

CS401 Short Notes Mid term

Lec 1 to8

BY vuonlinehelp.blogspot.com

YOUTUBE LINK:

WEBSITE LINK:

[HTTPS://VUONLINEHELP.BLOGSPOT.COM/](https://vuonlinehelp.blogspot.com/)

IMPORTANT NOTE

You should never copy paste the same answer which we are providing. Please do make some changes in the solution file. If you appreciate our work, please share our website vuonlinehelp.blogspot.com with your friends.

آپ کو کبھی بھی وہی جواب کاپی پیسٹ نہیں کرنا چاہیے جو ہم فراہم کر رہے ہیں۔ براہ کرم حل فائل میں کچھ تبدیلیاں کریں۔ اگر آپ ہمارے کام کی تعریف کرتے ہیں تو براہ کرم ہماری ویب سائٹ vuonlinehelp.blogspot.com کو اپنے دوستوں کے ساتھ شیئر کریں

Lecher 1 Introduction to Assembly Language

Address Bus Data Bus

Data Bus In computer architecture, the data bus is a wired connection dedicated for the transmitting the data between the CPU, peripheral devices and other hardware components. The data bus is a part of the system bus in addition to address bus and the control bus.

REGISTERS

A register is a piece of high-speed memory located directly on the processor. It is used to store data while the processor manipulates it. On the iAPX8088, there are 14 registers.

INSTRUCTION

A command that tells the processor to do something, like add two numbers or get some data from the memory. Some manufacturers name the mnemonics for data movement instructions as “move,” some call it “load” and “store” and still other names are present.

INTEL IAPX88 ARCHITECTURE

Technically iAPX88 stands for “Intel Advanced Processor Extensions 88.” It was a very successful processor also called 8088 and was used in the very first IBM PC machines. Our discussion will revolve around 8088 in the first half of the course while in the second half we will use iAPX386 which is very advanced and powerful processor. 8088 is a 16bit processor with its accumulator and all registers of 16 bits. 386 on the other hand, is a 32bit processor.

Lax 2 Addressing Modes

DATA DECLARATION

A data definition statement sets aside storage in memory for a variable, with an optional name. Data definition statements create variables based on intrinsic data types such as BYTE, WORD, DWORD SBYTE, SWORD etc. In Assembly language, there are no distinct data types like char,string,int,float,double,etc Instead there are very basic data types according to their sizes. They are: byte – 8 bits. word – 16 bits.

DIRECT ADDRESSING

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the

variables used in the program. Our numerical machine can only work with numbers. We used symbols for our ease to label or tag certain positions in our program. The assembler converts these symbols into the appropriate numbers automatically.

Size Mismatch Errors

In general assembly language gives a lot of power to the programmer but power comes with responsibility. Assembly language programming is not a difficult task but a responsible one. To keep the declarations and their access synchronized is the responsibility of the programmer and not the assembler. The assembler allows the programmer to do everything he wants to do, and that can possibly run on the processor. The assembler only keeps us from writing illegal instructions which the processor cannot execute. This is the difference between a syntax error and a logic error. So the assembler and debugger have both done what we asked them to do but the programmer asked them to do the wrong chore. The programmer is responsible for accessing the data as word if it was declared as a word and accessing it as a byte if it was declared as a byte.

REGISTER INDIRECT ADDRESSING

Register-Indirect addressing is perhaps the simplest of ARM's addressing modes. To load a value from memory into a register using register-indirect addressing, we use a second register, known as the base register. This base register holds the actual memory address that the program is interested in.

There are four registers in iAPX88 architecture that can hold address of data and they are BX, BP, SI, and DI.

REGISTER + OFFSET ADDRESSING

Direct addressing and indirect addressing using a single register are two basic forms of memory access. Another possibility is to use different combinations of direct and indirect references. In the above example we used BX to access different array elements which were placed consecutively in memory like an array.

SEGMENT ASSOCIATION

All the addressing mechanisms in iAPX88 return a number called effective address. For example in base + offset addressing, neither the base nor the offset alone tells the desired cell in memory to be accessed. It is only after the addition is done that the processor knows which cell to be accessed. This number which came as the result of addition is **called the effective address**.

ADDRESS WRAPAROUND

There are two types of wraparounds. One is within a single segment and the other is inside the whole physical memory. Segment wraparound occurs when during the effective address calculation a carry is generated. This carry is dropped giving the effect that when we try to access beyond the segment limit, we are actually wrapped around to the first cell in the segment.

