Intel Assembly Language Programming Via NASM

Programming in assembly language deals with hardware details of a given computer. Different hardware systems lead to different assembly language abstractions, and hence assembly languages. In this course, we use Intel-based hardware systems distinguished by their ‘family-oriented’ processor architectures. The machine language of the Intel-family is upward compatible: from the early days of the 8080 to the recent days of the Pentium, upward compatibility is maintained by augmenting the original 8080 design with extensions that do not conflict with earlier generations. The machine language compatibility facilitates portability of executable program codes across different hardware platforms of the same family.

Even for a same Intel-family, different assembly languages do exists. Differences of assembly languages arise when software (operating systems) platforms change, for example, from a WINDOWS environment to a Linux environment. These differences are minor in nature but may have rather different appearances. It is important to recall that the main difference between an assembly language (assembly/translator dependent) and a machine language (hardware dependent) lies in the following:

- The former uses symbols to encode operations and operands, whereas the latter uses binary bits. Hence the former is for human use and the latter for hardware use.

  Example:

  ```
  (Assembly language form)        add   ecx, [ebx]
  (Machine language form)         03    C0    1B    (hex)
  ```

Hence the processor will be fetching and executing instructions such as the above (03C01B h) and according to the design specification.
It is obviously that symbolic representation is easier for the programmer to program with, and leave the assembler to take care of the binary equivalent of the resulting program. In order to do such translation, the manufacturer handbook on the processor instruction format is the key reference. We will not consider these details here.

- An assembly language program may include some non-hardware instructions, called ***assembly directives and pseudo-ops***. These are extra 'commands' executed by the software translator (assembler) for facilitating its translation of the assembly language source code into the machine language equivalent.

  Example: *length* equ 20

  The above directs the assembler to give a value of 20 to every appearance of the symbol *length* in the assembly language code. The opcode 'equ' is a pseudo-op, it is performed by the assembler during translation time, rather than by the processor when the translated program is executed. There are other examples that we will go through later when we discuss the NASM assembler.

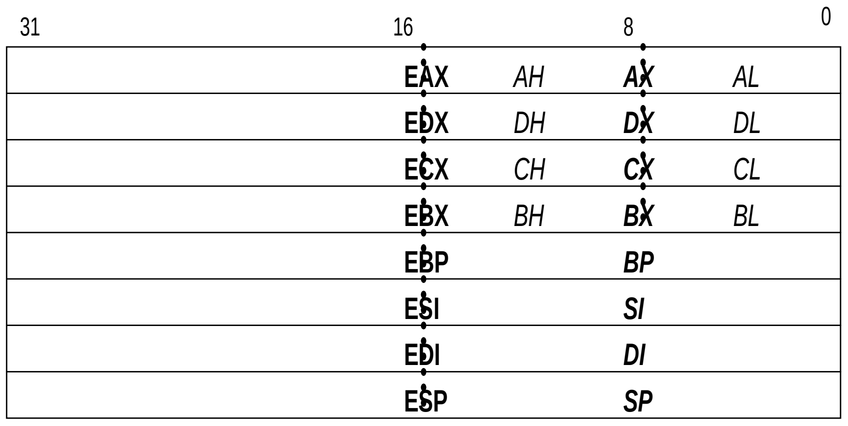
3.1 Basic Elements in Instruction Set Repertoire

It is necessary to learn about the hardware (processor) architecture before picking up the details of an assembly language based on that architecture. The following details of the Intel-based processor are relevant. For the purpose of this course, we will present only the features that are needed for application programmers. In fact, a simplified view of the 32-bit programming architecture will be used wherever possible.

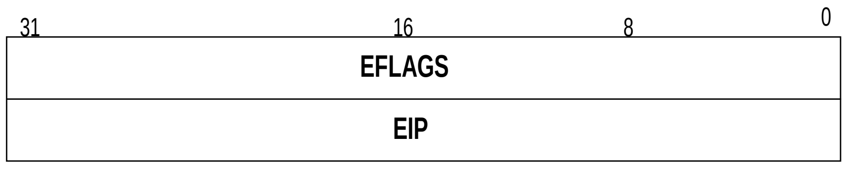
A processor is distinguished by its instruction set, or set of machine (binary coded) instructions that the processor understands and can perform (in replacing the human). This set is usually rather small (possibly a couple of hundred instructions). We will learn the more common ones needed in writing assembly/machine language programs.

3.1.1 *Programmable Architecture: What a Programmer Sees*

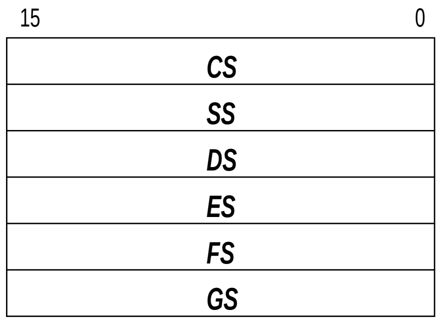
As noted before, an assembly (and machine) language programmer has the privilege of using registers as temporary buffers of program variables to enhance performance through the principle of locality. In the case of the 80xxx family of processors, there are three distinct sets of registers serving different functionalities.



Eight 32-bit General Registers



Two Special Purpose Registers



Six Segment Registers

3.1.2 General Registers eax, edx, ecx, ebx, ebp, esi, edi, esp

These registers can contain operand or operand addresses, hence the name general (purpose) register. Each is 32-bit wide. Each lower half (the less significant 16 bits) is given a distinct name (for 16 bit manipulations): ax, dx, cx, bx, bp, si, di, sp respectively. In turn, the first four registers are further partitioned into two smaller registers that can be used independently of the rest (for 8 bit manipulations).

Specifically: ax = ah : al (: denotes concatenation)
dx = dh: dl
cx = ch: cl
bx = bh: bl

Hence, bits 0 to 7 (the least significant byte) of eax form (sub-) register al, bits 8 to 15 form (sub-) register ah, and bits 0 to 15 form (sub-) register ax. These separately named (sub-) registers can be used in programming as if they are independent registers. Hence they provide flexibility in choosing between a byte-operand, word (2-byte) operand, and double-word (4-byte) operand. The notion of 'word' can be viewed as a packet consisting of multiple bytes that will be processed together. Usually a processor defines its word size. For example, here a word is 2 byte.

