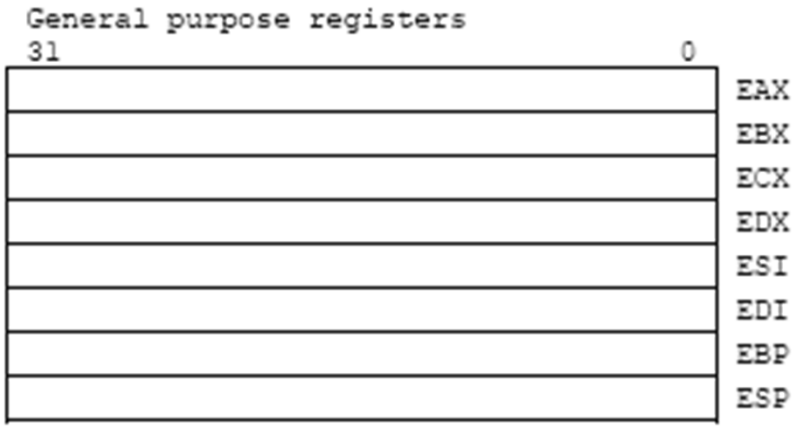
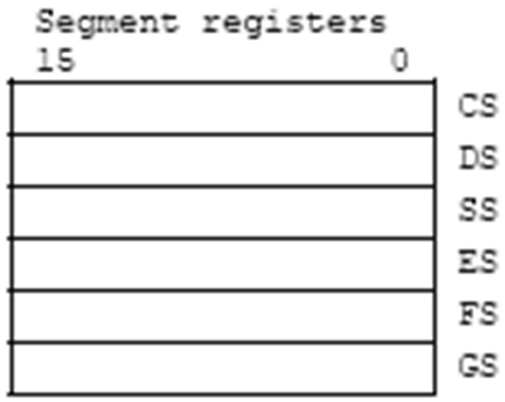
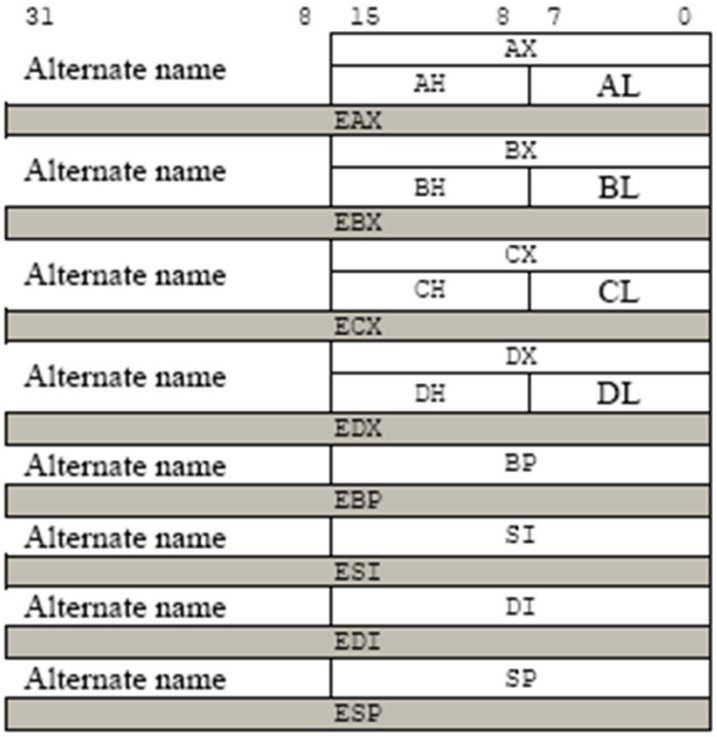
The Basic of x86 Architecture

MICROPROCESSOR PROCESSOR REGISTERS (INTEL x86-32bit)

- 8 general-purpose registers – 32 bits.
- 6 segment registers – 16 bits.
- 1 EFLAGS register – 32 bits.
- 1 EIP, Instruction Pointer register – 32 bits.