ADDRESSING MODES SUMMARY

The iAPX88 processor supports seven modes of memory access. Remember that immediate is not an addressing mode but an operand type. Operands can be immediate, register, or memory. If the operand is memory one of the seven addressing modes will be used to access it. The memory access mechanisms can also be written in the general form “base + index + offset” Part of a register cannot be used to access memory. Like BX is allowed to hold an address but BL or BH are not. Address is 16bit and must be contained in a 16bit register. BX-SI is not possible. The only thing that we can do is addition of a base register with an index register. Any other operation is disallowed. BS+BP and SI+DI are both disallowed.

Lec 3 Branching

COMPARISON AND CONDITIONS

Conditional jump was introduced in the last chapter to loop for the addition of a fixed number of array elements. The jump was based on the zero flag. There are many other conditions possible in a program. For example an operand can be greater than another operand or it can be smaller. We use comparisons and Boolean expressions extensively in higher level languages. They must be available

is some form in assembly language, otherwise they could not possibly be made available in a higher level language. In fact they are available in a very fine and purified form.

What is conditional jump and unconditional jump?

Unconditional Jump Instructions: Transfers the program sequence to the described memory address. Conditional Jump Instructions Transfers the program sequences to the described memory address only if the condition is satisfied.

RELATIVE ADDRESSING

This is position relative addressing in contrast to absolute addressing. It is not telling the exact address rather it is telling how much forward or backward to go from the current position of IP in the current code segment.

TYPES OF JUMP

The three types of jump, near, short, and far, differ in the size of instruction and the range of memory they can jump to with the smallest short form of two bytes and a range of just 256 bytes to the far form of five bytes and a range covering the whole memory.

- Near Jump
- Short Jump
- Far Jump

Near Jump

When the relative address stored with the instruction is in 16 bits as in the last example the jump is called a near jump. Using a near jump we can jump anywhere within a segment

Short Jump

Conditional jumps can only be short. A short jump can go +127 bytes ahead in code and -128 bytes backwards and no more. This is the limitation of a byte in signed representation.

Far Jump

Far jump is not position relative but is absolute. Both segment and offset must be given to a far jump. The previous two jumps were used to jump within a segment. Sometimes we may need to go from one code segment to another, and near and short jumps cannot take us there. Far jump must be used and a two byte segment and a two byte offset are given to it. It loads CS with the segment part and IP with the offset part. Execution therefore resumes from that location in physical memory. The three instructions that have a far form are JMP, CALL, and RET, are related to program control. Far capability makes intra segment control possible.

Lec 4 Bit Manipulations

MULTIPLICATION ALGORITHM

A multiplication algorithm is **an algorithm (or method) to multiply two numbers**. Depending on the size of the numbers, different algorithms are used. Efficient multiplication algorithms have existed since the advent of the decimal system.

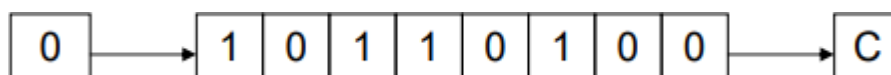
We take the first digit of the multiplier and multiply it with the multiplicand. As the digit is one the answer is the multiplicand itself. So we place the multiplicand below the bar. Before multiplying with the next digit a cross is placed at the right most place on the next line and the result is placed shifted one digit left.

SHIFTING AND ROTATIONS

The set of shifting and rotation instructions is one of the most useful set in any processor's instruction set. They simplify really complex tasks to a very neat and concise algorithm. The following shifting and rotation operations are available in our processor.

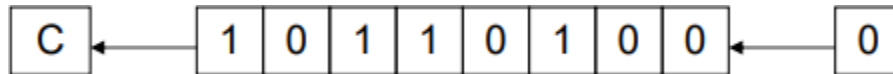
Shift Logical Right (SHR)

The shift logical right operation inserts a zero from the left and moves every bit one position to the right and copies the rightmost bit in the carry flag.