For example, we can choose to use:

- ax to hold an integer of 16 bits
- ah to hold a single ASCII coded character
- eax to hold an integer of 32 bits (hence double word)
- eax to hold a character string of 4 ASCII characters

For this course, you can ignore the segment registers.

The following summarizes how an effective memory address for data (source or destination operand) can be computed:

An operand address in an instruction is specified by at least one of the following fields:

Base:	{eax, ecx, edx, ebx, ebp, esi, edi, esp}
Index * scale:	{eax, ecx, edx, ebx, ebp, esi, edi} * {1, 2, 4, 8} (Scale is a multiplier that can be specified, with a default value of 1. It is useful to support 1 byte, 2 byte, 4 byte or 8 byte operand containers.)
Displacement:	0 or 8 or 32 bit displacement specified.

Examples:

[ebx+esi*2]:	access at offset given by ebx+esi*2. Here ebx serves as the base, and esi the index to be scaled by a factor of 2, similar to the based index addressing shown earlier. Suppose ebx points to the starting address of a data array of integers. Each integer is 2 bytes. Then esi is the index used to refer to a particular element of the array. The scale factor of 2 is used so as to accommodate the size of each element (integer) is 2 bytes.
[ebp – 4]:	access at offset given by ebp – 4 (note: –4 is specified as the displacement.)
[schar]:	access at offset equivalent to the symbolic address schar. If the assembler has mapped (allocated) address 100000h to schar, then this is equivalent to offset 100000h.

3.2 Instructions

An instruction contains two types of information:

- Operation to be performed (**Opcode**)
- Operands or their addresses (**Addressing modes** and **Addresses**)

Opcodes can be broadly classified into three major types:

- Data movement: moves data around within the computer system (within processor/memory, between processor and memory)
- Arithmetic and Logic: performs basic arithmetic and logic operations on operands
- Control flow: changes the control flow from strictly sequential to a specified instruction conditionally (depending on status register) or unconditionally

We will study these three general types of instructions (applicable to all computers) and their machine specific details, including the addressing modes and use restrictions.

3.2.1 Data movement operations

These instructions move data between storages, such as memory locations and registers, depending on the addressing modes used. In assembly language, an instruction is symbolically represented. Its counterpart in machine language, the instruction is encoded in binary (or equivalently, in hexadecimal for the human).

Example:

Assembly form:	mov ax, bx
Machine language:	89 D8h

The translation from an executable assembly language instruction into its machine language counterpart can be easily carried out from the instruction format specification of the given processor. We will not examine further details of such an instruction format. In the above example, the mov ax, bx has an opcode that is translated into 89h, and the specification of the operands (ax and bx) is further identified via the byte D8h. Hence the instruction occupies two bytes or two memory location. It should be noted that the number of bytes of an instruction depends on the specific instruction and can vary from a single byte to many bytes.

The mov ax, bx instruction copies the content of register bx into register ax.
Suppose initially ax = FEFEh, bx = 8080h.
After executing the instruction, ax = bx = 8080h.

For obvious reasons, in this example, bx serves as the ***source*** operand and ax serves as the ***destination*** operand. To be a valid instruction, the size of source and destination operands should match with each other. Notice that the instruction modifies the content of ax but not the upper half of register eax.

mov eax, [*schar*]

This copies 4 bytes starting from memory location symbolically labeled as *schar* to register eax. [Hence schar is a ***symbolic memory*** address.] The bracket is used to distinguish between the container (schar) and its content ([schar]). Notice that the size (number of bytes) of the operand is implicitly specified by the reference to register eax which is 4 bytes long. With this as the destination container, the assembler will interpret the source is also 4 bytes, and hence 4 bytes from memory will be moved, starting from the location represented by symbolic name schar. This symbolic location (name) is known to the assembler as it must be declared in the program in the data segment. The assembler is responsible for assigning a particular (distinct) address in memory to correspond to this symbolic address. Hence at translation, schar will be translated to a ***displacement*** corresponding to this assigned address.

Suppose initially eax = FEFEFEFEh, [schar] = ‘c’, [schar+1]=’o’. [schar+2] = ‘m’ and [schar+3]=’p’. (Note: each location holds a byte or a character).
After executing the instruction, eax = 706D6F63h (= ‘comp’).

Addressing Modes

- ***Register addressing***

  The specified register contains **the operand.**

  Example: mov ax, bx
  ax is the destination register, bx the source register.

  Hence an operand or temporary result can be buffered in a register and used repeatedly. Any of the general registers can be used for this purpose (instead of registers ax, bx)

- ***Register indirect addressing (Based addressing)***

  The specified register contains the **address of the operand.** Symbolically, we use ***[ ]*** to denote indirect addressing. Hence [ebx] stands for register indirect addressing in which case register ebx contains the memory address of the operand, rather than the operand itself (as in the case of ebx instead). This also corresponds to the based addressing mentioned in section 3.1.

  Example: mov eax, [ebx]
  Suppose initially eax = 0000FEFEh, ebx = 00008080h, and 00001010h is stored in from locations 00008080h up to 00008087h.
  After execution, eax = 00001010h, ebx and memory storages remain unchanged.

  In comparison, move eax, ebx will lead to moving ebx into eax and hence after execution eax = 00008080h instead.

Hence the address of the operand needed can be first stored or computed in a register before the operand is referenced via register indirect. All eight general registers can be used in this manner (for indirect addressing).

- ***Immediate Operand***

The (value of the) operand is contained in the instruction itself (in the displacement field). Hence the operand is in neither a register nor a memory location. In fetching the instruction from memory, the processor has already fetched the immediate operand.

Example: mov ax, 0x1234 //in NASM 0x1234 = 1234h
Suppose initially eax = 1010FEFEh.
After execution, eax = 10101234h.

Example: mov eax, schar
Suppose initially eax = 1010FEFEh, address (memory location) corresponding to schar in the data is 10001000h.
After execution, eax = 10001000h.

A question may have popped up in your mind: how does the computer know the address of schar is 10001000h in this case? As assembly language programmer, you do not have to know. You simply write it down as schar. However, the assembler (translator) examines the data segment that you have declared, it derives/computes the corresponding relative address of each symbolic address such as schar that you have declared in the program. Then it replaces the symbol schar by its binary/hexadecimal equivalent, in this case 10001000h, when it translates the instruction mov eax, schar into the machine language (binary) form.

