Accelerated Learning Series

(www.ALearn.net – A site dedicated to education)

Modules in Computer Science & Engineering

x86 Assembler Programming

A set of notes detailing fundamental concepts in low level Programming.

Author: SKG.
Dec 2007.

Revision History

Version 1.0 (Dec 2007)

- First version created

Table of Contents.

Prerequisites

- General Purpose Computer Architecture (ALS notes in Hardware Engineering)

Preface

As I started working on this set of notes, I was confounded by many questions related to the most fruitful approach to take. To start with it is fair to question the relevance of Assembler level programming in the midst of the more popular and pervasive high level programming languages. Once one justifies its relevance, there is still the question of which processor to choose to illustrate the concepts in Assembler programming.

Over the last two decades I have observed the number of Assembler programmer positions in the industry dwindle substantially. Today there are probably only two computer industries that exploit the skills of Assembler programmers. The first would be those who write very specialized software that cannot be adequately generated by generic compilers. The other industry would be those who need to understand and resolve difficult problems that hinge on the hardware in which the problems manifest. Besides these two industries however, the average high level computer programmer also benefits, to a lesser extent admittedly, from knowledge of Assembler programming in understanding shortcomings of a compiler and sometimes even revealing faults in their high level code. Even if you find that you will never program a computer at the Assembler level, a preliminary course in Assembler programming primes the student for better grasping constructs in higher level languages.

Since the Personal Computer (PC) has become ubiquitous and since the PC is based on the **Intel** x86 processor, it seemed convenient to use the x86 architecture for illustration here. The fact the **Microsoft** offers the use of their x86 Assembler (**MASM**) at no charge (for non-commercial purposes) also played a role in this choice of processor.

The x86 processor operates in 32-bit mode in most modern personal computers. For illustrative purposes in this set of notes however, all the examples will be based on the 16-bit mode of the x86 processor. Since the latest version of MASM is tuned for the 32-bit mode, I have saved the earlier 16-bit MASM at http://www.geocities.com/skgalearn/ML.zip for download. The student will need to download and save the two files (assembler and linker) to a directory in their local hard-drive and add that directory to the **path** to practice the samples provided in this set of notes. Since most of the schools that are currently availing themselves of the ALS notes are using Windows XP SP2, I have provided more detailed steps on how this can be done in Windows XP in section 1.3.

Acknowledgements

I am sincerely grateful to A. Walker and J. Elliott for taking the time to review this document and for providing valuable and constructive feedback.

1.0 – 16-Bit x86 Architecture

The general purpose computer architecture prepared us with an understanding of the various components that make up a computer. Hence we are familiar with concepts surrounding the CPU, Registers, Memory, data buses, address buses, instruction mnemonics and operands. In this section we delve a little deeper into a specific processor type - namely the x86 processor.

The emphasis in this section however, will be to expose the student to the mechanics of writing assembler code, converting that to machine code and finally executing and debugging the code.

1.1 – Real-Mode 8086 Registers

As discussed in the General Purpose Computer Architecture notes, Registers are very fast access memory locations in the CPU.

The 8086 processor has five types of 16-bit registers – **General**, **Segment**, **Index**, **Pointer** and the **Flags** register.

General Registers:

The **General** registers are the **AX**, **BX**, **CX** and **DX** registers. Each of these 16-bit registers are also byte accessible by replacing the "**X**" in the register name with "**L**" (for the lower byte) or "**H**" (for the higher byte). For example the high byte in the AX register can be accessed as "**AH**". Similarly the low byte in the AX register can be accessed as "**AL**". The General registers are commonly used to store the input and output values of arithmetic operations.

Segment Registers:

The **Segment** registers are the **CS**, **DS**, **SS** and **ES** registers. To understand the purpose of these registers, a brief exposure to the layout and operation of software is necessary. Hence we will briefly digress in the next few paragraphs to gain this exposure.

Software is generally a set of instructions that operate on a set of variables. The **instructions** are referred to as "**Code**" and the **variables** are referred to as "**Data**". The "**CS**" register holds the base address for the **Code segment** and the "**DS**" register holds the base address for the **Data segment**.

It is often useful to group software instructions in a functional manner to avoid having to repeat the same set of instructions each time a particular functional operation is required. By so doing, we can point the CPU to the address where the code that implements a particular **function** resides when that particular functional operation is required.

This sort of grouping of software into functions begs the question as to how the CPU can trace its way back to the code that called a particular function when the function completes its set of instructions. The problem gets further compounded when you think of the possibility of a one function that ends up calling yet another function and that in turn calling yet another and so on.

The CPU resolves this problem using what is commonly referred to as the "**Stack"**. Think of the stack as a contiguous piece of memory. The "**SS**" register holds the base address for the **Stack segment** of a software program. Every time one function calls another function, the CPU will save the address of the code to which it needs to return when the called function completes its set of instructions, on the stack. In computer lingo, this saved address is called the "**Return Address**".

The "**ES**" register is the "**Extra Segment**" register. This functions much like the other segment registers in that it allows the user to refer to a relative address with respect to this register as the base address.

It is worth noting that the use of a segment base address when referring to the code, data or stack addresses is not essential. CPU architectures could allow absolute addresses without the need to refer to base segment addresses. The 8086 architecture is often referred to as having a "**segmented architecture**" because of the fact that it chooses to use these segment base addresses instead of absolute addresses.

Index Registers:

Some computer instructions operate on contiguous memory locations starting at a particular address and for a certain size. A common example would be an instruction that copies a string (an array of characters that often ends with a NULL character) from one location in memory to another location. This instruction would need the start address of the source string, the length of the source string and the start address of the destination string. The instruction can then **index** with reference to the start address of the source and destination locations to access each subsequent memory location.

The "**SI**" and "**DI**" are both 16-bit registers that are commonly used as **source** and **destination index registers**.

Pointer Registers:

"**SP**" is the 16-bit **Stack Pointer register**.

"**IP**" is the 16-bit **Instruction Pointer register**.

"**BP**" is the 16-bit Base Pointer register, that is commonly used by functions to save the "SP" register before reusing the "SP" register to allocate memory on the stack of local variables.

Flags Register:

The CPU stores the results of certain operations in the "Flags" register.

The following defines the 8 bits in the flags register along with the common representation to indicate if the bit is set or not.

Overflow Flag: NV (No overflow) or OV (Overflow) – **Used with signed overflow.**

Direction Flag: DN (Decrement) or UP (Increment)

Interrupt Flag: EI (Interrupts enabled) or DI (Interrupts disabled)

Sign Flag: PL (Sign flag off) or NG (Sign flag on) – **Set to the sign bit of the last operation.**

Zero Flag: ZR (Flag is on) or NZ (Flag is off) – **Set when the last operation resulted in a zero.**

Auxilliary Carry Flag: AC (Aux carry flag is on) or NA (Aux carry flag is off) – **Nibble carry.**

Parity Flag: PE (Even parity) or PO (Odd parity) – **Set when there is an even number of 1s.**

Carry Flag: CY (Carry flag is on) or NC (Carry flag is off) – **used with unsigned overflow.**

1.2 – Assembler Directives

A computer program is nothing more than a set of instructions. Some of these instructions target the software that converts the program to machine language. The rest of the instructions are meant to be executed by the processor for which the program is written.