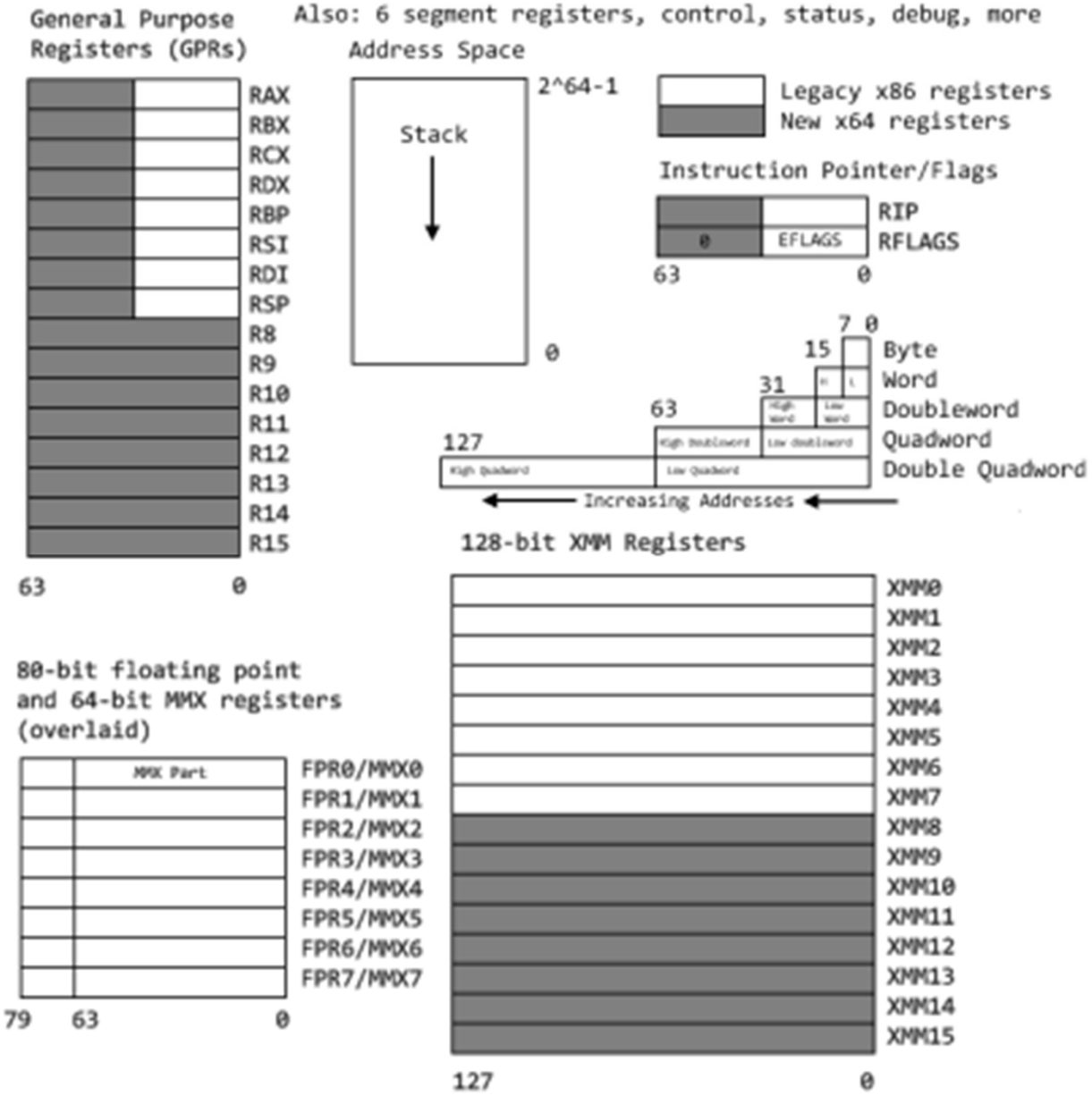


Register Name	Size (in bits)	Purpose
AL, AH/AX/EAX	8,8/16/32	Main register used in arithmetic calculations. Also known as accumulator, as it holds results of arithmetic operations and function return values.
BL, BH/BX/EBX	8,8/16/32	The Base Register. Pointer to data in the DS segment. Used to store the base address of the program.
CL, CH/CX/ECX	8,8/16/3	The Counter register is often used to hold a value representing the number of times a process is to be repeated. Used for loop and string operations.
DL, DH/DX/EDX	8,8/16/32	A general purpose registers. Also used for I/O operations. Helps extend EAX to 64-bits.
SI/ESI	16/32	Source Index register. Pointer to data in the segment pointed to by the DS register. Used as an offset address in string and array operations. It holds the address from where to read data.
DI/EDI	16/32	Destination Index register. Pointer to data (or destination) in the segment pointed to by the ES register. Used as an offset address in string and array operations. It holds the implied write address of all string operations.
BP/EBP	16/32	Base Pointer. Pointer to data on the stack (in the SS segment). It points to the bottom of the current stack frame. It is used to reference local variables.
SP/ESP	16/32	Stack Pointer (in the SS segment). It points to the top of the current stack frame. It is used to reference local variables.

THE SEGMENT REGISTERS

Segment Register	Size (bits)	Purpose	
CS	16	Code segment register. Base location of code section (.text section). Used for fetching instructions.	These registers are used to break up a program into parts. As it executes, the segment registers are assigned the base values of each segment. From here, offset values are used to access each command in the program.
DS	16	Data segment register. Default location for variables (.data section). Used for data accesses.	
ES	16	Extra segment register. Used during string operations.	
SS	16	Stack segment register. Base location of the stack segment. Used when implicitly using SP or ESP or when explicitly using BP, EBP.	
FS	16	Extra segment register.	
GS	16	Extra segment register.	

MICROPROCESSOR PROCESSOR REGISTERS (INTEL x86-64bit)



MEMORY ORGANIZATION

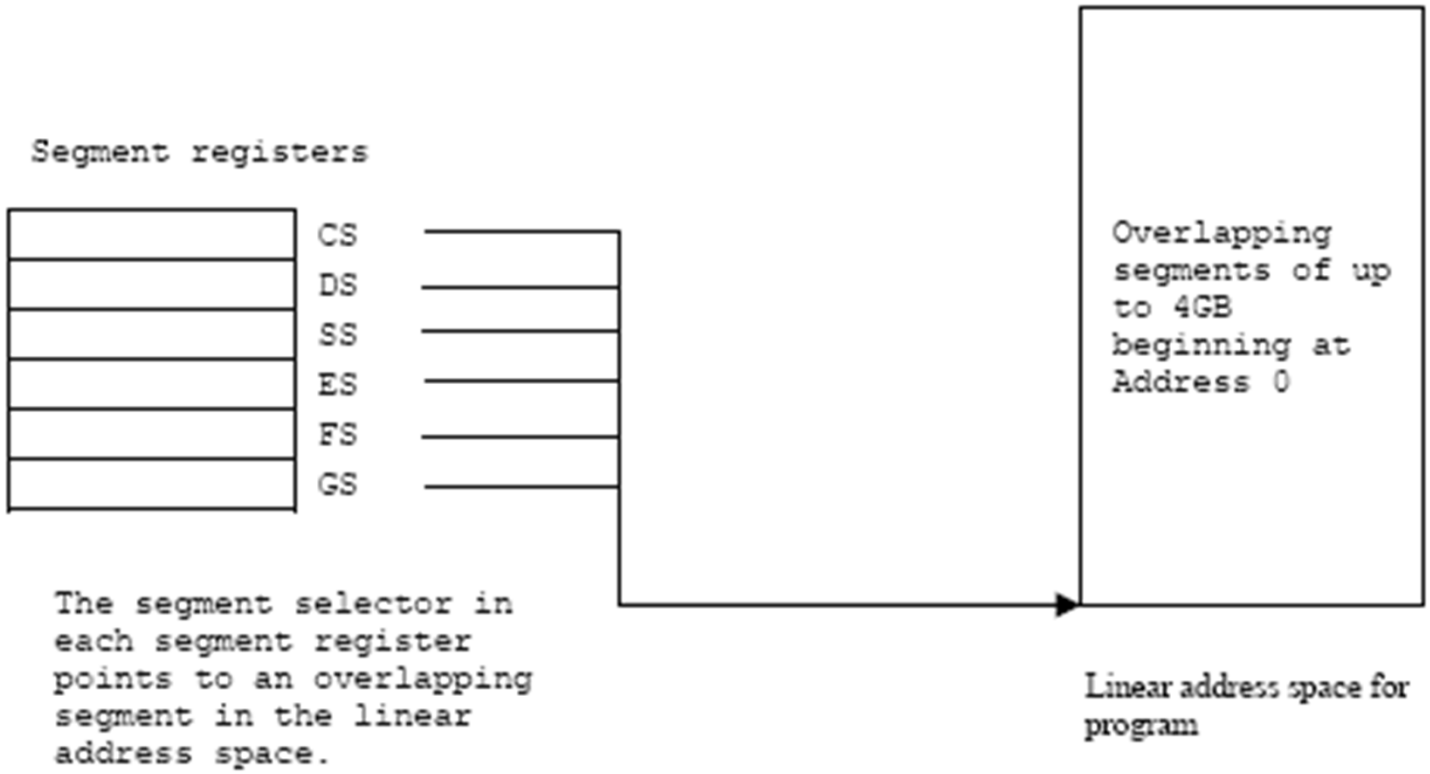
- Registers, Physical memory: Random Access Memory (RAM)
- Inside processor address space that will be used for addressing the RAM
- Logically the address space is organized as a sequence of 8-bit bytes.
- Each byte is assigned a unique address, called a physical address
- The physical address space of the processor ranges from zero to the maximum of 2^{32} -1(4 GB) or 2^{36} -1(64GB) if physical address extension mechanism is used

