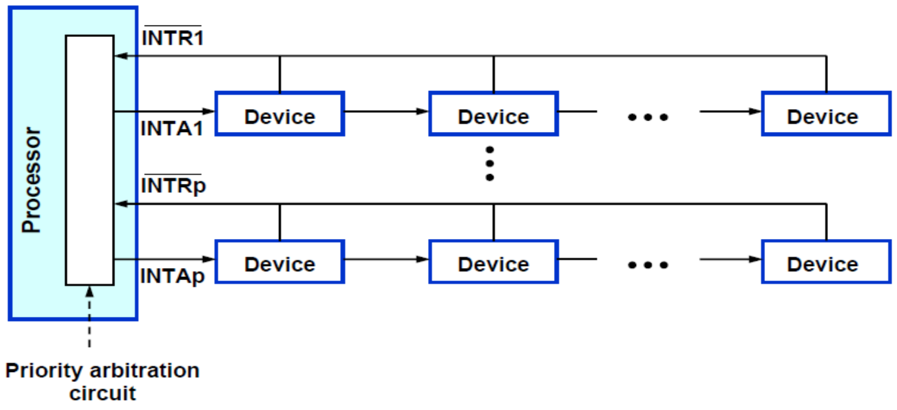

$\overline{\text{INTR1}}$
INTA1
$\overline{\text{INTRp}}$
INTAp
Processor
Device
Device
Device
Device
Device
Device
Priority arbitration circuit

Fig (c): Interrupt priority schemes

DIRECT MEMORY ACCESS

Basically for high speed I/O devices, the device interface transfer data directly to or from the memory without informing the processor. When interrupts are used, additional overhead involved with saving and restoring the program counter and other state information. To transfer large blocks of data at high speed, an alternative approach is used.

A special control unit may be provided to transfer a block of data directly between an I/O device and the main memory, **without continuous intervention by the processor**. This approach is called **direct memory access or DMA**. Control unit which performs these transfers is a part of the I/O device interface circuit. This control unit is called as a **DMA controller**. The DMA controller performs functions that would be normally carried out by the processor when accessing the main memory. For each word transferred, it provides the memory address and all the control signals. To transfer a block of data, it increments the memory addresses for successive words and keep track of the number of transfers.

DMA controller can transfer a block of data from an external device to the processor, without any intervention by the processor. However, the operation of the DMA controller must be under the control of a program executed by the processor. That is, the processor must initiate the DMA transfer. **To initiate the transfer of block of words, the processor sends the starting address, the number of words in the block and direction of transfer (I/O device to the memory, or memory to the I/O device).** Once the DMA controller completes the DMA transfer, it informs the processor by raising an interrupt signal.

While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer.

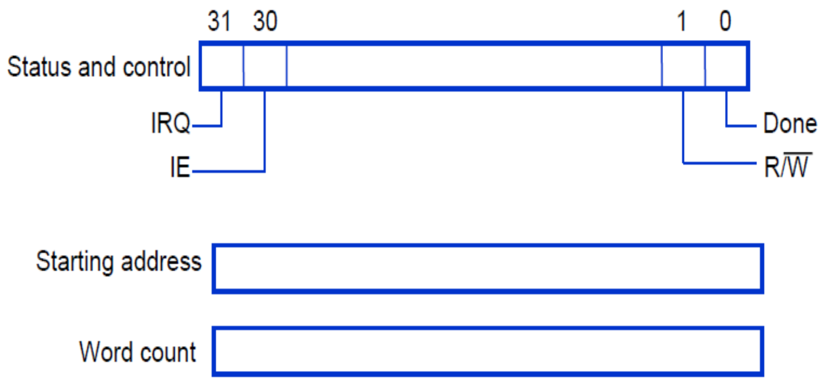


Fig: Registers in a DMA interface

The above figure shows an example of DMA controller registers that are accessed by the processor to initiate transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. **The R/W bit determines the direction of transfer.** When this bit is set by a program instruction, the controller performs a **read operation**, that is, it **transfers data from the memory to the I/O device**. Otherwise, it performs a write operation. When the controller has completed transferring a block of data and it is ready to receive another command, it sets the **Done flag** to 1. When IE is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. When IRQ bit sets to 1 by the controller when it has requested an interrupt.