The software that coverts an assembly language program to machine language is called an **Assembler**. The instructions directed at the assembler are referred to as **Assembler directives**. Let us examine some common assembler directives

Stack Segment Directive:

When a program is loaded, a certain amount of memory needs to be reserved for stack space. As previously discussed, this is the space used to keep track of the procedure calls made from other procedures. The following is an example of how to tell the 16-bit x86 assembler to reserve 100 bytes of stack space.

```
StackSg     Segment     Para    Stack   'Stack'
            db          200     dup(?)
StackSg     EndS
```

Here "StackSg" is a label (a user defined name) for a Segment. "Segment" is a keyword known to the assembler. Similarly 'Stack' is another keyword known to the assembler. The assembler reads the first line in the directive above and knows that you are trying to tell it to reserve a stack segment. The assembler reads the next line to find out how many bytes you want to reserve.

The "db" instruction tells the assembler that the number that follows refers to the unit of "bytes". The "dup(?)" instruction tells the assembler that you don't care to initialize the memory you are reserving. If the instruction had read "db 200 dup(0)" instead, then you are asking the assembler to duplicate "0" 200 times. Or in other words you reserve 200 bytes and initialize it to all zeros.

Finally the "StackSg EndS" directive tells the assembler that you are done defining your stack size.

Data Segment Directive:

Most programs will need to use certain constant values throughout the program. It would be useful to refer to these "**constants"** by name and define the constant just once and get the assembler to replace every instance where that name is used with its constant value. This aids the programmer by having to change the constant in only a single location instead of every location where it is used, should the constant value have to change at some point.

Similarly every program will need some reserved memory to keep track of values that change from time to time. For ease of use, we will give these memory locations unique names and will refer to them as "**variables**".

Constants and variables are defined in the data segment. Below is an example of a Data directive.

```
DataSg      Segment     Para    'Data'
        CR          equ         14      ; Carriage Return
        LF          equ         10      ; Linefeed
        DOSWRITE    equ         40h     ; DOS function to write
        StdOut      equ         1       ; Standard output file number

        MsgString   db          'My First Program', CR, LF
        MsgLength   equ         $-MsgString
DataSg      EndS
```

As with the stack segment, we indicate to the assembler that we are defining a segment of type 'Data' and will from now on refer to it using the label "DataSg".

Then we define four constants. The first two constants are defined by the American Standard Code for Information Interchange (**ASCII**) which is responsible for defining the binary codes used to define characters in a computer. The codes "14" (**CR**) and "10" (**LF**) were originally intended as instructions to a printer to carriage return and line feed respectively.

The **DOSWRITE** constant is a value known by the DOS operating system. When a program interrupts the DOS operating system and asks it to perform a task, it has to give it a code defining the task that needs to be performed. The code "40" in hexadecimal (h) indicated to DOS that it was expected to perform a write instruction.

Similarly, the "**StdOut**" constant is a value indicating to DOS where it should write to. The constant "1" indicated the standard output, usually the display terminal.

The assembler will replace the constant values associated the "CR", "LF", "DOSWRITE" and "StdOut" every where they are used in the program as part of the assembling process.

Note the "**;**" after each directive - these are comments for ease of reading the program. It is a directive to the assembler to ignore the rest of the line.

"**MsgString**" is a variable and not a constant. Hence we have to reserve memory in the data segment to store its value. We use the "db" (define byte) directive to tell the assembler to reserve memory and initialize it to the string value we have initialized it to. The assembler knows how long the string is and so we don't need to tell it how much memory to reserve.

However, we will need to know the length of the string at different times in our program. Hence we define a constant for this purpose. Unlike the previously defined constants, we get the assembler to set this constant. This saves us from counting the number of characters ourselves and also allows us to modify the string later without having to change the length constant each time we change the string. The **"$-MsgString"** is a directive that effectively tells the assembler to take the current address ($) and subtract from it the address associated with the label "MsgString".

Finally we indicate to the assembler that we have completed defining the data segment using the "**DataSg EndS**" directive.

Code Segment Directive:

Like the Stack and Data segment directives, the code segment is defined using the keyword "Segment" with the 'Code' identifier as shown below.

```
CodeSg      Segment     Para    'Code'
            Assume      cs:CodeSg, ss:StackSg, ds:DataSg
Main        Proc        Far

            mov     ax, DataSg
            mov     ds, ax

Main        EndP
CodeSg      Ends
```

The "Code" segment contains the actual instructions that are intended to be executed by the processor.

The "Assume" directive tells the assembler that it can assume that you will set up the code, stack and data segment registers to point to the addresses of the labels associated with those segments.

The "Main Proc Far" is a declaration indicating that you are starting a procedure called "Main".

The "mov" instructions are instructions known to the x86 processor. They are not assembler directives. "mov ax, DataSg" is an instruction asking the processor to move the address associated with the label "DataSg" to the "ax" register. Similarly "mov ds, ax" is an instruction to move the contents of the "ax" register into the "ds" register.

The “Main EndP” declaration indicates that it is the end of the “Main” procedure.

Finally we indicate the end of the code segment using the “CodeSg Ends” directive.

1.3 – Your First Computer Program – "Hello World!"

To download the Microsoft Assembler that can be used with the samples provided in this set of notes, follow these steps.

1) Download the Assembler and Linker from http://www.geocities.com/skgalearn/ML.zip and save it in a known location on the local disk (eg. "C:\ml")
2) Unzip the zip file that was saved in step 1. If you are using Windows XP, the "WinZip" applet is available if you right click on the zip file and select "Extract to here". This will extract the Assembler, Linker and the samples from the zip file.
3) Add the location of the unzipped Assembler and Linker to your system path so that you can access them from anywhere. To do this in Windows XP, press the "Start" button on the task bar, then right click on "My Computer" and select "Properties". This will bring up a dialog with several tabs. Select the "Advanced" tab and click on the "Environment Variables" button. This will display two list boxes with "User variables" and "System Variables". In the list box with "System Variables" you will notice there is already a "Path" variable. Double click on this "Path" variable to allow you to edit it. Append the location where you stored the Assembler and linker to this "Path" variable. Ensure that you do not alter the existing paths but just append a new path.

Once you have downloaded the 16-bit assembler and added it to your path, it is time to time try our hands at a simple program.

A program is generally written using a text editor. The most common text editor in Windows XP is the notepad. To open the "notepad" editor, click on the start menu and then click on "Run". This is will display a dialog box. Type "notepad.exe" in this dialog box and click on "Ok". This will launch the notepad editor.