……

- Operating systems that employ the processor will use processor's memory management facilities (Memory Management Unit – MMU) to access the actual memory
- These facilities provide features such as segmentation and paging
- When using the processor's memory management facilities, programs do not directly address physical memory. Instead they access memory using any of three memory model: flat, segmented or real-address mode.

FLAT MEMORY MODEL

- Memory appears to the program as a single, continuous address space, called a **linear address space**



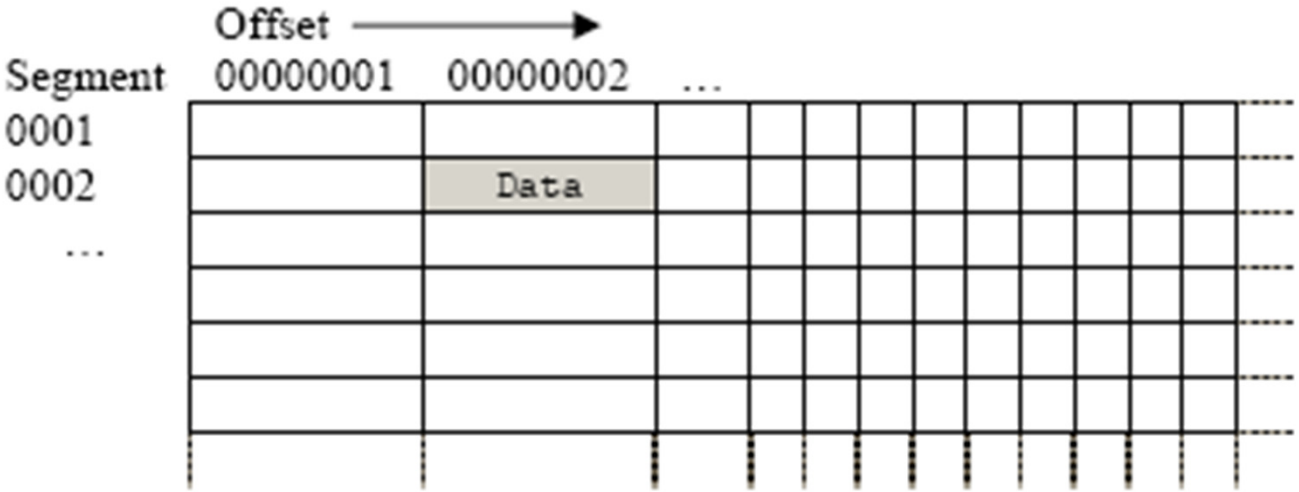
SEGMENTED MEMORY MODEL

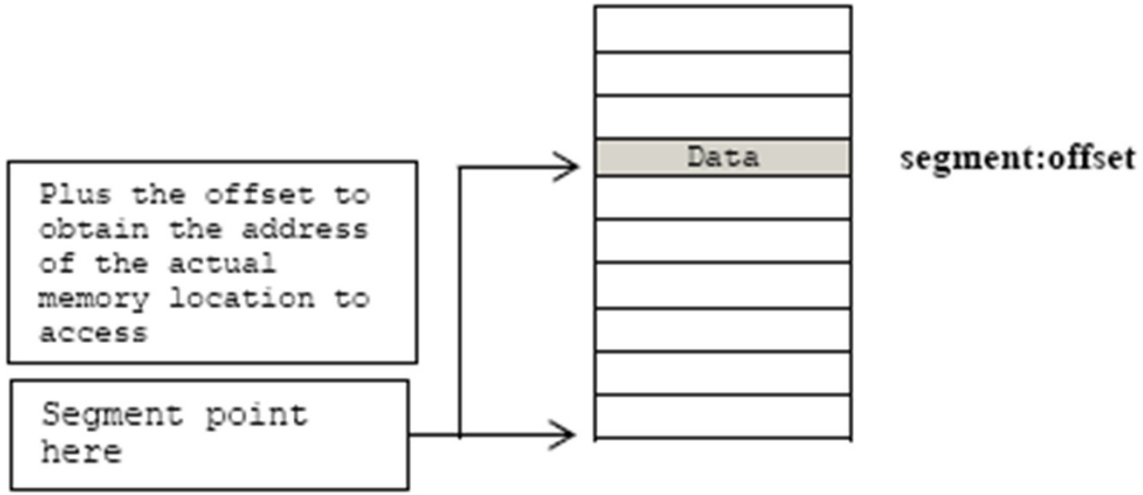
- Memory appears to a program as a group of independent address spaces called segments.
- Code, data and stacks are typically contain in separate segments
- To address a byte in a segment, a program must issue a logical address (often referred to as a far pointer), which consists of a segment selector and an offset.
- All the segments that are defined for a system are mapped into the processor's linear address space.
- To access a memory location, the processor translates each logical address into a linear address. This translation is transparent to the application/program