- ***Indexed Addressing***

The address is specified by an index register together with a scale factor, as appeared earlier on page 15.

Example: mov ax, [esi*2]
Suppose initially esi = 1000h and [2000h] = 10h and [2001h] = 10h.
After execution, ax = 1010h.

A good use of the indexed addressing is to initialize esi to point to the starting address of a structure data, such as an array (example: character string schar). Subsequently, different elements of the array can be accessed (read/written) by incrementing/decrementing/modifying esi. Typically, it is also used in combination with based addressing so that the base of the data structure (such as an array) is pointed to (identified by) a base address register, as in the following.

- ***Based and Indexed (with/without Displacement)***

We can use both the base register (ebx) and the index register (si) together to access a structured data. For example, we could use ebx to point to the starting address of schar in the earlier example (i.e., the location that stores schar[0]), and then by modifying si, we can access different bytes of the character string schar, as in the following.

Example: mov ax, [ebx+esi] (based + indexed addressing)

Suppose initially ebx = 10001000h (starting address of schar), and esi = 2h.
After execution, ax = ‘mp’ = 706Dh.

This is often used for accessing structured data such as an array: with ebx pointing to the starting address and si the index of the element within the array being referenced. Notice without specifying a value, the default value of the scale factor is used in esi.

Example: mov ax, [ebx+esi+2]
(based indexed addressing with displacement)
Here an additional displacement of 2 is specified and can be used together with ebx and esi to specify the memory address of the source operand. In this case, after execution, ax = ‘28’ = 3832h.

By now, you should realize that the availability of various addressing modes provides you with enough flexibility to access memory and registers in program design. You can access memory indirectly by using a register to store the address through register indirect addressing. This is particularly useful when at programming time, you do not know exactly where the data of an instruction is to be found or that location can change during the execution. Perhaps the use of data in that instruction is decided only at runtime, depending on the result computed so far. Register indirect makes it easy.

Sample Programs

(1) Hello World

```
segment  .data                           ;data segment            [1]
msg      db   'Hello World! PleaseType Your ID', 0xA              [2]
len      equ  $ - msg                    ; length of message      [3]
segment  .bss                            ;uninitialized data      [4]
id       resb 10                         ; reserve 10 bytes for ID [5]
segment  .text                           ; code segment           [6]
         global _start                   ; global program name    [7]
_start:                                  ; program entry          [8]
         mov  eax, 4                     ; select kernel call #4  [9]
         mov  ebx, 1                     ; default output device
         mov  ecx, msg                   ; second argument: pointer
                                         ; to message
         mov  edx, len                   ; third argument: length
         int  0x80                       ; invoke kernel call to write
         mov  eax, 3                     ; select kernel call #3  [10]
         mov  ebx, 0                     ; default input device
         mov  ecx, id                    ; pointer to id
         int  0x80                       ; invoke kernel call to read
         mov  eax, 4                     ; select kernel call #4  [11]
         mov  ebx, 1
         int  0x80                       ; echo id read
exit:    mov  eax, 1                     ; select system call #1  [12]
         int  0x80                       ; invoke kernel call
```

The lines shown in bold correspond to the ***program skeleton*** that can be reused in writing input/output via system services provided by the Linux platform. The skeleton is assembler (MASM/NASM) and platform (Linux/Windows) dependent. The lines in italic (but not in bold) are assembler directives: they tell the assembler what needs to be done, including initialization of the contents of some data locations. Here input and output are performed by calling a set of system services (subroutines) available instead of writing your own (usually you need to be a systems programmer rather than application programmer to do so). They are also platform dependent. Words following ‘;’ in each line are comments for documentation purposes.

Further explanation of the sample program is given below. Referring to the [1] – [12] notes labeled on the program:

Note [1]: segment .data is a ‘directive’ to the assembler (NASM) that the following lines specify how the data segment should be initialized (both symbolic addresses and values contained in those memory locations).

Note [2]: msg is a symbolic location starting from which the assembler should initialize with bytes corresponding to the ASCII string ‘Hello World …..ID’ followed by 0Ah (0Ah = the ASCII code of line feed). Upon seeing this, the assembler reserves a specific location in memory for msg, and initializes the 31 bytes (ASCII characters) starting from that location (so that when the resulting machine code is loaded, it will be loaded with these values there).

Note [3]: equ is a pseudo-op; it directs the assembler to assign a constant value to the ***symbol*** len equal to the current location (after the msg string) – msg. As a result, len has a value equal to the length of message string (31). Henceforth, every appearance of the symbol len in the program will be replaced with this value (of 31).

Note [4]: .bss is the directive to the assembler to create an uninitialized data segment. In other words, the locations created in this segment do not have initial values (i.e. can be arbitrary).

Note [5]: id is a symbolic location starting from which the assembler will reserve (resb) 10 bytes to buffer data that will be created later by the program when it is run.

Note [6]: .text is a directive to the assembler to interpret the following segment as the code segment containing assembly language instructions to be translated.

Note [7]: global denotes that _start is a globally known name that can be used (linked) with other program modules to be run together. This is useful when independently written programs can be put together to run; they need to know each other by these global names.

Note [8]: _start is the program entry point.

Note[9]: The 14 instructions that follow are each translated into machine language instructions.
The first 4 mov instructions set up some specific values in registers eax, ebx, ecx and edx (parameter registers), followed by ***int 0x80*** which is a 'software interrupt'. Essentially it invokes a system service available in the platform to complete a service specified in the parameter registers. In this case, the four registers carry the parameters to be used by the system service. Register ebx identifies the device (output), ecx contains the address of the message to be output, and edx contains the length of the message.
Specifically the system is asked to write/ouput (print) the message [msg] on the system console. Control is passed back from the system to this program when the service completes.

Note [10]: System call #3 is next invoked to read (input) a character string into the reserved locations starting from the symbolic address id. Similarly, register ebx identifies the input device (keyboard in this case), and ecx identifies the starting address of the memory to receive the input. This service will terminate when a line feed is received from the input.

Note [11]: The next system call will echo the id character string and output it to the system console. Notice when the system call # 3 is completed, edx will contain a value corresponding to the number of bytes received through the input. So when this output system call is executed, exactly the same number of bytes will be output/echoed to the system console, according to the same interpretation rules of this system service as in note [9].

Note [12]: Finally, control is returned to system kernel with ‘exit’ system call # 1 (specified in register eax).