DMA controller connects a high-speed network to the computer bus. The disk controller, which controls two disks also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk has its own DMA controller. The registers to store the memory address, word count and status and control information are duplicated, so that one set can be used with each device.

Memory access by the processor and DMA controllers are interwoven. Requests by DMA devices for using the bus are always given higher priority than the processor to access the bus. Among different DMA devices, top priority is given to high-speed peripherals such as a disk or a graphics display device. Since the processor originates most memory access cycles on the bus. **DMA controller can be said to "steal" memory access cycles from the processor. This interweaving technique is called as "cycle stealing**". Alternatively, the **DMA controller may be given exclusive access to main memory to transfer a block of data without interruption. This is known as block or burst mode.**

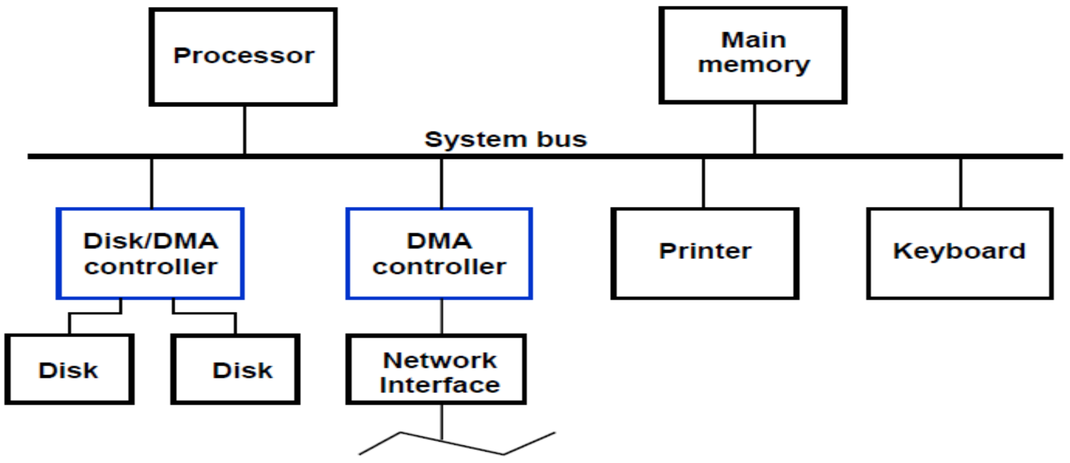


Fig: Use of DMA controllers in a computer system

A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory. Too resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.

BUS ARBITRATION

The device that is **allowed to initiate data transfers on the bus at any given time** is called the **bus master**. **Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it.** The selection of the bus master must take into account the needs of various devices by establishing priority system for gaining access to the bus. There are two approaches to bus arbitration: **centralized and distributed.**

Centralized Arbitration:

Here the **processor is the bus master** and it may grants bus mastership to one of its DMA controller. A DMA controller indicates that it needs to become the bus master by activating the **Bus Request line (BR)**. The signal on BR is the logical OR of the bus request from all devices connected to it. When BR is activated the processor activates **the Bus Grant Signal (BGI)** and indicated the DMA controller that they may use the bus when it becomes free. This signal is connected to all devices using a **daisy chain arrangement**. If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating **Bus Busy (BBSY).**