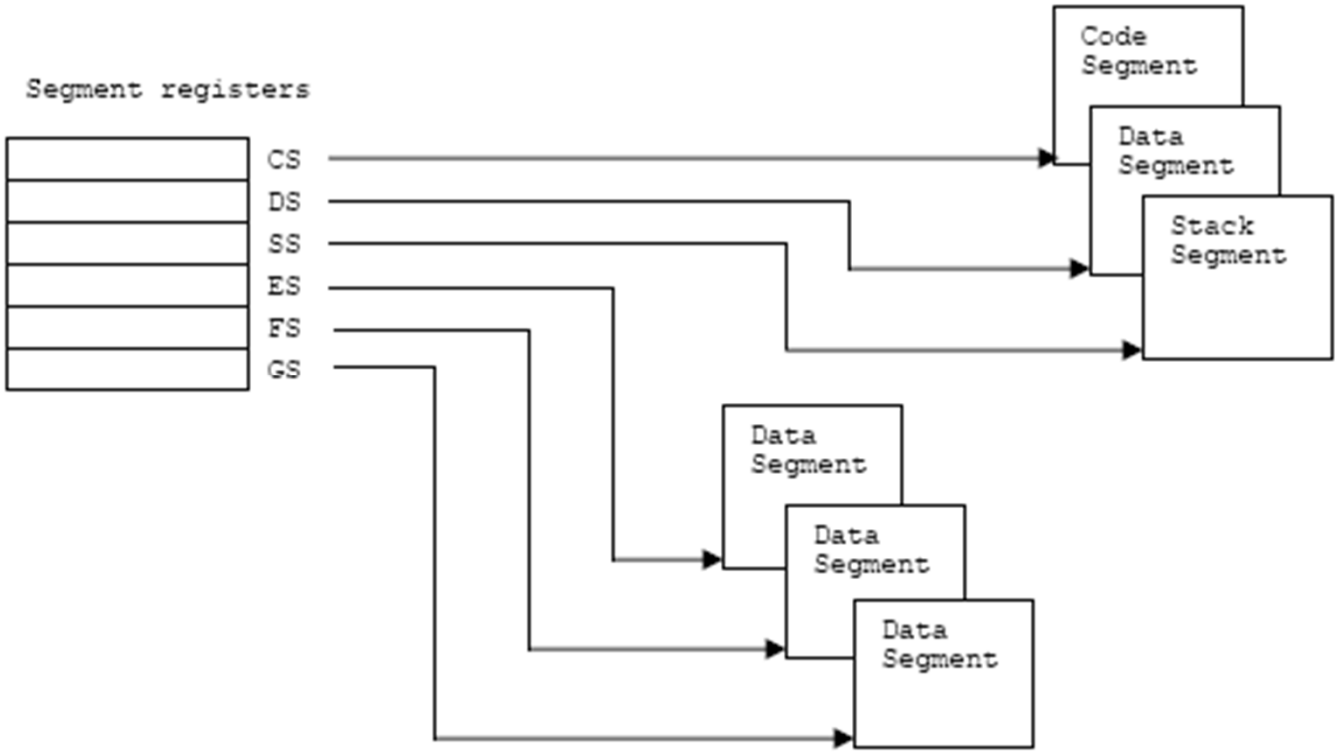
....

Segmented addressing uses two components to specify a memory location:

- A segment value and
- An offset within that segment.




Data
segment:offset
Plus the offset to obtain the address of the actual memory location to access
Segment point here

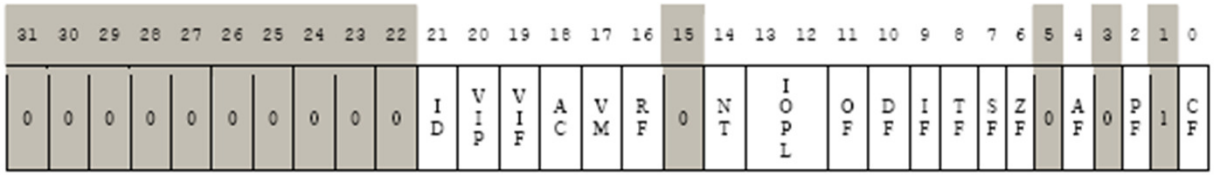

Segment registers
CS
DS
SS
ES
FS
GS
Code Segment
Data Segment
Stack Segment
Data Segment
Data Segment
Data Segment

REAL ADDRESS MODE MEMORY MODEL

- This model used for the Intel 8086 processor.
- This memory model supported in the Intel 32 bits architecture for compatibility with existing programs written to run on the Intel 8086 processor.

THE EFLAGS

- The 32 bits EFLAGS register contains a group of status flags, a control flags and a group of system flags



Flag	Mean	Type
ID	ID Flag	X
VIP	Virtual Interrupt Pending	X
VIF	Virtual Interrupt Flag	X
AC	Alignment Check	X
VM	Virtual 8086 Mode	X
RF	Resume Flag	X
NT	Nested Task	X
IOPL	IO Privilege Level	X
OF	Overflow Flag	X
DF	Direction Flag	C
IF	Interrupt Enable Flag	X
TF	Trap Flag	X
SF	Sign Flag	S
ZF	Zero Flag	S
AF	Auxiliary Carry Flag	S
PF	Parity Flag	S
CF	Carry Flag	S
	X - System Flags	
	C - Control Flags	
	S - Status Flags	

Six of the most important status flags

Flag	Bit	Purpose
CF	0	Carry flag. Set if an arithmetic operation generate a carry or a borrow out of the most significant bit of the result, cleared otherwise. This flag indicate an overflow condition for unsigned integer arithmetic. It is also used in multiple-precision arithmetic.
PF	2	Parity flag. Set if the least-significant byte of the result contains an even number of 1 bit, cleared otherwise.
AF	4	Adjust flag. Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result, cleared otherwise. This flag is used in Binary-Coded-Decimal (BCD) arithmetic.
ZF	6	Zero flag. Set if the result is zero, cleared otherwise.
SF	7	Sign flag. Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. 0 indicates a positive value, 1 indicates a negative value.
OF	11	Overflow flag. Set if the integer result is too large a positive number or too small a negative number, excluding the sign bit, to fit in the destination operand, cleared otherwise. This flag indicates an overflow condition for signed-integer that is two's complement arithmetic.