Type the following lines of code into your notepad editor, or copy and paste it into the editor.

```
;=====================================================================
; Sample1.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                db              200     dup(?)
StackSg         EndS

;=====================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString       db              'Hello World!', CR, LF
MsgLength       equ             $-MsgString
DataSg          EndS

;=====================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength
                        mov             dx, offset MsgString
                        int             21h

                        mov             ax, 4c00h
                        int             21h
Main            EndP
```

```
CodeSg          EndS
End             Main
```

Save the file in the notepad editor in a directory of your choice with the name “Sample1.asm” (eg. **C:\ml\Sample1\Sample1.asm**).

Bring up a command window. To do this in Windows XP, click on the “Strart” menu and then click on “Run”. Then type “Cmd” in the dialog box and click “Ok”.

Navigate to the directory where you saved the file using the Notepad editor. If you saved it in the suggested directory above, you can type “**cd \ml\sample1**”.

Now invoke the assembler by typing “**ml /omf sample.asm**”. Assuming there are no errors, this will generate a new file in that directory called “sample1.obj”. This has the raw binary code and is referred to as the object file.

We need to convert the object file to a format that is recognized by the operating system’s loader. The most common format used in the Windows operating system is called the Portable Executable (PE). The Linker is the tool that will generate a binary in that format.

To invoke the linker, type “**link sample1.obj**” in the same directory. The linker will prompt you for a few options. You can select the default options by simply entering return each time. The link will then generate a file called “**Sample1.exe**”.

Sample1.exe is your first program that you can ask the Operating System (OS) to load and run by simply typing “Sample1.exe” in the Command window while you are in the directory where that file is located.

If everything has gone according to plan you should see the “Hello World!” message printed on the screen.

Now let us go back and look at our program.

The Stack, Data and Code declarations must be familiar to you based on the discussion in the previous section.

The Code segment uses the DOS interrupt to print a message to the Standard Output. Let us study this in a bit more detail.

```
mov             ah, DOSWrite
mov             bx, StdOut
mov             cx, MsgLength
mov             dx, offset MsgString
int             21h
```

The first thing we do is move the DOS Write command to the AH register. Then we move the Standard Output identifier to the BX register. We move the length of the message we wish to print to the CX register. Then we point the DX register to the starting address of the message string. Finally we execute the interrupt (INT) instruction with an argument of 21h (DOS interrupt). This will cause the operating system to invoke the DOS interrupt handler. This handler will use the information supplied in the AH, BX, CX and DX registers to print the message defined by MsgString. In our case that is “Hello World!”.

1.4 – Debugging a program

Once you get through the Assembler and linker phase and create an executable file, there is always an urge to run the program and see if it behaves as you expect it to. It is advisable to use a tool called a "**debugger**" to walk through the code to ensure that the logic that is being executed is exactly what was intended. A debugger allows you to walk over (trace) individual instructions and confirm the output of each instruction.

For 16-bit x86 executables, the command window in Windows XP had a built-in debugger called "DEBUG".

To load "Sample1.exe" in the debugger, type "**debug sample1.exe**" in the directory where sample1.exe is located. This will give you a prompt like "-". At the prompt you can type "?" to get a list of commands that the debugger understands. You should see the following options.

```
-?
assemble     A [address]
compare      C range address
dump         D [range]
enter        E address [list]
fill         F range list
go           G [=address] [addresses]
hex          H value1 value2
input        I port
load         L [address] [drive] [firstsector] [number]
move         M range address
name         N [pathname] [arglist]
output       O port byte
proceed      P [=address] [number]
quit         Q
register     R [register]
search       S range list
trace        T [=address] [value]
unassemble   U [range]
write        W [address] [drive] [firstsector] [number]
allocate expanded memory        XA [#pages]
deallocate expanded memory      XD [handle]
map expanded memory pages       XM [Lpage] [Ppage] [handle]
display expanded memory status  XS
-_
```

Let us use the unassemble command to read the code that is about to be executed. Type "u 0 20". This asks the debugger to display 0x20 bytes starting at address 0. This will display the following code for sample1.exe

```
-u 0 20
0B8F:0000 B88E0B        MOV     AX,0B8E
0B8F:0003 8ED8          MOV     DS,AX
0B8F:0005 B440          MOV     AH,40
0B8F:0007 BB0100        MOV     BX,0001
0B8F:000A B90E00        MOV     CX,000E
0B8F:000D BA0000        MOV     DX,0000
0B8F:0010 CD21          INT     21
0B8F:0012 B8004C        MOV     AX,4C00
0B8F:0015 CD21          INT     21
0B8F:0017 4E            DEC     SI
0B8F:0018 4E            DEC     SI
0B8F:0019 42            INC     DX
0B8F:001A 3039          XOR     [BX+DI],BH
0B8F:001C F0            LOCK
0B8F:001D 0200          ADD     AL,[BX+SI]
0B8F:001F 0000          ADD     [BX+SI],AL
-
```

Note that the CS register is set to 0x0B8F. The first instruction at address offset 0x0000 is "B88E0B". "B8" is a "MOV AX" instruction. The "8E0B" is the data for the move instruction.

Why does the data show “8EOB” when the code had “0B8E”? This has to do with the order in which bytes are stored in memory. There are 2 common formats – “**little endian**” and “**big endian**”.

Little endian refers to the format where the least significant byte (little end) is represented at the first (lowest) address and each more significant byte is represented at the next higher address. This representation requires a reversal of bytes when read by the human eye because addresses are often displayed in increasing sequence and yet we expect more significant bytes to be displayed first.

Big endian is the opposite of little endian. Here the most significant byte (big end) is represented at the first (lowest) address.

Intel uses the “little endian” representation. Most other processors use the “big endian” representation.

Note that our linker has determined that our “DataSg” is located at 0x0B8E”. The first instruction in our program was “MOV AX, DataSg”. The debugger shows this as “MOV AX, 0x0B8E”.

The second “INT 21” is our last instruction. The instructions beyond that are not relevant to us.