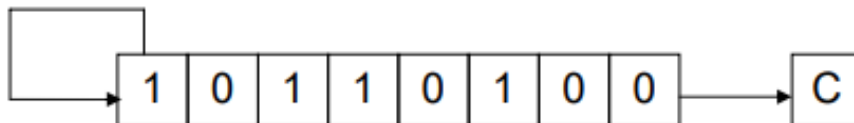
Shift Logical Left (SHL) / Shift Arithmetic Left (SAL)

The shift logical left operation is the exact opposite of shift logical right. In this operation the zero bit is inserted from the right and every bit moves one position to its left with the most significant bit dropping into the carry flag. Shift arithmetic left is just another name for shift logical left.



Shift Arithmetic Right (SAR)

A signed number holds the sign in its most significant bit. If this bit was one a logical right shifting will change the sign of this number because of insertion of a zero from the left. The sign of a signed number should not change because of shifting.



EXTENDED OPERATIONS

Assembly language only provides us the basic building blocks. We build a plaza out of these blocks, or a building, or a classic piece of architecture is only dependant upon our imagination. With our logic we can extend these algorithms as much as we want. Our next example will be multiplication of 16bit numbers to produce a 32bit answer.

BITWISE LOGICAL OPERATIONS

The 8088 processor provides us with a few logical operations that operate at the bit level. The logical operations are the same as discussed in computer logic design; however our perspective will be a little different. The four basic operations are AND, OR, XOR, and NOT.

AND operation

AND performs the logical bitwise and of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.

OR operation

A bit in the result is set if either or both corresponding bits in the original operands are set otherwise the result bit is cleared.

XOR operation

XOR (Exclusive Or) performs the logical bitwise “exclusive or” of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared) otherwise the result bit is cleared.

NOT operation

NOT inverts the bits (forms the one’s complement) of the byte or word operand. Unlike the other logical operations, this is a single operand instruction, and is not purely a logical operation in the sense the others are, but it is still traditionally counted in the same set.

MASKING OPERATIONS

Selective Bit Clearing Another use of AND is to make selective bits zero in its destination operand. The source operand is loaded with a mask containing one at positions which are retain their old value and zero at positions which are to be zeroed. **The effect of applying this operation on the destination with mask in the source is to clear the desired bits. This operation is called masking.**

Selective Bit Setting

The OR operation can be used as a masking operation to set selective bits. The bits in the mask are cleared at positions which are to retain their values, and are set at positions which are to be set.

Selective Bit Inversion

XOR can also be used as a masking operation to invert selective bits. The bits in the mask are cleared at positions, which are to retain their values, and are set at positions, which are to be inverted.

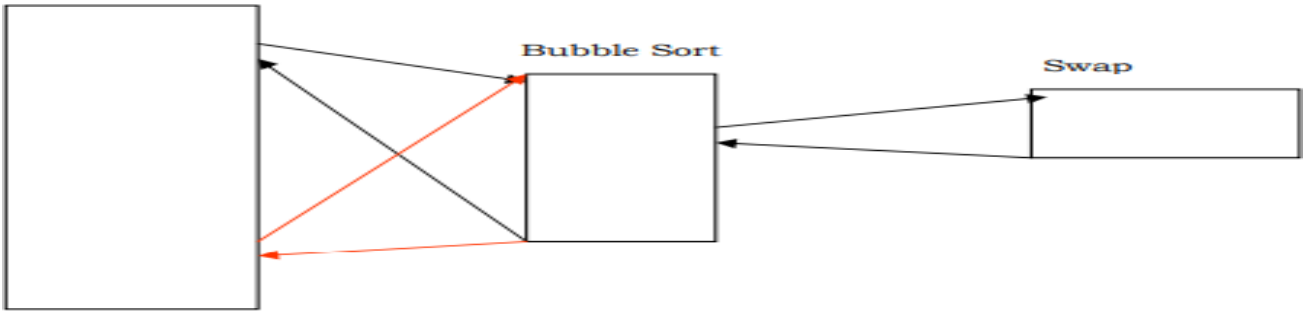
Selective Bit Testing

AND can be used to check whether particular bits of a number are set or not. Previously we used shifting and JC to test bits one by one. Now we introduce another way to test bits, which is more powerful in the sense that any bit can be tested anytime and not necessarily in order.

Lec 5 Subroutines

Program flow

Program flow is a general term which describes the order in which your lines of code are executed. This means that some lines will only be read once, some multiple times, and others may be skipped completely, depending on the situation.