INSTRUCTION POINTER REGISTER – EIP(32bit)/RIP(64bit)

- EIP/RIP register contains the offset in the current code segment for the next instruction to be executed
- EIP/RIP cannot be accessed directly by software. It is controlled implicitly by control-transfer instructions such as JMP, JCC, CALL, RET and IRET, interrupts and exceptions

....

- The only way to read the EIP/RIP register is to execute the CALL instruction and then read the value of the return instruction pointer from the function stack
- when the CALL instruction executed, the EIP/RIP content of the next address immediately after the CALL, is saved on the stack as return address of the function. Then, the EIP/RIP can be loaded indirectly by modifying the value of a return instruction pointer on the function stack and executing a return, RET/IRET instruction.

ASSEMBLER on x86 PLATFORM

Assembly Registers

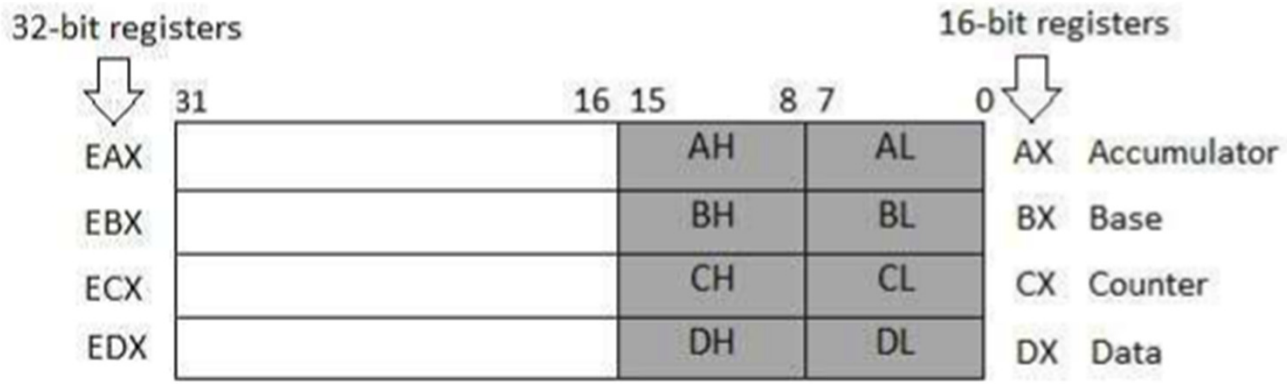
- To speed up the processor operations, the processor includes some internal memory storage locations, called registers.
- The registers stores data elements for processing without having to access the memory.
- A limited number of registers are built into the processor chip

Processor Registers

- Ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories:
  - General registers
  - Control registers
  - Segment registers
- General registers are further divided into the following groups:
  - Data registers
  - Pointer registers
  - Index registers

Data Registers

- Four 32-bit data registers are used for arithmetic, logical and other operations. These 32-bit registers can be used in three ways:
  - As complete 32-bit data registers: EAX, EBX, ECX, EDX.
  - Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
  - Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.

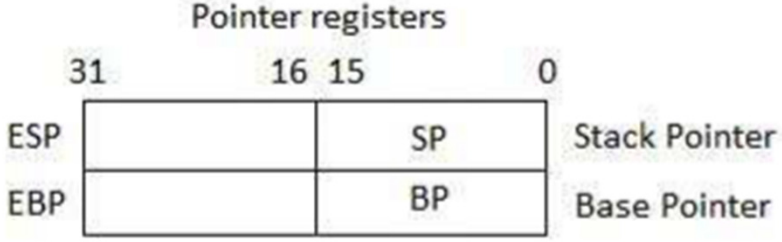


- AX is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX, or AX or AL register according to the size of the operand.
- BX is known as the base register as it could be used in indexed addressing.
- CX is known as the count register as the ECX, CX registers store the loop count in iterative operations.
- DX is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

Pointer Registers

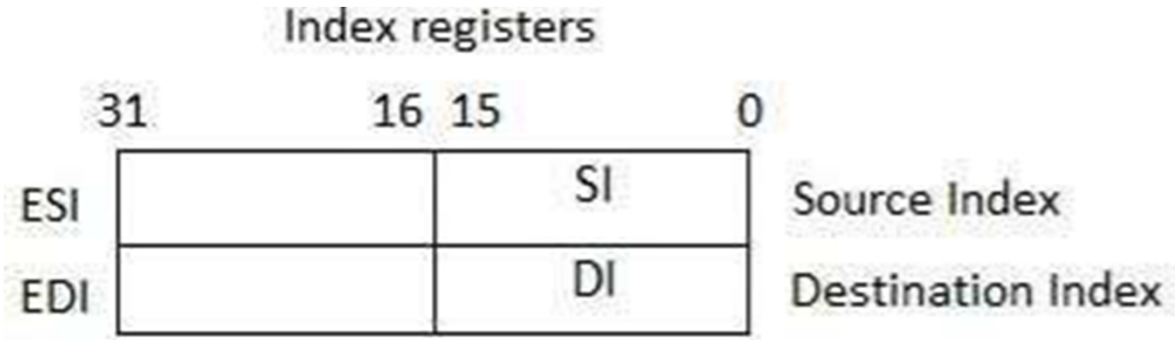
The pointer registers are 32-bit EIP, ESP and EBP registers and corresponding 16-bit right portions IP, SP and BP. There are three categories of pointer registers:

- Instruction Pointer (IP) - the 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- Stack Pointer (SP) - the 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- Base Pointer (BP) - the 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.