Let us now walk over each instruction and see how it impacts the registers. To do this type “t” for trace.

```
-t

AX=0B8E  BX=0000  CX=0484  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B71  ES=0B71  SS=0B81  CS=0B8F  IP=0003   NV UP EI PL NZ NA PO NC
0B8F:0003 8ED8          MOV     DS,AX
-t

AX=0B8E  BX=0000  CX=0484  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=0005   NV UP EI PL NZ NA PO NC
0B8F:0005 B440          MOV     AH,40
-t

AX=408E  BX=0000  CX=0484  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=0007   NV UP EI PL NZ NA PO NC
0B8F:0007 BB0100        MOV     BX,0001
-t

AX=408E  BX=0001  CX=0484  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=000A   NV UP EI PL NZ NA PO NC
0B8F:000A B90E00        MOV     CX,000E
-t

AX=408E  BX=0001  CX=000E  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=000D   NV UP EI PL NZ NA PO NC
0B8F:000D BA0000        MOV     DX,0000
-t

AX=408E  BX=0001  CX=000E  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=0010   NV UP EI PL NZ NA PO NC
0B8F:0010 CD21          INT     21
-t =12

AX=4C00  BX=0001  CX=000E  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=0015   NV UP EI PL NZ NA PO NC
0B8F:0015 CD21          INT     21
-t =17

AX=4C00  BX=0001  CX=000E  DX=0000  SP=00C8  BP=0000  SI=FFFF  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B8F  IP=0018   NV UP EI NG NZ AC PE NC
0B8F:0018 4E            DEC     SI
-
```

Note that when the first “MOV AX, 0B8E” is executed, the AX register shows “0B8E”. and the IP register show “0003”. This means the next instruction to be executed is at an offset of “3”. The debugger displays the instruction at this address for us.

Trace through the subsequent instructions and confirm that what you see in the registers is what you would expect.

In general most debuggers will have problems tracing an interrupt operation. So walk over to the next instruction when you are about to execute an interrupt. To do this use the “-t [=Address]” instruction to ask the debugger to go to a particular address. You can also use the “go” (g) instruction for this purpose.

You can play with the other debug instructions to get a feel for the power of the debugger. Note that some of the instructions allow you to change the code on the fly. Use these with caution.

Finally use the “q” command to exit the debugger.

1.5 – Interacting with the User

In almost any computer program, there is a need to get input from the user of the program. The program's behavior is often dependent on the input that is supplied by the user. In this section we will write a sample that gets input from the user using the Standard Input used by the DOS operating system.

Before we proceed with this sample, I encourage you to decide on a text editor that is more appropriate than notepad for coding purposes. At a minimum, you will need an editor that displays line numbers. If you are unaware of any editor, you can use the Microsoft Visual C++ express edition that is a free download at http://msdn2.microsoft.com/en-us/express/default.aspx.

The most efficient way to master any programming language is to practice writing your own sample programs. Hence I encourage you to study the samples provided in each of these sections and attempt to duplicate their behavior on your own by using the same or similar instructions.

Another very useful programming technique is to do incremental additions. For example, with the sample below, you can first try and put up a prompt to the user. Then try and collect information from the user and then finally try and display the received input. At each of these interim stages, assemble and link your program to confirm that it is behaving as you would expect.

The only new construct that is introduced in the sample below is the use of the DOSRead function code. This code indicates to DOS that you are asking it to read input from the user. The number of characters read by DOS is going to be available in the AX register. This is also referred to as the "return value". In the x86 assembler, the return value from any call is usually passed using the AX register.

Once you have studied the following sample, assemble and link the sample using the "ml /omf sample2.asm" and "link sample2.asm" commands (assuming you choose "sample2.asm" as your filename).

You may observe that in the sample below, I have used the "xor ax, ax" instruction when I wanted to zero the contents of a register. An "xor" instruction is an exclusive-or operation. So if you apply that operation to the same register in the source and destination operand fields, you are bound to make the contents of the register to be zero.

You may wonder why I chose an "xor" over a more direct "mov ax, 0" instruction. This has to do with efficiency. In the older processors, a "mov" instruction from a memory to a register would use 4 clock cycles of the CPU, whereas an xor usually cost only 2 clock cycles. Needless to say these sorts of savings are not worth much (if anything at all) with the increased clock speeds and more efficient instructions of modern processors.

```
;========================================================================
; Sample2.asm
; STACK Segment
StackSg         Segment Para    Stack   'Stack'
                db              100     dup(?)
StackSg         EndS

;========================================================================
; DATA Segment
DataSg          Segment Para    'Data'

CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
DOSRead         equ             3Fh             ; DOS function code to read
StdOut          equ             1               ; Standard output file number
StdIn           equ             0               ; Standard input file number

; Prompt message and its length
EnterMsg        db              CR, LF, 'Please enter your name: '
EnterMsgLen     equ             $-EnterMsg

; Greeting message and its length
```

```
Greeting        db              'Hello '
GreetingLen     equ             $-Greeting

; User's name and its maximum possible length
UserName        db              100     dup(?)
MaxLen          equ             $-UserName

; User's name length
NameLen dw              1       dup(?)

; Finish message and its length
FinishMsg       db              'Nice to meet you. ', CR, LF
FinishMsgLen    equ             $-FinishMsg

DataSg          EndS

;=================================================================
; CODE Segment
CodeSg          Segment         Para    'Code'
                Assume          cs:CodeSg, ss:StackSg, ds:DataSg
Main    Proc            Far

                ;Set up Data segment   register
                mov             ax, DataSg
                mov             ds, ax

                ;Ask user to enter their name
                xor             ax, ax          ;Initialize ax to zero
                mov             ah, DOSWrite    ;Set ah to DOS Write function code
                mov             bx, StdOut      ;Set bx to Standard Output handle
                mov             cx, EnterMsgLen         ;Set cx to the Prompt Message length
                mov             dx, offset EnterMsg     ;Set ds to address of prompt buffer
                int             21h                     ;Issue DOS interrupt


                ;Read user's name and save it.
                xor             ax, ax                  ;Initialize ax to zero
                mov             ah, DOSRead             ;Set ah to DOS Read function code
                mov             bx, StdIn               ;Set bx to Standard Input handle
                mov             cx, MaxLen              ;Set cx to user name buffer length
                mov             dx, offset UserName     ;Set dx to address of name buffer
                int             21h                     ;Issue DOS interrupt
                mov             [NameLen], ax           ;Save number of chars read


                ;Print the 'Hello' greeting.
                xor             ax, ax                  ;Initialize ax to zero
                mov             ah, DOSWrite            ;Set ah to DOS Write function code
                mov             bx, StdOut              ;Set bx to Standard output handle
                mov             cx, GreetingLen         ;Set cx to Greeting length
                mov             dx, offset Greeting     ;Set dx to Greeting address
                int             21h                     ;Issue DOS interrupt


                ;Print the user's name.
                xor             ax, ax                  ;Initialize ax to zero
                mov             ah, DOSWrite            ;Set ah to DOS Write function code
                mov             bx, StdOut              ;Set bx to Standard output handle
                mov             cx, NameLen             ;Set cx to the Name length
                mov             dx, offset UserName     ;Set dx to UserName address
                int             21h                     ;Issue DOS interrupt


                ;Print the Finish message
                xor             ax, ax                  ;Initialize ax to zero
                mov             ah, DOSWrite            ;Set ah to DOS Write function code
                mov             bx, StdOut              ;Set bx to Standard Output handle
                mov             cx, FinishMsgLen        ;Set cx to Finish msg length
                mov             dx, offset FinishMsg    ;Set dx to Finish msg address
                int             21h                     ;Issue DOS interrupt
```

```
                ;Return         control to DOS
                mov             ax, 4c00h
                int             21h

Main            EndP

CodeSg          EndS
                End     Main
```

2.0 – Jump Instructions

In the previous section we gained the knowledge to write sequential instructions that let us get input from a user and process it. While sequential instructions are the primary mechanism describing a set of steps, it can be very limiting in our ability to reuse our code.