CALL and RET

The instructions for permanent diversion in 8088 are the jump instructions, while the instruction for temporary diversion is the CALL instruction. The CALL instruction allows temporary diversion and therefore reusability of code. CALL takes a label as argument and execution starts from that label, until the RET instruction is encountered and it takes execution back to the instruction following the CALL.

Parameters

Such pieces of information that may change from invocation to invocation and should be passed from the caller to the subroutine are called parameters. There must be some way of passing these parameters to the subroutine. Revising the subroutine temporary flow breakage mechanism, the most straightforward way is to use registers. The CALL mechanism breaks the thread of execution and does not change registers, except IP which must change for processor to start executing at another place, and SP whose change will be discussed in detail later.

What is a subroutine?

In computer programming, a subroutine is a sequence of program instructions that performs a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.

STACK

Stack is a data structure that behaves in a first in last out manner. It can contain many elements and there is only one way in and out of the container. When an element is inserted it sits on top of all other elements and when an element is removed the one sitting at top of all others is removed first. The operation of placing an element **on top of the stack is called pushing the element** and the operation of removing an element from the top of the stack is called **popping the element**. The last thing pushed is popped out first; the last in first out behavior.

Apart from CALL and RET, the operations that use the stack are PUSH and POP. Two other operations that will be discussed later are INT and IRET. Regarding the stack, the operation of PUSH is similar to CALL however with a register other than the instruction pointer. For example “push ax” will push the current value of the AX register on the stack.

SAVING AND RESTORING REGISTERS

LONG

The subroutines we wrote till now have been destroying certain registers subroutines need to implement some mechanism of retaining the callers’ value of any registers used. The trick is to use the PUSH and POP operations and save the callers’ value on the stack and recover it from there on return. Our swap subroutine destroyed the AX register while the bubble sort subroutine destroyed AX, CX, and SI. BX was not modified in the subroutine.

Inside the debugger we can observe that the registers before and after the CALL operation are exactly identical. We can observe how their value is saved on the stack by the PUSH instructions and recovered from their before exit. Saving and restoring registers this way in subroutines is a standard way and must be followed.

PUSH

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

CALL

An assembly language instruction telling the assembler to perform the subroutine CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. For an intra-segment direct CALL, SP is decremented by two and IP is pushed onto the stack.

RET

important short

At the end of the subroutine, a RET instruction pops the old value of IP from the stack into the instruction pointer, and execution resumes right where the CALL instruction left off. RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RETF (inter segment RET) is used the word at the top of the stack is popped into the IP register and SP is incremented by two. The word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP.

PARAMETER PASSING THROUGH STACK

To pass parameters to a subroutine, the calling program pushes them on the stack in the reverse order so that the last parameter to pass is the first one pushed, and the first parameter to pass is the last one pushed. This way the first parameter is on top of the stack and the last one is at the bottom of the stack.

What is local variable in computer?

A variable whose scope is confined to a particular unit of code, such as module-level code, or a procedure. A local variable is a variable that is only accessible within a specific part of a program. These variables are usually local to a subroutine and are declared or defined within that routine. Parameters that are defined by value can also be considered as local variables.

Lec 6 Display Memory

ASCII CODES

The computer listens, sees, and speaks in numbers. Even a character is a number inside the computer. For example the keyboard is labeled with characters however when we press 'A', a specific number is transferred from the keyboard to the computer. When the same number comes on display, the Video Graphics Adapter (VGA) in our computer shows the shape of 'A'. Even the shape is stored in binary numbers with a one bit representing a pixel on the screen that is turned on and a zero bit representing a pixel that is not glowing.

An 'A' on any computer and any operating system is an 'A' on every other computer and operating system. This is because a standard numeric representation of all commonly used characters has been developed. This is called the ASCII code, where ASCII stands for American Standard Code for Information Interchange.

Standard ASCII has 128 characters with numbers assigned from 0 to 127. When IBM PC was introduced, they extended the standard ASCII and defined 128 more characters. Thus extending the total number of symbols from 128 to 256 numbered from 0 to 255 fitting in an 8-bit byte. The newer characters were used for line drawing, window corners, and some non-English characters. The need for these characters was never felt on teletype terminals, but with the advent of IBM PC and its full screen display, these semi-graphics characters were the need of the day.