(2) Input and Reverse Output of String Character

```
segment  .data
msg      db    ‘Type in 20 characters’, 0xA
len      $ - msg
segment  .bss
buffer   resb  20
segment  .text
         global      _start
_start:
         mov  eax, 4
         mov  ebx, 1
         mov  ecx, msg
         mov  edx, len
         int  0x80
         mov  eax, 3
         mov  ebx, 0
         mov  ecx, buffer
         int  0x80
         mov  ecx, 20
show:    mov  eax, 4
         mov  ebx, 1
         mov  edx, 1
         mov  esi, ecx
         add  ecx, buffer – 1
         int  0x80
         mov  ecx, esi
         loop show                   ; repeat 20 times
exit:    mov  eax, 1                 ; select system call #1
         int  0x80                   ; invoke kernel call
```

Observe the use of the skeleton and the similarity between the two example programs, the use of register, register indirect, based + displacement, immediate addressing in the program. Trace the execution of this program as an exercise.

Exercises

4. Give an instruction that involves moving the content of some memory location to another memory location.

5. Write a program that moves 100 bytes from memory location input_buf to memory location output_buf.

6. Give an assembly language 'instruction' that does not correspond to a machine language instruction.

7. Write a program that accepts 25 characters from the keyboard and buffer them in memory location input_buf. Afterwards, the characters are printed.

8. Identify the (program) state changes that occur after execution each instruction during the execution of your program in problem 3.

9. Identify all the instructions in sample program 1 that constains:
   (a) immediate operands
   (b) displacement in operand addresses

10. Identify all the symbolic addresses that appear in the example program 1.

3.2.2 Arithmetic and Logical Instructions

These are instructions that perform primitive arithmetic and logical functions directly supported by the processor. Since the Intel processor is a two-address machine (an instruction can specify at most two operands), in arithmetic/logical operations, one of the specified operands would serve as both source and destination, while the other is a source.

For example, add eax, ebx is the instruction that adds the content of register ebx to that of eax or equivalently in some high level language notation: eax = eax + ebx. Hence eax serves as both source and destination operand while ebx serves as the other source operand.

We will start with some of the relevant arithmetic instructions that we often use and illustrate their functionalities. As in the case of the mov instruction, an operand can be specified using one of the many addressing modes already introduced. This will be assumed. There is an important deviation from high level languages or human languages: arithmetic instructions are binary operators: each time at most two source operands are used. In other words, we cannot express a complex arithmetic expression using a single instruction.

Assume the following initial contents (in hexadecimal) of registers/memory in each of the following examples:

eax = FFFFFFFF   ebx = 00000010   edx = FFFF0000
Starting from schar the memory stores the following bytes: FFFFFF00

```
Examples:     add   eax, ebx          //adding ebx to eax
              After execution:        eax = 0000000F
```
(***S,Z,C,O***) = (0,0,1,0)

The eflag register is affected. In particular, the ***S***ign of the result is 0, the result is non - ***Z***ero (***Z*** = 0), a ***C***arry-out is generated in the addition (***C*** = 1), and ***O***verflow has not occurred (***O*** = 0). [Refer to later section on computer arithmetic and logic for more explanations.]

```
Specifically:         eax =      FFFFFFFF
                      ebx =      000000010
                eax + ebx =     100000000F
```

The addition gives rise to a 32-bit result = 0000000F. Hence the sign = 0. The carry-out generated at the most significant bit is 1. The result is non-zero. Overflow does not arise in the arithmetic (why? We have added a positive integer = 16 to a negative integer = -1, leading to a positive result = 15 that is stored properly in the 32-bit result container eax. The result in eax is correct, right?)

The following cases are similar:

```
add   eax, [schar]        //adding [schar] to eax
After execution:          eax = FFFFFEFF
                          (S,Z,C,O) = (1,0,1,0)

             eax =        FFFFFFFF
         [schar] =        FFFFFF00
   eax + [schar] =       1FFFFFEFF

sub   eax, ebx            // subtract ebx from eax
After execution:          eax = FFFFFFEF
                          (S,Z,C,O) = (1,0,1,0)

             eax =        FFFFFFFF
           - ebx =       +FFFFFFF0
       eax – ebx =       1FFFFFFEF
```

Multiplication operation needs some further explanation. There are two types of integers. Hence two types of integer multiplications, one for unsigned (distinguished with opcode mul) and the other for signed integers (opcode imul). The multiplication of two integers may generate an integer whose significance (number of bytes) is twice that of the original. For example, suppose we multiply a single byte integer with another one. The largest single byte unsigned integer is 255. But 255 * 255 cannot be stored as a single byte and would need 2 bytes instead. Hence by default, the result register is always twice the length of either the source or destination result.
Specifically, the destination operand is always stored in register dx:ax for 16 bit multiplication and edx:eax for 32 bit multiplication.

```
mul  bx                  // dx:ax = ax * bx
After execution:         dx:ax = FFFFFFF0
                         (S,Z,C,O) = (0,0,0,0)

dx:ax = xxxx* FFFF
    bx =          0010
----------------------
dx:ax =   000FFFF0

*x stands for ignore these values.

imul  ebx                // edx:eax = eax * ebx
                         // in 2's complement
After execution:         edx:eax = FFFFFFFF:FFFFFFF0

edx:eax =  xxxxxxxxFFFFFFFF
      ebx = 00000000 0000 0010
------------------------------
edx:eax = FFFFFFFFFFFFFFF0
```

Logic Operators

Boolean variables involve Boolean operators. Typically these are:

and
or
not (or negate)
exclusive-or (or xor)

For simplicity, true = 1 and false = 0.

The following (truth) table defines these Boolean operators.

Operands		Results			
A	B	***not A***	***A and B***	***A or B***	***A xor B***
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

The above definition applies to every bit of a data word. Hence with 4 bits in each of a and b, the result will also be 4 bits in each instance of the following example.

Example:		
	a	= 1011
	b	= 0111
	a and b	= 0011
	a or b	= 1111
	a xor b	= 1100
	not a	= 0100

Logic operators are useful in programming. Besides the direct use in defining conditions to be used in conditional jumps, logic operands are often used in

(i) masking: selecting a subset of bits of a data word while resetting the rest of 0 (using a and b)

(ii) toggling: toggle selected bits of a data word from 0 to 1 or 1 to 0 (using a xor b).