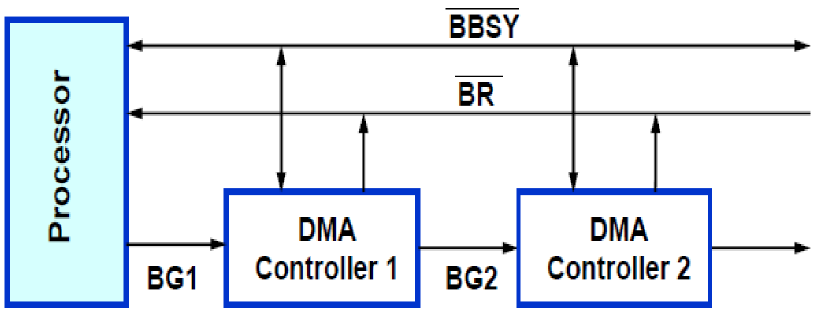


Fig: A simple arrangement for bus arbitration using a daisy chain

Distributed Arbitration:

It means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process. Each device on the bus is assigned a 4 bit ID. **When one or more devices request the bus, they assert the Start-Arbitration signal and place their 4 bit ID number on four open collector lines, ARB0 to ARB3.** A winner is selected as a result of the interaction among the signals transmitted over these lines. The net outcome is that the code on the four lines represents the request that has the highest ID number. The drivers are of open collector type. Hence, if the input to one driver is equal to 1, the input to another driver connected to the same bus line is equal to 0 (i.e. the bus is in low-voltage state).

Example: Assume two devices A and B have their ID 5 (0101), 6(0110) and their code is 0111. Each devices compares the pattern on the arbitration line to its own ID starting from MSB. If it detects a difference at any bit position, it disables the drivers at that bit position. It does this by placing 0 at the input of these drivers. In our example, 'A' detects a difference in line ARB1, hence it disables the drivers on lines ARB1 and ARB0. This causes the pattern on the arbitration line to change to 0110 which means that B has won the contention.

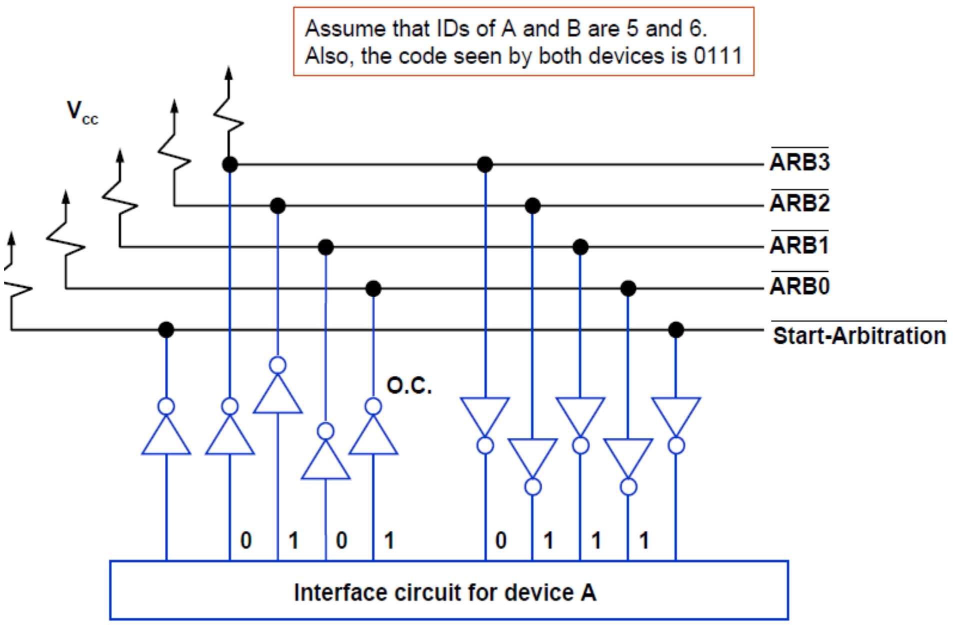

Assume that IDs of A and B are 5 and 6.
Also, the code seen by both devices is 0111
Vcc
ARB3
ARB2
ARB1
ARB0
Start-Arbitration
O.C.
0
1
0
1
0
1
1
1
Interface circuit for device A

Fig: Distributed arbitration scheme