DISPLAY MEMORY FORMATION

LONG

When 0x40 is sent to the VGA card, it will turn pixels on and off in such a way that a visual representation of 'A' appears on the screen. The video device is seen by the computer as a memory area containing the ASCII codes that are currently displayed on the screen and a set of I/O ports. The VGA memory is seen by the computer just like its own memory. ASCII of 'A' is placed somewhere in that block, the shape of 'A' will appear on the screen at a corresponding place. This correspondence must be defined as the memory is a single dimensional space while the screen is two dimensional having 80 rows and 25 columns. The memory is linearly mapped on this two dimensional space, just like a two dimensional is mapped in linear memory. There is one word per character in which a byte is needed for the ASCII code and the other byte is used for the character's attributes discussed later. Now the first 80 words will correspond to the first row of the screen and the next 80 words will correspond to the next row.

The three important things that we discussed are.

- One screen location corresponds to a word in the video memory
- The video controller memory is accessible to the processor like its own memory.

- ASCII code of a character placed at a cell in the VGA memory will cause the corresponding ASCII shape to be displayed on the corresponding screen location.

HELLO WORLD IN ASSEMBLY LANGUAGE

To declare a character in assembly language, we store its ASCII code in a byte. The assembler provides us with another syntax that doesn't force us to remember the ASCII code. The assembler also provides a syntax that simplifies declaration of consecutive characters, usually called a string. The three ways used below are identical in their meaning.

```
db 0x61, 0x62, 0x63
db 'a', 'b', 'c'
db 'abc'
```

When characters are stored in any high level or low level language the actual thing stored in a byte is their ASCII code. The only thing the language helps in is a simplified declaration. Traditionally the first program in higher level languages is to print “hello world” on the screen.

Lec 7 String Instructions

String instructions

String instructions work on a block of data. A block has a start and an end. The instructions can work from the start towards the end and from the end towards the start. In fact they can work in both directions, and they must be allowed to work in both directions otherwise certain operations with overlapping blocks become impossible. This problem is discussed in detail later. The direction of movement is controlled with the Direction Flag (DF) in the flags register. If this flag is cleared the direction is from lower addresses towards higher addresses and if this flag is set the direction is from higher addresses to lower addresses. If DF is cleared, this is called the autoincrement mode of string instruction, and if DF is set, this is called the autodecrement mode. There are two instructions to set and clear the direction flag.

- `cld` ; clear direction flag

- std ; set direction flag

The operation of each of the string instructions and each of the repetition prefixes is discussed below.

STOS

STOS transfers a byte or word from register AL or AX to the string element addressed by ES:DI and updates DI to point to the next location. STOS is often used to clear a block of memory or fill it with a constant.

LODS

LODS transfers a byte or word from the source location DS:SI to AL or AX and updates SI to point to the next location. LODS is generally used in a loop and not with the REP prefix since.

SCAS

SCAS compares a source byte or word in register AL or AX with the destination string element addressed by ES:DI and updates the flags. DI is updated to point to the next location. SCAS is often used to locate equality or in-equality in a string through the use of an appropriate prefix.

MOVS

MOVS transfers a byte or word from the source location DS:SI to the destination ES:DI and updates SI and DI to point to the next locations. MOVS is used to move a block of memory.

CMPS

CMPS subtracts the source location DS:SI from the destination location ES:DI. Source and Destination are unaffected. SI and DI are updated accordingly. CMPS compares two blocks of memory for equality or inequality of the block. It subtracts byte by byte or word by word. If used with a REPE or a REPNE prefix it repeats as long as the blocks are same or as long as they are different. For example it can be used to find a substring.

REP Prefix

REP repeats the following string instruction CX times. The use of CX is implied with the REP prefix. The decrement in CX doesn't affect any flags and the jump is also independent of the flags, just like JCXZ.

REPE and REPNE Prefixes

REPE or REPZ repeat the following string instruction while the zero flag is set and REPNE or REPNZ repeat the following instruction while the zero flag is not set. REPE or REPNE are used with the SCAS or CMPS instructions.

LODS EXAMPLE – STRING PRINTING

The use of LODS with the REP prefix is not meaningful as only the last value loaded will remain in the register. It is normally used in a loop paired with a STOS instruction to do some block processing. We use LODS to pick the data, do the processing, and then use STOS to put it back or at some other place. For example in string printing, we will use LODS to read a character of the string, attach the attribute byte to it, and use STOS to write it on the video memory.