Imagine how difficult it would be if we had to explicitly write the code to read user input every time we needed to get user input within a program. It would be so much easier if we could jump to the code that does the reading of user input every time we need to get user input.

The construct of Jump instructions is designed to do just that. They allow the coder to move around the program without the sequential processing limitation.

In this section we will study the common Jump instructions available in the x86 architecture.

2.1 – Unconditional Jumps (JMP)

An unconditional jump, as the name implies, allows the transfer of execution from one part of your program to another without any conditions.

All jump instructions work by altering the value of the IP register. There are four kinds of unconditional jumps in the 16-bit x86 architecture.

- Near Short Jump
- Near Long Jump
- FAR Short Jump
- FAR Long Jump

A short jump happens when a 1-byte value is provided to the Jump instruction. This 1-byte is sign added to the IP register to determine the next instruction address. Since the addition is a signed addition, a short jump only allows you to jump +/- 127 bytes from your current location (for a Near jump).

A long jump happens when a 2-byte value is provided to the Jump instruction. This allows you to go anywhere within the current code segment.

A NEAR jump refers to the case where the CS register remains unchanged.

A FAR jump involves switching between different code segments by not only changing the IP register but also the CS register. We won't worry too much about FAR jumps in this set of notes.

Below is a sample that shows the operation of a Near Short Jump. Once you assemble and link this sample, you should see the "Ooops! Forgot to say Hello!" message.

```
;=====================================================================
; Sample3.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;=====================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Hello World!', CR, LF
MsgLength1      equ             $-MsgString1
MsgString2      db              'Ooops! Forgot to say Hello!', CR, LF
MsgLength2      equ             $-MsgString2
DataSg          EndS

;=====================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength1
                        mov             dx, offset MsgString1

                        jmp             OOPS
PRINT:
```

```
                        int             21h
                        jmp             FINISH

OOPS:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength2
                        mov             dx, offset MsgString2
                        jmp             PRINT


FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP
CodeSg          EndS
End             Main
```

Follow the logic that exploits the unconditional jump instructions. Then load the binary in the debugger and check if a short or a long jump was used by the assembler. Note that a long jump will work in all situations, but a short jump is limited.

2.2 – Compare instruction (CMP)

The compare instruction is essentially a subtract instruction that does not alter the value of the operands but impacts the value of the flags registers just like a subtract instruction would. We study the compare instruction because its impact on the flags register is exploited by many Jump instructions.

You can type the following instructions into one of your earlier samples and trace each instruction and see how it impacts the flags register.

```
mov             ax, 9
mov             bx, 8
mov             cx, 9

cmp             ax, bx

cmp             bx, ax

cmp             ax, cx
```

The following debugger output shows that the three “mov” instructions did not impact the flags register.

The “cmp ax, bx” involves “9 - 8”. In both signed and unsigned arithmetic, this leads to a +1. So we expect the “PL”, “NZ” and “NC” flags to be set.

The “cmp bx, ax” involves “8 – 9”. In signed arithmetic, this leads to “-1”. So we expect the “NG”, “NZ”, and “CY” flags to be set.

The “cmp ax, cx” involves “9 – 9”. This will yield “0” and hence the “ZR” flag is set.

```
AX=0B8E  BX=0000  CX=057C  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=0005   NV UP EI PL NZ NA PO NC
0B91:0005 B80900        MOV     AX,0009
-t

AX=0009  BX=0000  CX=057C  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=0008   NV UP EI PL NZ NA PO NC
0B91:0008 BB0800        MOV     BX,0008
-t

AX=0009  BX=0008  CX=057C  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=000B   NV UP EI PL NZ NA PO NC
0B91:000B B90900        MOV     CX,0009
-t

AX=0009  BX=0008  CX=0009  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=000E   NV UP EI PL NZ NA PO NC
0B91:000E 3BC3          CMP     AX,BX
-t

AX=0009  BX=0008  CX=0009  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=0010   NV UP EI PL NZ NA PO NC
0B91:0010 3BD8          CMP     BX,AX
-t

AX=0009  BX=0008  CX=0009  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=0012   NV UP EI NG NZ AC PE CY
0B91:0012 3BC1          CMP     AX,CX
-t

AX=0009  BX=0008  CX=0009  DX=0000  SP=00C8  BP=0000  SI=0000  DI=0000
DS=0B8E  ES=0B71  SS=0B81  CS=0B91  IP=0014   NV UP EI PL ZR NA PE NC
```

2.3 – Zero or Equality Jumps (JZ, JE, JNZ, JNE)

.The Jump Zero (“JZ”) and the Jump Equal (“JE”) instructions do the exact same thing – they both check if the ZERO flag is set and if it is, they jump to the tag provided in the operand.

Similarly the Jump Not Zero (“JNZ”) and the Jump Not Equal (“JNE”), jump to the tag provided in the operand if the ZERO flag is not set.

The sample below demonstrates the use of these instructions. Note that I have used the “JE” and “JNE” instructions. You can replace these with “JZ” and “JNZ” respectively, without altering the behavior.

Instead of moving “8” to the ax register, change the code to move “9” into the ax register and confirm that the jump to “JUMP_ZERO” tag happens.

Note that the Jump instructions do not change the value of the flags register and so we can have multiple conditional jumps subsequent to the compare instruction.

```
;=====================================================================
; Sample4.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;=====================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Hello World!', CR, LF
MsgLength1      equ             $-MsgString1
MsgString2      db              'Last CMP instruction set the Zero Flag!', CR, LF
MsgLength2      equ             $-MsgString2
MsgString3      db              'Last CMP instruction did not set the Zero Flag!', CR, LF
MsgLength3      equ             $-MsgString3

DataSg          EndS

;=====================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             ax, 8
                        mov             bx, 9
                        cmp             bx, ax
                        je              JUMP_ZERO
                        jne             JUMP_NOT_ZERO

JUMP_ZERO:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength2
                        mov             dx, offset MsgString2
                        int             21h
                        jmp             FINISH

JUMP_NOT_ZERO:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
```

```
                        mov             cx, MsgLength3
                        mov             dx, offset MsgString3
                        int             21h
                        jmp             FINISH


FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP
CodeSg          EndS
End             Main
```

2.4 – Unsigned Jumps (JA, JAE, JB, JBE)

The Unsigned jumps use the ZERO and CARRY flags.