For example, suppose eax contains 32-bit data, and we are interested in extracting the 8-bit character stored in the least significant byte. This is data masking:

```
    and     eax, 0xFF
```

These bit manipulation techniques are useful in bit manipulation functions as well as in I/O programming, such as I/O interface registers for control/handshaking with I/O devices.

Example Program:

This program performs the vector dot product computation on two vectors stored as v1 and v2. Each element of the vector is a single byte.

```
segment   .data
v1        db    1,2,3,4,5,6
v2        db    5,6,7,8,9,0

segment   .bss
dotprod   resw  1

segment   .text
          global      _asm_main
_asm_main:
          mov   ecx, 6           ; initialize count to 6
          sub   esi, esi         ; equivalent to mov esi, 0
          sub   dx, dx           ; initialize dx (result) to 0
cont:     mov   al, [esi+v1]    ;based index addressing to vector*
          mov   bl, [esi+v2]
          imul  bl               ; ax = al * bl
          add   dx, ax           ; update the dot product in dx
          add   esi, 1           ; increment esi by 1
          sub   ecx, 1           ; decrement ecx by 1
          jnz   cont             ; if ecx ≠ 0, jump to cont
          mov   [dotprod], dx    ; update dot product in memory
exit:     mov   eax, 1           ; return control to kernel
          int   0x80
```

Exercise:

11. Modify the above program so that the result is printed.
    Hint: you need to consider conversion from integer represented as unsigned binary number to ASCII before they could be displayed.

12. Identify the state changes after executing the following instructions independently. Assume the following initial contents: eax = 12345678h, ebx = FEDCBA98h, (S,Z,C,O) = (1,1,1,1)
    (i) add eax, ebx
    (ii) mul bx
    (iii) xor ax, bx
    (iv) sub ebx, eax

Shift Operations

Shift operation on a data transforms the data (bit pattern) according to the specification of the shift. There are two main types of shifts: logical vs arithmetic. There are also two directions of shifts: left shift or right shift.

Logical Shift

This operation shifts the operand (memory or register) a number of bits to the left or right. The first operand specifies the operand to be shifted, and the second specifies the number of bits. In logical shift, 0's are shifted into vacated bit positions on the right (in left shift) or on the left (in right shift).

Example:

Operation	content of ax	C

Assume initially in each case	***FEFE***	
shl ax, 1	FDFC	1
shr ax, 1	7F7F	0
shr ax, 2	3FBF	0

An interpretation: A left shift by one bit can be interpreted as multiplication by 2, and similarly for multiple bit left shifts. The result will be correct as long as the original number is 'small' enough. What is the exact limit? Right shift corresponds to division by 2 (unsigned). You should verify this yourself.

Rotate

Rotate allows us retain the bit(s) that are shifted out and move them to the other end, specifically:

$$\textbf{Rotate}(b_{k-1}\, b_{k-2} \ldots b_0) = b_{k-2}\, b_{k-3} \ldots b_0\, b_{k-1}\,.$$

Operation	content of ax	C
Assume initially in each case	***FEFE***	***0***
rol ax, 1	FDFD	1
ror ax, 1	7F7F	0
rol ax, 2	FBFB	1

Rotate allows a programmer to manipulate the data (say in a register) in some manner so that after a full rotation the original data word is preserved. For example, suppose the register is 32 bits. A sequence of rotations equivalent to 32 bits will lead to the same register content as the original.

Arithmetic Shifts

Arithmetic (right) shifts differ from logical (right) shifts in that the most significant (sign) bit is always preserved (unchanged).

Hence $\boldsymbol{ARS(b_{k-1}\, b_{k-2} \ldots b_0) = b_{k-1}\, b_{k-1} \ldots b_1}$.

Operation	Content of ax	C
Assume initially in each case	***FEFE***	***0***
sar ax, 1	FF7F	0
sar ax, 2	FFBF	1

Arithmetic right shift is related to division by 2 for signed integers. It is often called shifts with sign-extension. Essentially the sign of the data (in 2's complement) is retained. Hence in the first example, ARS(-8) = -4 or equivalently we accomplished a division by 2.

3.2.3 Control Flow Operations

These instructions change the flow of control, i.e., modify the value of the program counter to some address other than the next sequential location following the instruction being executed. This change of flow is desirable. From time to time, what should be done next may depend on the result obtained so far (the current state) of the program. To accommodate this possibility, we need to have a choice that is decided at runtime. Conditional jump is a means to achieve this.

Direct changes of control flow within a program (procedure) body can be effected conditionally or unconditionally.

Unconditional Jump:

jmp *label* // jump to instruction at *label*

After executing this instruction, eip will be changed to become the address (offset) corresponding to the symbolic address *label* within the code segment. When jmp is executed, the control is passed to the instruction stored at the symbolic address *label*. In other words, computation continues at the instruction found there.

Conditional Jump:

Conditionally, a jump can be enforced depending on the satisfaction of some condition (specified using the eflag) at the time. This is what is needed to support two or more futures to be chosen, depending on the state of the program retained in the eflags. Usually the satisfaction of the condition is determined by examining the value of a subset of the eflags, as specified by the opcode.

Example:

```
jz      label                // jump to label iff (in eflag) Z = 1
```

The instruction jz is simple: control flow will go the symbolic location label if the current value of Z = 1. In other words, the most recent instruction executed that could affect Z has generated a result = 0. If the result is nonzero (i.e., Z = 0), the jump will fail and execution continues at the instruction that follows the jz instruction in the program body.

Suppose label corresponds to address 10000000h and eip = 0FFFFFF0h (corresponding to the address of the instruction immediately following jz label in the program).

If Z = 1, then eip will be changed to 10000000h and instruction execution continues at 10000000h. Otherwise, instruction execution continues at 0FFFFFF0h.