SCAS EXAMPLE – STRING LENGTH

Many higher level languages do not explicitly store string length; rather they use a null character, a character with an ASCII code of zero, to signal the end of a string. In assembly language programs, it is also easier to store a zero at the end of the string. We use SCASB with REPNE and a zero in AL to find a zero byte in the string. In CX we load the maximum possible size, which is 64K bytes regarding SCAS and CMPS is that if they stop due to equality or inequality, the index registers have already incremented. Therefore when SCAS will stop DI would be pointing past the null character.

LES and LDS Instructions

String instructions need their source and destination in the form of a segment offset pair, there are two special instructions that load a segment register and a general purpose register from two consecutive memory locations. LES loads ES while LDS loads DS. These instructions have two parameters, one is the general purpose register to be loaded and the other is the memory location from which to load these registers. The major application of these instructions is when a subroutine receives a segment offset pair as an argument and the pair is to be loaded in a segment and an offset register. According to Intel rules of significance the word at higher address is loaded in the segment register while the word at lower address is loaded in the offset register.

MOVS EXAMPLE – SCREEN SCROLLING

MOVS has the two forms MOVSB and MOVSW. REP allows the instruction to be repeated CX times allowing blocks of memory to be copied. We will perform this copy of

the video screen. Scrolling is the process when all the lines on the screen move one or more lines towards the top or towards the bottom and the new line that appears on the top or the bottom is cleared. Scrolling is a process on which string movement is naturally applicable. REP with MOVS will utilize the full processor power to do the scrolling in minimum time.

CMPS EXAMPLE – STRING COMPARISON

The subroutine will take two segment offset pairs containing the address of the two null terminated strings. The subroutine will return 0 if the strings are different and 1 if they are same. The AX register will be used to hold the return value. Inside the debugger we observe that REPE is shown as REP. This is because REP and REPE are represented with the same prefix byte. When used with STOS, LODS, and MOVS it functions as REP and when used with SCAS and CMPS it functions as REPE.

Lec 8 Software Interrupts

INTERRUPTS

Interrupts in reality are events that occurred outside the processor and the processor must be informed about them. Interrupts are asynchronous and unpredictable. Asynchronous means that the interrupts occur, independent of the working of the processor, i.e. independent of the instruction currently executing. Synchronous events are those that occur side by side with another activity. Interrupts must be asynchronous as they are generated by the external world which is unaware of the happenings inside the processor. The software interrupt mechanism in 8088 uses vectored interrupts meaning that the address of the interrupt routine is not directly mentioned in an interrupt call, rather the address is lookup up from a table. 8088 provides a mechanism for mapping interrupts to interrupt handlers. Introducing a new entry in this mapping table is called hooking an interrupt.

We have discussed three things till now.

- The INT and IRET instruction format and syntax
- The formation of IVT (interrupt vector table)
- Operation of the processor when an interrupt is generated

HOOKING AN INTERRUPT

To hook an interrupt we change the vector corresponding to that interrupt. As soon as the interrupt vector changes, that interrupt will be routed to the new handler. Our first example is with the divide by zero interrupt. The normal system defined behavior in response to divide by zero is to display an error message and terminate the program. We will change it to display our own message.

BIOS AND DOS INTERRUPTS

In IBM PC there are certain interrupts designated for user programs to communicate with system software to access various standard services like access to the floppy drive, hard drive, vga, clock etc. Since the manufacturer knows the hardware it burns the software to control its hardware in ROM. Such software is called firmware and access to this firmware is provided through specified interrupts. **This basic interface to the hardware is called BIOS (basic input output services).** When the computer is switched on, BIOS gets the control at a specified address. The messages at boot time on the screen giving BIOS version, detecting different hardware are from this code. BIOS has the responsibility of testing the basic hardware including video, keyboard, floppy drive, hard drive etc and a special program to bootstrap. Bootstrap means to load OS from hard disk and from there OS takes control and proceeds to load its components and display a command prompt in the end.

There are two important programs; BIOS and OS. OS services are high level and build upon the BIOS services. BIOS services are very low level. A level further lower is only directly controlling the hardware. BIOS services provide a hardware independent layer above the hardware and OS services provide another higher level layer over the BIOS services. We have practiced direct hardware access with the video device directly without using BIOS or DOS. The layer of BIOS provides services like display a character, clear the screen, etc. All these layers are optional in that we can skip to whatever lower layer we want.