Jump if Above (“JA”) instruction jumps to the tag provided in the operand if both the ZERO flag and the CARRY flag are not set. If the ZERO flag is set, we know the numbers used in the last compare were equal. If the CARRY flag was set we know the last compare involved subtracting a larger number from a smaller number. If neither of these flags were set, the last compare involved subtracting a smaller number from a larger number. In this case the “JA” instruction will jump to the tag provided in the operand.

Similarly Jump if Above or Equal (“JAE”) causes a jump if either the Zero flag is set or if CARRY flag is not set.

Jump if Below (“JB”) instruction jumps to the tag provided in the operand if the ZERO flag is not set but the CARRY flag is set.

Jump if Below or Equal (“JBE”) causes a jump if either the Zero flag is set or if the CARRY flag is set.

Another way to look at these instructions is to consider a compare between unsigned numbers. If the first number is equal to the second number, the ZERO flag is set. So both the “JAE” and “JBE” will cause a jump in this case.

If the first number is greater than the second number, both the ZERO flag and the CARRY flag are not set. In this case both the “JA” and “JAE” instructions will cause a jump.

If the first number is less than the second number, the ZERO flag is not set, but the CARRY flag is set. In this case both the “JB” and “JBE” will cause a jump.

Hence these jump instructions are designed to compare unsigned numbers.

In the sample below change the values in the ax and bx register before the compare instruction and try and explain why it is not possible to jump to the “JUMP_BELOW_EQUAL” tag irrespective of the values used in ax and bx.

```
;=====================================================================
; Sample5.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;=====================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Jump Above!', CR, LF
MsgLength1      equ             $-MsgString1
MsgString2      db              'Jump Above or Equal!', CR, LF
MsgLength2      equ             $-MsgString2
MsgString3      db              'Jump Below!', CR, LF
MsgLength3      equ             $-MsgString3
MsgString4      db              'Jump Below or Equal!', CR, LF
MsgLength4      equ             $-MsgString4

DataSg          EndS

;=====================================================================
; CODE Segment
```

```
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             ax, 8
                        mov             bx, 8
                        cmp             bx, ax
                        ja              JUMP_ABOVE
                        jae             JUMP_ABOVE_EQUAL
                        jb              JUMP_BELOW
                        jbe             JUMP_BELOW_EQUAL

JUMP_ABOVE:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength1
                        mov             dx, offset MsgString1
                        int             21h
                        jmp             FINISH

JUMP_ABOVE_EQUAL:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength2
                        mov             dx, offset MsgString2
                        int             21h
                        jmp             FINISH

JUMP_BELOW:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength3
                        mov             dx, offset MsgString3
                        int             21h
                        jmp             FINISH

JUMP_BELOW_EQUAL:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength4
                        mov             dx, offset MsgString4
                        int             21h
                        jmp             FINISH


FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP
CodeSg          EndS
End             Main
```

2.5 – Signed Jumps (JG, JGE, JL, JLE)

The Signed jumps use the SIGN, ZERO and OVERFLOW flags.

Jump if Greater (“JG”), Jump if Greater or Equal (“JGE”), Jump if Less (“JL”) and Jump if Less or Equal (“JLE”) are very similar to the “JA”, “JAE”, “JB”, “JBE” instructions respectively. However these apply to signed numbers.

The sample below shows the use of signed jumps.

```
;=========================================================================
; Sample6.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;=========================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Jump Greater!', CR, LF
MsgLength1      equ             $-MsgString1
MsgString2      db              'Jump Greater or Equal!', CR, LF
MsgLength2      equ             $-MsgString2
MsgString3      db              'Jump Less!', CR, LF
MsgLength3      equ             $-MsgString3
MsgString4      db              'Jump Less or Equal!', CR, LF
MsgLength4      equ             $-MsgString4

DataSg          EndS

;=========================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             ax, -4
                        mov             bx, -8
                        cmp             bx, ax
                        jg              JUMP_GREATER
                        jge             JUMP_GREATER_EQUAL
                        jl              JUMP_LESS
                        jle             JUMP_LESS_EQUAL

JUMP_GREATER:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength1
                        mov             dx, offset MsgString1
                        int             21h
                        jmp             FINISH

JUMP_GREATER_EQUAL:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength2
                        mov             dx, offset MsgString2
                        int             21h
                        jmp             FINISH
```

```
JUMP_LESS:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength3
                        mov             dx, offset MsgString3
                        int             21h
                        jmp             FINISH

JUMP_LESS_EQUAL:
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength4
                        mov             dx, offset MsgString4
                        int             21h
                        jmp             FINISH


FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP
CodeSg          EndS
End             Main
```

2.6 – Other Jumps (JC, JNC, JO, JNO, JS, JNS, JCXZ)

In addition to the Unconditional, Zero, Unsigned and Signed jumps, there are three other jumps that target specific flags.

Jump if Carry ("JC") executes a jump if the CARRY flag is set. Similarly Jump if No Carry ("JNC") jumps if the CARRY flag is not set.

Jump if Overflow ("JO") executes a jump if the Overflow flag is set. Similarly Jump if No Overflow ("JNO") jumps if the Overflow flag is not set.

Jump if Sign ("JS") executes a jump if the Sign flag is set. Similarly Jump if No Sign ("JNS") jumps if the Sign flag is not set.

Yet another jump instruction that can be very useful in implementing a loop with a specific number of iterations is the Jump if CX Zero ("JCXZ") instruction.

Below is a sample showing how the "JCXZ" instruction can be used to implement a loop. Note that I have used three new instructions in this sample – "PUSH", "POP" and "DEC".

The "**PUSH**" and "**POP**" instructions are mechanisms to save data on the stack. In the sample below, the value of the "CX" register was about to be overwritten. Hence I saved its value on the stack with the "PUSH" instruction and later restored it with the "POP" instruction.

The "PUSH" instruction effectively translates to the following sub instructions;

```
sub     sp, 2
mov     [sp], <word data to be saved>
```

First we decrement the stack pointer (note the stack grows down), then we save the data in the new location of the stack pointer. We will talk more about the indirect addressing mode (the square brackets around sp) later on.

The "POP" instruction is the inverse of the "PUSH". The following sub instructions effectively sum up the "POP instruction.

```
mov     <word data to be retrieved>, [sp]
add     sp, 2
```

The decrement ("**DEC**") instruction simply subtracts 1 from the operand. Similarly the increment ("**INC**") instruction adds 1 to the operand.

```
;===========================================================================
; Sample7.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;===========================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Looping... ', CR, LF
MsgLength1      equ             $-MsgString1
DataSg          EndS

;===========================================================================
```

```
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             cx, 10

START_LOOP:
                        jcxz    FINISH
                        push    cx

                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength1
                        mov             dx, offset MsgString1
                        int             21h

                        pop             cx
                        dec             cx

                        jmp             START_LOOP


FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP
CodeSg          EndS
End             Main
```

3.0 – Loop Instructions

In the previous two sections we learnt two constructs in writing code - sequential processing and the ability to jump to addresses that are not necessarily in sequence. In this section we introduce a third construct in programming called looping.