Other often-used conditional jumps are listed below:

Mnemonic (i.e., symbolic opcode)	Result
jp	jump if parity (P=1)
jg	jump if greater than (S= 0, Z = 0)
jle	jump if less than or equal
jne	jump if not equal (Z = 0)
jo	jump if overflow (O =1)

Generally, most arithmetic and compare instructions affect the flags (S, Z, O, C). Hence by using these conditional jump instructions, one can alter the control flow of the program dynamically at runtime, depending on the results generated at the time. For example, we can proceed to add a number to another number. If the result is negative, add a third number to it.

```
Example:        add   eax, ebx    ;add the two numbers
                jge   cont        ;if result is positive, jump
                add   eax, edx    ; else add a third number
          cont  :::
```

In the above program fragment, jge checks the S flag. If it is 0 (indicating a positive number in the most recent result from the add instruction), then execution continues at the symbolic address cont. Otherwise, we should add the third number stored in edx to eax before continuing the execution at the same address cont. The program at the jge instruction has two immediate futures: one associated with the condition S=0 and the other associated with the condition S=1. It is the case that in this example, the two futures almost immediate merge again at the symbolic location cont. In general, depending on what instructions you place in the fragment, the two futures may run independently before possibly merging at some later instruction address, if ever.

Two other vital control flow instructions are needed to support control flow that gets out of the body of a program (procedure). They are:

```
call   proc        //call subroutine named proc

ret                //exit subroutine; return to caller
```

We will leave these two to a later section on procedure call and return.

Write a program fragment that computes the sum of 0 + 1 + 2 + … + x where x is an integer already stored in register eax [if x = 0 then a result of 0 should be returned in register ebx.]

```
            mov  ebx, 0       // initialize ebx to 0
again:      cmp  eax, 0       // integer > 0?
            jle  done         // integer ≤ 0, get out
            add  ebx, eax
            dec  eax
            jmp  again
done        ::::::
```

3.2.4 Selected Opcodes

We will learn only a subset of the opcodes. If you are interested, you could read the rest from the manual. Usually in a current generation machine, the size of the instruction set can grow to several hundred instructions.

Mnemonics	Meaning
add	addition of two integers
sub	subtraction between two integers
mul/imul	multiply two unsigned/signed integers
div/idiv	divide one integer by another
move	copy from source to destination
inc	increment operand by 1
dec	decrement operand by 1
cmp	compare two operands by sub
and	and of two operands
or	or of two operands
xor	xor of two operands
ror or rol	rotate shift right or left
shr or shl	logical shift right or left
jmp	jump to a different instruction
loop	repeat loop with cx as counter
je	jump if equal
jz	jump if zero
jg	jump if greater than
jge	jump if greater than or equal
jl	jump if less than
jle	jump if less than or equal
jne	jump if not equal
jo	jump if overflow
push	push operand onto stack
pop	pop operand from stack
call	jump to subroutine
ret	return from subroutine

3.4 Procedure Call and Return

Procedure (subroutine) provides us with modularity and reusability of a program module. It resembles the use of a contractor to do work for you in everyday life. An example that has already surfaced earlier on is the input/output subroutine calls through *int 0x80*. In this section, we need to revisit the special data structure, the stack, introduced earlier, as well as picking up the details of the two special linkage instructions (call and ret instructions).

To learn about the underlying mechanics of procedures, we need to understand

- The interface specification of a procedure in the form of parameters
- The control linkage between the caller and callee (the procedure)
- The hardware/instruction support to mechanize the parameter passing and control linkage

3.4.1 Stack

Now it is time for us to study the details of the structure and operation of a stack. A stack is a special data/storage structure that resembles exactly a stack (of books, for example) in everyday life. It has one access point, called the top of the stack. An item can be saved on the stack by pushing (placing) it on top, and can be retrieved (destructively) by popping (removing) it from the top as well. Hence it is also called LIFO (Last-In-First-Out) storage/structure.

In the 80xxx processors, the stack is implemented in the memory with the use of two pointers:

- ss (stack segment) register that points to the bottom of the stack (where you can start pushing the first item)
- esp (stack pointer) register that points to the current top of the stack (where the most recent item has been pushed)

Stack Instructions:

The two common stack instructions (among the derivatives obtained from them) are (i) push reg and (ii) pop reg, where reg is one of the registers.

Example:

Suppose initially eax = 10001000h, esp = 000000EEh,
dword [esp] = FFFFFFFFh   (Here dword[esp] stands for a double length word, i.e., 4 bytes from location pointed by the stack pointer esp. NASM uses the directive dword/word/byte to specify the number of bytes needed in the memory operand whenever it is not apparent from the other operand.)

(i) After executing push eax:

    esp = 000000EAh (=000000EE-4)
    dword[esp] = 10001000h

In other words, the content of eax (10001000h) is pushed on the stack by storing it in the four locations (bytes) that are above the previous top of the stack. Hence the new top (address) of the stack is obtained by subtracting 4 from the previous top of the stack. The 4 is the size of the operand being pushed. In case a 2-byte operand is pushed, the esp register will be decremented by 2 only.

(ii) After executing pop eax:

```
        esp = 000000F2h
        eax = FFFFFFFFh
```

In this case, the pop is accomplished by copying the four bytes stored starting from esp = 000000EEh into eax. Afterwards, the destructive readout requires us to modify the top of the stack pointer (esp register) by incrementing it by 4. Here the 4 corresponds to the 4 locations (bytes) that are popped.

It is noteworthy that from a semantics perspective, once an item is popped from the stack, it should no longer be accessible by any program. Strictly speaking, its value is still residing in the memory (and has not yet been overwritten). But it should be regarded as garbage and should not be accessed by anyone. From security perspective, this may not be a good idea. A safer (and more expensive) implementation (in hardware) could actually overwrite the item with 0 before incrementing the sp register.

From the above, we can easily see that the stack is a dynamic structure that grows and shrinks as program execution proceeds. Hence it is an efficient use of storage (memory) space as the memory can naturally be used to serve useful purposes to the program(s) that the system executes. However, since the stack segment cannot grow and shrink indefinitely, the notion of stack overflow (and underflow) can occur. This represents error conditions as follows:

- Stack overflow

  Stack overflow occurs when a push operation grows the stack beyond the memory storage assigned to hold the stack segment. In other words, the new top of the stack pointer is smaller than the smallest address (stack limit) that can be assigned.

- Stack underflow

  Stack underflow occurs when a pop operation shrinks the stack beyond the base of the stack segment (ss register). In other words, we have performed more pop's than allowed, or equivalently, we try to pop an empty stack.

When stack overflow/underflow occurs, the program would be trapped and terminated.