Index Registers

- The 32-bit index registers ESI and EDI and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction.
- There are two sets of index pointers:
  - Source Index (SI) - it is used as source index for string operations
  - Destination Index (DI) - it is used as destination index for string operations.



Control Registers

- The 32-bit instruction pointer register and 32-bit flags register combined are considered as the control registers.
- Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

Common Flag bits

- **Overflow Flag (OF):** indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
- **Direction Flag (DF):** determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction.
- **Interrupt Flag (IF):** determines whether the external interrupts like, keyboard entry etc. are to be ignored or processed. It disables the external interrupt when the value is 0.
- **Trap Flag (TF):** allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.
- **Sign Flag (SF):** shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.

....

- **Zero Flag (ZF):** indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
- **Auxiliary Carry Flag (AF):** contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
- **Parity Flag (PF):** indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
- **Carry Flag (CF):** contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

The position of flag bits in the 16-bit Flags register

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Segment Registers

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments:

- **Code Segment:** it contains all the instructions to be executed. A 16 - bit Code Segment register or CS register stores the starting address of the code segment.
- **Data Segment:** it contains data, constants and work areas. A 16 - bit Data Segment register of DS register stores the starting address of the data segment.
- **Stack Segment:** it contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

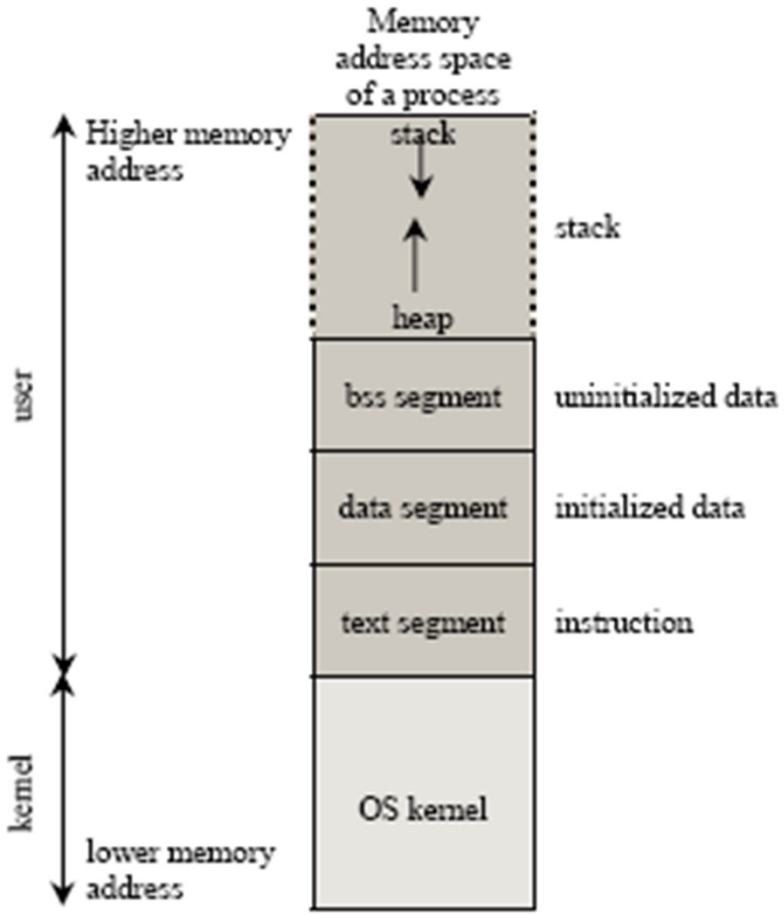
Assembly Basic Syntax

- An assembly program can be divided into three sections:
  - The data section
  - The bss section
  - The text section

The data Section

- The data section is used for declaring initialized data or constants

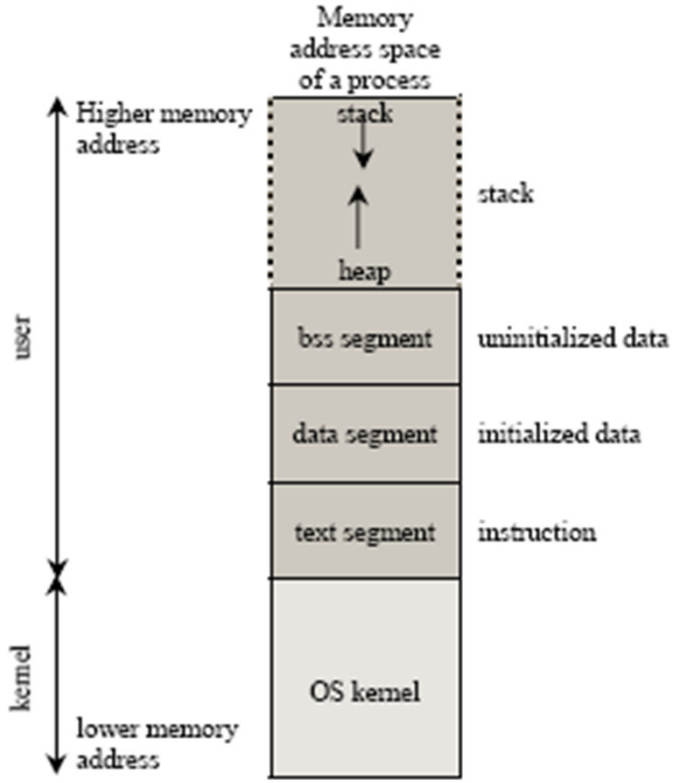
section .data



The bss Section

- bss (block starting symbol
- The bss section is used for declaring variables. The syntax for declaring bss section is:

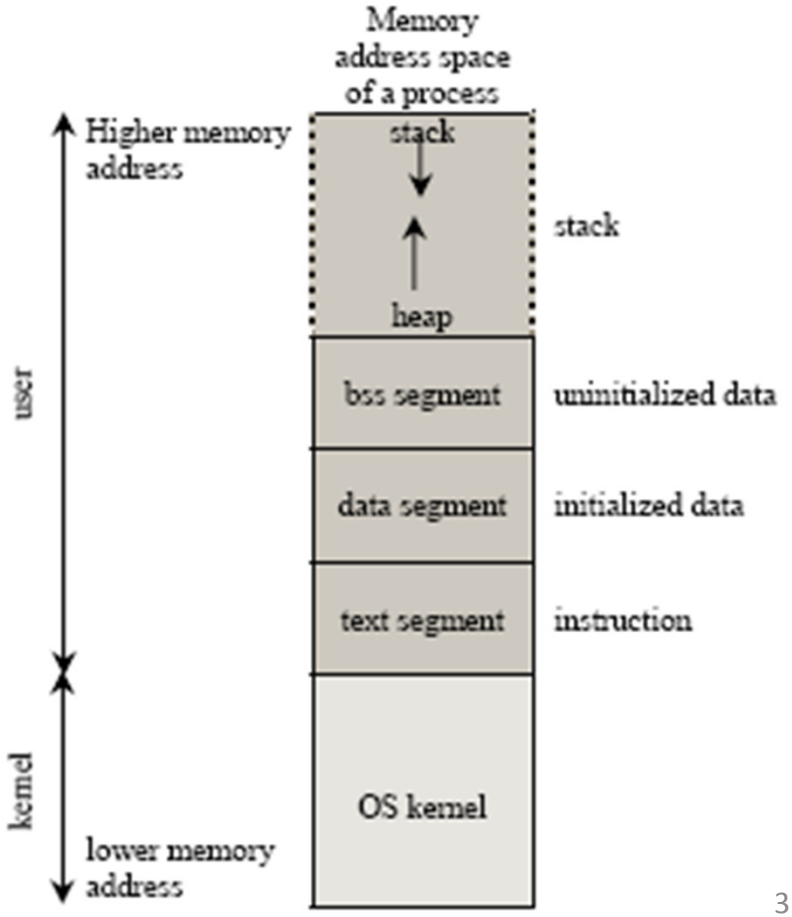
```
section .bss
```



The text section

- The text section is used for keeping the actual code. This section must begin with the declaration **global main**, which tells the kernel where the program execution begins.

```
section .text
  global main
main:
```



Comments

- Assembly language comment begins with a semicolon (;)

Assembly Language Statements

Assembly language programs consist of three types of statements:

- Executable instructions or instructions
- Assembler directives
- Macros

Executable instruction

- Executable instructions tell the processor what to do
- Each instruction consists of an operation code (opcode). Each executable instruction generates one machine language instruction.

Assembler directives

- The assembler directives or pseudo-ops tell the assembler about the various aspects of the assembly process.
- These are non-executable and do not generate machine language instructions

Macros

- Macros are basically a text substitution mechanism

Syntax of Assembly Language Statements

- Assembly language statements are entered one statement per line. Each statement follows the following format:

  [label] mnemonic [operands] [;comment]

  - mnemonic is the name of the instruction

Examples of typical assembly language statements

```
INC COUNT          ; Increment the memory variable COUNT
MOV TOTAL, 48      ; Transfer the value 48 in the  memory variable TOTAL
ADD AH, BH         ; Add the content of the  BH register into the AH register
AND MASK1, 128     ; Perform AND operation on the  variable MASK1 and 128
ADD MARKS, 10      ; Add 10 to the variable MARKS
MOV AL, 10         ; Transfer the value 10 to the AL register
```

Hello World Program

```
section  .text
   global main       ;must be declared for linker (ld)
main:                ;tells linker entry point
   mov  edx,len      ;message length
   mov  ecx,msg      ;message to write
   mov  ebx,1        ;file descriptor (stdout)
   mov  eax,4        ;system call number (sys_write)
   int  0x80         ;call kernel


 mov  eax,1       ;system call number (sys_exit)
 int  0x80        ;call kernel

section  .data
msg db 'Hello, world!', 0xa  ;our dear string
len equ $ - msg              ;length of our dear string
```

Assembly System Calls

- System calls are APIs for the interface between user space and kernel space
- Linux System Calls

Linux System Calls

- You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:
  - Put the system call number in the EAX register.
  - Store the arguments to the system call in the registers EBX, ECX, etc.
  - Call the relevant interrupt (80h)
  - The result is usually returned in the EAX register

system call sys_exit

```
mov     eax,1           ; system call number (sys_exit)
int     0x80            ; call kernel
```

system call sys_write

```
mov     edx,4           ; message length
mov     ecx,msg         ; message to write
mov     ebx,1           ; file descriptor (stdout)
mov     eax,4           ; system call number (sys_write)
int     0x80            ; call kernel
```

Some of the system calls

- All the syscalls are listed in /usr/include/asm/unistd.h

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Example

```
section  .data ;Data segment
   userMsg db 'Please enter a number: ' ;Ask the user to enter a number
   lenUserMsg equ $-userMsg              ;The length of the message
   dispMsg db 'You have entered: '
   lenDispMsg equ $-dispMsg

section .bss          ;Uninitialized data
   num resb 5
section .text         ;Code Segment
      global main
      main:
      ;User prompt
      mov eax, 4
      mov ebx, 1
      mov ecx, userMsg
      mov edx, lenUserMsg
      int 80h
```

```
 ;Read and store the user input
      mov eax, 3
      mov ebx, 2
      mov ecx, num
      mov edx, 5      ;5 bytes (numeric, 1 for sign) of that information
      int 80h
      ;Output the message 'The entered number is: '
      mov eax, 4
      mov ebx, 1
      mov ecx, dispMsg
      mov edx, lenDispMsg
      int 80h

      ;Output the number entered
      mov eax, 4
      mov ebx, 1
      mov ecx, num
      mov edx, 5
      int 80h
; Exit code
      mov eax, 1
      mov ebx, 0
      int 80h
```

Assembly Variables

- The syntax for storage allocation statement for initialized data is:

[variable-name] define-directive initial-value [,Initial-value]...