Loop instructions allow the user to repeatedly execute a set of instructions until one or more conditions are met. In some respects a loop instruction is a special case of a Jump instruction, but the construct is powerful enough to warrant a dedicated section. Besides, most processors including the x86 processor, have specialized instructions for looping.

3.1 – Basic Loop (LOOP)

Our knowledge of the Jump if Not Zero (“JNZ”) and jump if Zero (“JZ”) instructions will allow us to implement a very basic loop.

The sample below is a slight modification of the sample in the previous section. There are two loops in this sample.

In the first loop I have replaced the “JCXZ” instruction with a JNZ. I am exploiting the fact that the “DEC” instruction sets the ZERO flag.

The second loop introduces the “**LOOP**” instruction. The “LOOP” instruction decrements the CX register by one and loops to the operand label if the CX register is not zero. Note that it does not alter the flags register.

```
;=========================================================================
; Sample8.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                db              200     dup(?)
StackSg         EndS

;=========================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Looping 1... ', CR, LF
MsgLength1      equ             $-MsgString1
MsgString2      db              'Looping 2... ', CR, LF
MsgLength2      equ             $-MsgString2

DataSg          EndS

;=========================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                mov             ax, DataSg
                mov             ds,ax

                mov             cx, 10
                cmp             cx,0    ;check to confirm we need to enter loop
                jz              FINISH

START_LOOP1:
                ; This loop uses a "dec" and "jnz"
                push    cx

                mov             ah, DOSWrite
                mov             bx, StdOut
                mov             cx, MsgLength1
                mov             dx, offset MsgString1
                int             21h

                pop             cx
                dec             cx
                jnz             START_LOOP1

                ; We have come out of the first loop.
                ; Re-initialize the loop count
                mov             cx, 10
                cmp             cx,0    ;check to confirm we need to enter loop
```

```
                        jz              FINISH

START_LOOP2:
                        ; This loop replaces the "dec" and "jnz" with a "loop"
                        push    cx

                        mov             ah, DOSWrite
                        mov             bx, StdOut
                        mov             cx, MsgLength2
                        mov             dx, offset MsgString2
                        int             21h

                        pop             cx
                        loop    START_LOOP2

FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP
CodeSg          EndS
End             Main
```

3.2 – Other Loops (LoopE,LoopZ,LoopNE,LoopNZ)

The Loop if Zero ("**LOOPZ**") or Loop if Equal ("**LOOPE**") instructions are similar to the "LOOP" instruction with one additional condition – they only loop if the ZERO flag is set.

The Loop if Not Zero ("LOOPNZ") or Loop if Not Equal ("LOOPNE") instructions are also similar to the "LOOP" instruction with the additional condition that they only loop if the ZERO flag is **not** set.

These instructions are handy in cases where the number of loops is not always a constant but rather based on a condition.

4.0 – Calling Procedures

In the previous sections we discussed Sequential, Jump and Loop instructions. The main benefits of the latter two programming constructs are that it allows a coder to reuse code that is written for generic purposes (eg. reading user input).

While a Jump instruction allows a coder to jump to any location within the code, it does not provide a mechanism to return to the location from which the Jump occurred once the generic code is executed. This brings us to the fourth programming construct – Procedures.

Procedure calling is designed specifically to remedy the problem of knowing where and in what state to return to, once the generic code is executed.

4.1 – Calling a Procedure and Returning

A Procedure is essentially a set of instructions to do a specific task. For example in most of the samples thus far we had a need to display an output. We would have been more effective to isolate that code to a procedure and call the procedure each time we needed to display an output, rather than duplicate the code each time.

In the sample below, I have written a procedure called “StrOut” to display a string. The caller is expected to pass in the offset to the string to be displayed in the “SI” register and the count of the number of characters in the string in the “CX” register. These are referred to as the “input” parameters to the procedure. If the procedure was expected to return a value back to the caller, that would be referred to as an “output” parameter. By convention, output parameters are passed in the “AX” register.

The “**CALL**” instruction does 2 things – it saves the IP register on the stack and jumps to the procedure.

The “**RET**” instruction pops the previously saved IP register.

If the procedure is going to save anything on the stack (and it almost always will), it is very important that it pops everything back before calling the “RET” instruction, else an invalid value will be popped into the IP register.

```
;=====================================================================
; Sample10.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;=====================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Procedure StrOut Called.. ', CR, LF
MsgLength1      equ             $-MsgString1
DataSg          EndS

;=====================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             cx, MsgLength1
                        mov             si, offset MsgString1
                        call    StrOut

FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP

;*************************************************************
;StrOut: Display a string to StdOut
;Input SI: points to string to be displayed
;Input CX: has the number of characters to display
;
;*************************************************************
StrOut          Proc    Far
```

```
                        mov             ah, DOSWrite
                        mov             bx, StdOut
                                                    ;cx should already be set by caller
                        mov             dx, si      ; si was passed in by caller
                        int             21h

                        ret

StrOut          EndP


CodeSg          EndS
End             Main
```

4.2 – Saving and Restoring Context in Procedures

It is essential that a procedure does what it has promised to do without causing any side effects to the caller of the procedure. The caller of the Procedure will be unaware of the resources that the procedure needs to complete its task. Hence it is the responsibility of the procedure to ensure that it does not trample over any of the resources that the caller may already be using.

A common practice in this regard is to save all the registers on the stack at the beginning of a procedure and restore them back at the end of a procedure. Technically it is sufficient to save just the registers that the procedure is going to use. However, it is best practice to save and restore all registers to avoid having to identify all usages (Note sometimes it is not obvious which registers are being used).

I have modified the previous sample to ensure that the first thing done in the procedure is to save registers that will be tampered with on the stack. Similarly the last thing that is done is to restore those registers. **Never forget to pop everything that was previously pushed on the stack**.

"**PUSHF**" and "**POPF**" are instruction to push and pop the flags register respectively.