3.4.2 Procedures and Parameter Passing

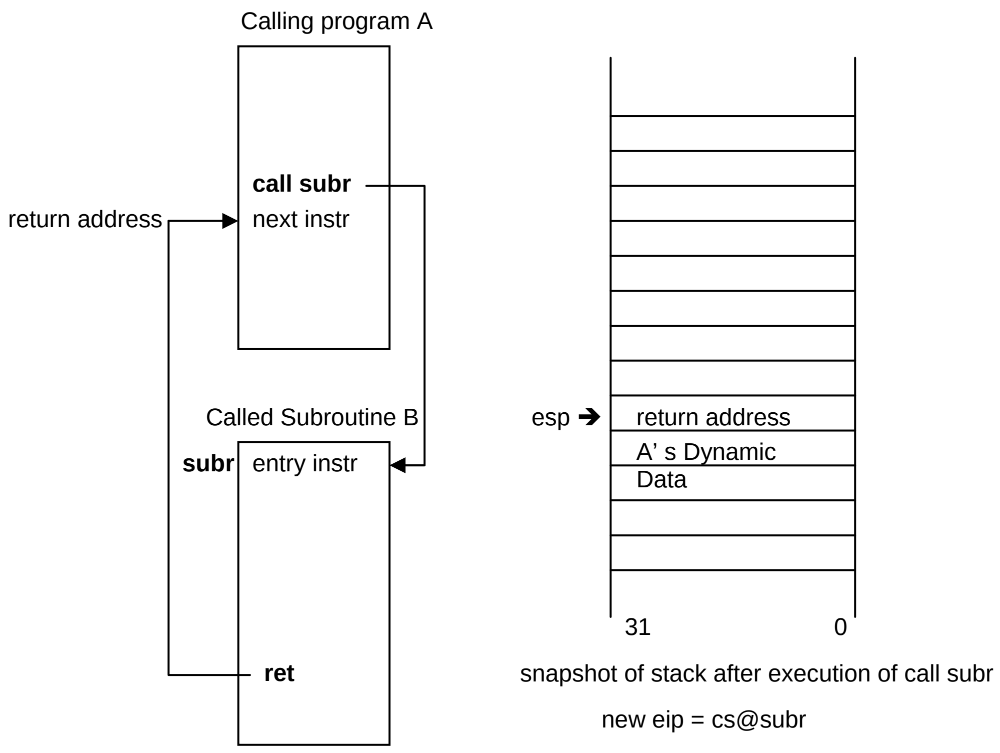
A program (procedure) can call another procedure (called subroutine in assembly language) by setting its eip to the entry target address of the latter. As in a jump instruction, control is passed to the target address at which execution continues. The instruction used to accomplish this is not the jump instruction, however, but the call *subr* instruction where *subr* is the symbolic name of the subroutine.

3.4.2.1 Control Linkage

Why we cannot use jmp subr to pass control to subroutine subr?

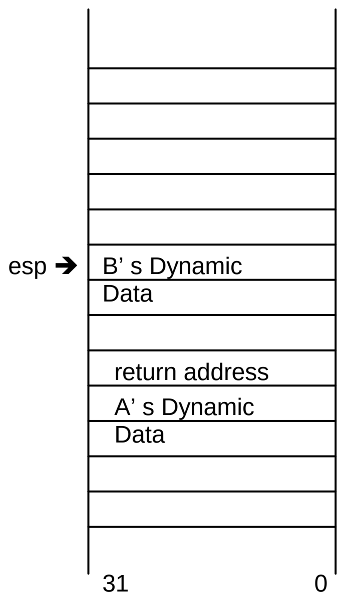
While both instructions end up with eip = subr, there is a vital difference between the call instruction and the jump instruction: somehow the processor has to remember where control has to be passed back (to the caller) when the called subroutine finishes. This is called control linkage. In executing a subroutine, when the subroutine completes its task and executes a ret(urn) instruction, control should be passed back to the caller at the instruction that immediately follows the call instruction (where the caller has paused). This is like the cooperation of two workers: one worker (the caller) decides he needs another worker to help perform some task. So he calls upon the latter for help. The latter takes over and finishes the task. Thereupon, he reports back to the former and allows the former to continue with his work, exactly at the point that follows the ‘cooperation call’.

The following diagrams illustrate the subroutine control passing semantics.

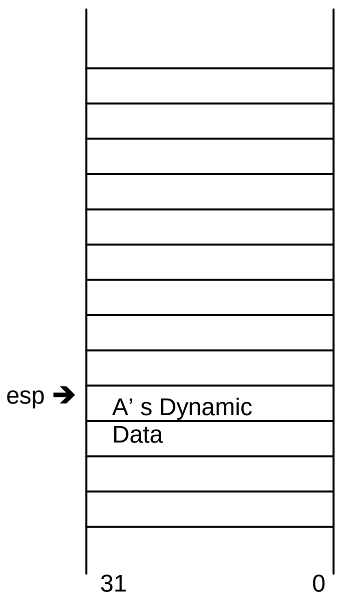


Observations:

- Program A executes call subr. This causes
    - Control is passed to subroutine B at the entry instruction.
    - The return address at A is pushed onto the stack, right on top of the dynamic data/context of A.
    - At this point in time, esp points to the return address that has been pushed and eip is changed to point to subr.



snapshot of stack while in subroutine B



snapshot of stack after execution of ret
new eip = cs@return address

Note: B’s dynamic data must be popped before ret is executed

Observations (continued):

- Subroutine takes over the control and executes. In the code of subroutine B, it may dynamically create and change its dynamic data/context on the stack, leading to the figure on the left hand side in the above. As usual, esp always points to the top of the stack.

- At some point in time, subroutine B completes its function and should return control back to A. Before it does so, it should remove its dynamic context by popping the stack up to but not including the return address.

- Then it executes ret that has the following effects:

    - It pops the return address from the stack (equivalent to doing pop eip)
    - Now the stack looks like the figure on the right hand side of the above diagram.

- Consequently control has been returned to program A (at the return address as eip = return address).

Notice that when control is returned to the caller at the return address, the stack has the same state (content) as that right before the call instruction was performed earlier by program A. It is important as that context belongs to A and is necessary for A to continue.

Hence the call and ret instructions are supported by using the stack as the medium that saves the return address to be used when ret is executed. This is the major difference between the jump and the call instruction.

3.4.2.2 Parameter Passing

The previous section explains how control can be passed from the caller to the callee and vice versa. But what about the details of work order and the results? Somehow, we need to allow these ***parameters*** to be passed from the caller to the callee and ***results*** from the callee to the caller.