- Where, variable-name is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment.
- There are five basic forms of the define directive

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Following are some examples of using define directives:

```
choice          DB      'y'
number          DW      12345
neg_number      DW      -12345
big_number      DQ      123456789
real_number1    DD      1.234
real_number2    DQ      123.456
```

Allocating Storage Space for Uninitialized Data

- There are five basic forms of the reserve directive:

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Multiple Initializations

- The TIMES directive allows multiple initializations to the same value
- For example, an array named marks of size 9 can be defined and initialized to zero using the following statement:

```
marks  TIMES  9  DW  0
```

Assembly Constants

- The EQU Directive
- The %assign Directive
- The %define Directive

The EQU Directive

- The EQU directive is used for defining constants. The syntax of the EQU directive is as follows:

CONSTANT_NAME EQU expression

- For example,

TOTAL_STUDENTS equ 50

- can then use this constant value in your code, like:

```
mov  ecx,  TOTAL_STUDENTS
cmp  eax,  TOTAL_STUDENTS
```

- The operand of an EQU statement can be an expression:

```
LENGTH equ 20
WIDTH  equ 10
AREA   equ length * width
```

Above code segment would define AREA as 200.

Example

```
SYS_EXIT  equ 1
SYS_WRITE equ 4
STDIN     equ 0
STDOUT    equ 1
section   .text
  global main    ;must be declared for using gcc
main:   ;tell linker entry point
      mov eax, SYS_WRITE
      mov ebx, STDOUT
      mov ecx, msg1
      mov edx, len1
      int 0x80

      mov eax, SYS_WRITE
      mov ebx, STDOUT
      mov ecx, msg2
      mov edx, len2
      int 0x80

      mov eax,SYS_EXIT    ;system call number (sys_exit)
      int 0x80            ;call kernel
```

```
section   .data
  msg1 db  'Hello, programmers!',0xA,0xD
  len1 equ $ - msg1
  msg2 db 'Welcome to the world of,', 0xA,0xD
  len2 equ $ - msg2
```

The %assign Directive

- The %assign directive can be used to define numeric constants like the EQU directive.
- This directive allows redefinition. For example, may define the constant TOTAL as:

```
%assign TOTAL 10
```

Later in the code you can redefine it as:

```
%assign  TOTAL  20
```

- This directive is case-sensitive.

The %define Directive

- The %define directive allows defining both numeric and string constants. This directive is similar to the #define in C. For example, may define the constant PTR as:

```
%define PTR [EBP+4]
```

- The above code replaces PTR by [EBP+4].
- This directive also allows redefinition and it is case sensitive.

Arithmetic Instructions

- INC destination
- DEC destination
- ADD/SUB destination, source
  - memory-to-memory operations are not possible using ADD/SUB instructions.
- MUL/IMUL multiplier
  - The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data.
  - Both instructions affect the Carry and Overflow flag.
- DIV/IDIV divisor

Logical Instructions

SN	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

Assembly Strings (1/2)

- Must have noticed that, the variable lengths strings can have as many characters as required. Generally, we specify the length of the string by either of the two ways:
  - Explicitly storing string length
  - Using a sentinel character

- Can store the string length explicitly by using the $ location counter symbol, that represents the current value of the location counter. In the following example:

msg db 'Hello, world!',0xa ;our dear string

len equ $ - msg ;length of our dear string

$ points to the byte after the last character of the string variable msg. Therefore, **$-msg** gives the length of the string.

- Can also write

msg db 'Hello, world!',0xa ;our dear string

len equ 13 ;length of our dear string

Assembly Strings (2/2)

Alternatively, can store strings with a trailing sentinel character to delimit a string instead of storing the string length explicitly. The sentinel character should be a special character that does not appear within a string.

For example:

message DB 'I am loving it!', 0

String Instructions

- Each string instruction may require a source operand, a destination operand, or both. For 32-bit segments, string instructions use ESI and EDI registers to point to the source and destination operands, respectively.
- For 16-bit segments, however, the SI and the DI registers are used to point to the source and destination respectively.

Five Basic String Instructions

- MOVS - This instruction moves 1 Byte, Word or Doubleword of data from memory location to another.
- LODS - This instruction loads from memory. If the operand is of one byte, it is loaded into the AL register, if the operand is one word, it is loaded into the AX register and a doubleword is loaded into the EAX register.
- STOS - This instruction stores data from register (AL, AX, or EAX) to memory.
- CMPS - This instruction compares two data items in memory. Data could be of a byte size, word or doubleword.
- SCAS - This instruction compares the contents of a register (AL, AX or EAX) with the contents of an item in memory.

Basic Instruction	Operands at	Byte Operation	Word Operation	Double word Operation
MOVS	ES:DI, DS:EI	MOVSB	MOVSW	MOVSD
LODS	AX, DS:SI	LODSB	LODSW	LODSD
STOS	ES:DI, AX	STOSB	STOSW	STOSD
CMPS	DS:SI, ES: DI	CMPSB	CMPSW	CMPSD
SCAS	ES:DI, AX	SCASB	SCASW	SCASD

Example

```
section  .text
    global main          ;must be declared for using gcc
main:    ;tell linker entry point
         mov      ecx, len
         mov      esi, s1
         mov      edi, s2
         cld
         rep      movsb
         mov      edx,20   ;message length
         mov      ecx,s2   ;message to write
         mov      ebx,1    ;file descriptor (stdout)
         mov      eax,4    ;system call number (sys_write)
         int      0x80     ;call kernel
         mov      eax,1    ;system call number (sys_exit)
         int      0x80     ;call kernel
section .data
s1 db 'Hello, world!',0 ;string 1
```

```
len equ $-s1
section   .bss
s2 resb  20              ;destination
```

Example

```
section .text
   global main          ;must be declared for using gcc
main:    ;tell linker entry point
   mov    ecx, len
   mov    esi, s1
   mov    edi, s2
loop_here:
   lodsb
   add al, 02
   stosb
   loop    loop_here
   cld
   rep     movsb
   mov     edx,20    ;message length
   mov     ecx,s2    ;message to write
   mov     ebx,1     ;file descriptor (stdout)
   mov     eax,4     ;system call number (sys_write)
   int     0x80      ;call kernel
   mov     eax,1     ;system call number (sys_exit)
   int     0x80      ;call kernel
section .data
s1 db 'password', 0 ;source
len equ $-s1
section .bss
s2 resb 10          ;destination
```

The End