A simple and common technique is to make use of the stack again. We can imagine that the portion that belongs to the dynamic contexts of the caller (right below the return address on the stack) can be accessed by both parties relative to esp. Hence a ***protocol*** can be defined for a subroutine so that whoever needs the service of the subroutine, the parameters used are specified according to the protocol on the caller's context block right below the return address. We will highlight this idea with examples.

There are actually two techniques in specifying parameters:

- Parameters are ***passed as values***: here actual values to be used by the callee or results generated by the callee are passed.

- Parameters are ***passed as address*** pointers: here the memory addresses that contain those values are passed instead.

1. ***Pass-by-value*** (the value of the argument is passed):

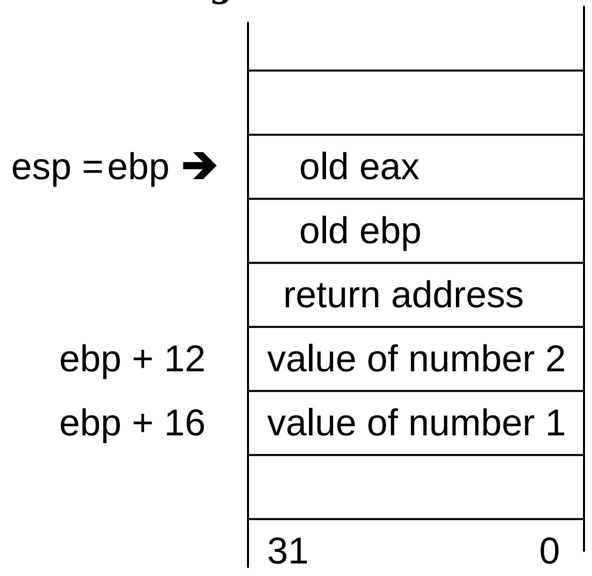
Example:

```
Caller                              Callee

::::
push dword [number1]
push dword [number2]                sum:
call  sum                               push ebp   ;save ebp
::::                                    push eax   ;save eax
                                        mov  ebp, esp
::::                                    mov  eax, [ebp+12]
                                        add  [ebp+16], eax
                                        pop  eax   ; restore
                                        pop  ebp
                                        ret
```

In the above, the protocol (interface) between the caller and callee is this: the integers to be added by the callee are on the stack right "below" the return address and the result is returned in the same stack slot as occupied by the first value passed. The caller is responsible for pushing the values onto the stack before calling sum. The callee is responsible for saving ebp and eax, and later restoring them before returning to the caller. We could have used pusha and popa that saves and restores all registers. However, the latter would have performed more redundant work and therefore may take longer time in switching the context from the caller to the callee and vice versa. In sum, the subroutine uses the base pointer (auxiliary pointer to the stack) to access items not at the top of but inside the stack. For example bp+12 refers to the fourth double word from the top of the stack (as the ebp has been set equal to esp). Since by now, the top elements of the stack are: saved value of eax, saved value of ebp, return address, number 2, number 1, the fourth and fifth items from the top are the two values passed from the caller. Hence the subroutine moves the value of number 2 to eax and adds this to number 1. Then it restores old values of eax and ebp that belong to the caller by using the pop instructions before executing the ret instruction.

Notice that the call instruction saves the 4-byte return address on the stack before setting eip to the starting addressing corresponding to sum. In general***, it is important for the callee to save registers that it may use (and change) and later restores their contents before returning to the caller.***



stack state right after the move ebp, esp instruction

After returning to the caller, it is the responsibility of the caller to remove those arguments from the stack (by incrementing esp) or popping items before proceeding further. If the protocol is for the callee to remove these arguments before the return, then we could replace the ret instruction in the callee by ret 4 in the example. The instruction 'ret 4' returns to the caller after adding 4 to the stack pointer (in addition to popping the return address from the stack). In so doing, upon return to the caller, esp is pointing to the position corresponding to number 1 which is now containing the result computed by the callee.

2. ***Pass-by-reference*** (the address of the argument is passed):

Example:

```
Caller                          Callee

push  x
push  y
call  sum                    sum:
                                push ebp          ;save bp
                                push eax
                                push ebx
                                mov  ebp, esp
                                mov  eax, [ebp + 16]
                                mov  ebx, [ebp + 20]
                                add  [ebx], [eax]
                                pop  ebx          ;restore
                                pop  eax
                                pop  ebp
                                ret  8
```

In the above example, the address of x and y are pushed onto the stack before the caller calls sum. Upon entry, the sum subroutine saves ebp, eax, ebx which are used in the subroutine onto the stack. Then it retrieves the addresses of x and y from the stack into ebx and eax respectively. The summation is performed and the value of x is updated directly via the add instruction. Before returning to the caller, the subroutine pops and restores ebx, eax and ebp respectively. Finally ret 8 returns control back to the caller after popping 8 extra bytes from the stack (below the return address). These 8 bytes correspond to the addresses of x and y. The extra pop’s are performed according to the protocol that assumes that the caller should not see the addresses of x and y on the stack upon return of control.

Instead of using the stack as the medium for parameter passing, it is certainly possible to use the general registers eax, ebx, etc to serve the same purpose. There are two potential limitations in doing so, however:

- The number of parameters that can be so directly passed is much more limited because of the small number of registers available.

- Registers are static entities that belong to the caller. It is difficult to use them when a subroutine can call itself directly or indirectly (this is called recursion).

Example Subroutine:

Write a procedure that obtains (and removes) the starting address and the size of an integer array x[.] from the stack, determines the largest value in the array and replaces the size of x with this value.

```
max:    pusha                   // saves all registers
        mov  ebp, esp
        mov  ebx, [ebp + 36]    // ebx = address of x[0]
        mov  ecx, [ebp + 40]    // ecx = size of array
        sub  esi, esi           // esi = 0
        dec  ecx                // decrement ecx once
        mov  eax, [ebx]         // eax = x[0], the largest value
                                // found so far
again:  inc  esi
        cmp  eax, [ebx + esi*4]
        jg   cont
        mov  eax, [ebx + esi*4]      //update the new maximum
 cont:  loop again              // repeat k times
        mov  [ebp+40], eax      // returns result on the stack
        popa                    // restores registers
        ret                     // return to caller
```