The stack operates on a last-in first-out basis. **Hence the order in which the pop occurs is the reverse order in which the push occurred.**

```
;===========================================================================
; Sample11.asm
; STACK Segment
StackSg         Segment Para Stack 'Stack'
                        db              200     dup(?)
StackSg         EndS

;===========================================================================
; DATA Segment
DataSg          Segment Para 'Data'
dstart          equ $
CR                      equ             13      ; Return
LF                      equ             10      ; Line Feed
DOSWrite        equ             40h             ; DOS function code to write
StdOut          equ             1               ; Standard output file number
MsgString1      db              'Procedure StrOut Called.. ', CR, LF
MsgLength1      equ             $-MsgString1
DataSg          EndS

;===========================================================================
; CODE Segment
CodeSg          Segment Para 'Code'
                        Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main            Proc    Far
                        mov             ax, DataSg
                        mov             ds,ax

                        mov             cx, MsgLength1
                        mov             si, offset MsgString1
                        call    StrOut

FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP

;***************************************************************
;StrOut: Display a string to StdOut
;Input SI: points to string to be displayed
;Input CX: has the number of characters to display
;
```

```
;*****************************************************************
StrOut          Proc    Far

                        push    ax
                        push    bx
                        push    cx
                        push    dx
                        pushf

                        mov             ah, DOSWrite
                        mov             bx, StdOut
                                                        ;cx should already be set by caller
                        mov             dx, si          ; si was passed in by caller
                        int             21h

                        popf
                        pop             dx
                        pop             cx
                        pop             bx
                        pop             ax

                        ret

StrOut          EndP


CodeSg          EndS
End             Main
```

5.0 – Addressing Modes

Accessing data in registers and in memory is an essential part of assembler programming. Every processor allows for different methods to specify source and destination addresses for various instructions.

In this section we will cover the most common addressing modes used by the x86 processor.

5.1 – Register Addressing Mode

This involves accessing data in registers. It is a very common and straight forward technique and we have used it in almost all the samples thus far. The following is an example of Register Addressing mode.

```
mov     ax, bx
```

Here we are moving the contents of the “BX” register into the “AX” register.

5.2 – Immediate Addressing Mode

When a data value is a constant, it can be made available as an operand. The following is an example of an Immediate Addressing Mode.

```
mov     cx, 9
```

This instruction moves “9” to the “CX” register.

5.3 – Direct Addressing Mode

If we wanted to access memory at a known address, we could enclose the known address in square brackets and offer that as our operand. The following is an example of Direct addressing.

```
mov     cx, [1000]
```

Here we move the contents at offset 1000 into the Data Segment (defined by the “DS” register) into the “CX” register.

5.4 – Register Indirect Addressing Mode

Sometimes a register contains an address offset into the data segment. In these cases we can access the value at that address by enclosing the register in square brackets and using that as our operand.

```
mov     cx, [bx]
```

Here we move the contents of memory whose address is in the “BX” register.

5.5 – Register Indirect Indexed Addressing Mode

The register indirect addressing mode can be extended to access elements of an array for example, by using the following syntax;

```
mov     cx, [bx+2]
```

This will fetch the contents of memory at the address defined by the “BX” register plus 2.

6.0 – More complex Instructions

In this last section on assembler programming, we will study a few more complex instructions available to us in the x86 processor. We will then conclude by trying to put together everything we have learned by writing a set of procedures to perform string operations.

6.1 – Bit Operations

OR

The “OR” instruction is used to turn on individual bits in the destination operand based on a mask that is provided as the source operand.

```
or al, 00001111b
```

In the example above, we ensure that the lowest 4 bits in the AL register are set.

AND

The “AND” instruction is used to turn off individual bits in the destination operand based on a mask that is provided as the source operand.

```
and ax, 0Fh
```

In the example above, only the lowest 4 bits of the AX register are preserved. All other bits are set to zero.

XOR

The Exclusive OR operation inverts all the bits in the destination operand, for which the corresponding mask bits are set.

```
xor ax, 0Fh
```

If AX had the value 0x29 before the XOR operation in the above example, AX will change to 0x26 after the XOR operation. Note the “2” is unchanged. “9” in binary is “1001”. If we invert those bits, we get “0110” or “6” in hexadecimal.

TEST

The “TEST” instruction is very similar to the “AND” operation with the exception that it does not alter the destination operand. Instead it only sets the flags register to indicate if the operation resulted in a zero or non-zero result. This can then be used by the JUMP instructions. The “TEST” is the BOOLEAN equivalent of a “CMP” instruction.

SHL

The Shift Left (“SHL”) instruction shifts the bits in the destination operand to the left. The number of bits to be shifted is provided as the source operand. During each 1 bit left shift, the bit shifted out of the most significant bit is placed in the Carry bit of the flags register and a bit ‘0” is shifted into the least significant bit.

Note that shifting left by 1 bit is the equivalent of multiplying by 0x02.

SHR

The Shift Right (“SHR”) instruction shifts the bits in the destination operand to the right. The number of bits to be shifted is provided as the source operand. During each 1 bit right shift, the bit shifted out of the least significant bit is placed in the Carry bit of the flags register and a bit “0” is shifted into the most significant bit.

Note that shifting right by 1 bit is the equivalent of unsigned division by 0x02.

SAR

The Shift Arithmetic Right (“SAR”) is similar to the “SHR” instruction with the exception that that the most significant bit will always remain unchanged. Since the most significant bit is the sign bit, this instruction can be useful for signed division.

ROL

The Rotate Left (“ROL”) instruction rotates the bits in the destination operand in the counter clockwise direction. The number of bit shifts is defined by the source operand. During each shift to the left, the most significant bit defines the value of the Carry flag.

ROR

The Rotate Right (“ROR”) instruction rotates the bits in the destination operand in the clockwise direction. The number of bit shifts is defined by the source operand. During each shift to the right, the least significant bit defines the value of the Carry flag.

RCL

The Rotate Left Through Carry (“RCR”) is similar to the ROL instruction with the exception that the bits are rotated counter clockwise through the Carry Flag.

RCR

The Rotate Right Through Carry (“RCR”) is similar to the ROR instruction with the exception that the bits are rotated clockwise through the Carry Flag.

6.2 – Arithmetic Operations

ADD

The Add instruction adds the source and destination operands and places the result in the destination operand.

ADC

The Add with Carry (“ADC”) instruction adds the source, destination and the “Carry Flag” bit and places the result in the destination operand.

SUB

The Subtract (“SUB”) instruction subtracts the source from the destination and places the result in the destination operand.

SBB

The Subtract with Borrow (“SBB”) subtracts the sum of the source and the “Carry flag” from the destination and places the result in the destination operand.

CBW

The Convert Byte to Word (“CBW”) instruction uses the input BYTE in the AL register and converts it to a WORD in the AX register.

This is a useful instruction for converting both signed and unsigned BYTEs to WORDs. It effectively performs a sign extension by ensuring that the most significant bit in the BYTE is replicated in all the extra bits provided by the WORD.

CWD

The Convert Word to Double (“CWD”) instruction is very similar to the “CBW” instruction. It takes the AX register as the input WORD and converts it to a DWORD in DX and AX registers where DX has the high WORD and AX has the lower WORD.

Converting a WORD to a BYTE

If you used WORD registers for a calculation but wish to store the result in a BYTE variable, this is feasible as long as your result fits in a BYTE. The following instructions will help confirm that it is possible to store the result in a WORD register into a BYTE variable. Assume the AX register holds the WORD value.

```
mov     bh, ah          ; Save the high byte of AX into BH.
cbw                     ; Convert BYTE in AL to a WORD in AX
cmp     ah, bh          ; Compare the new AH with the previously saved BH
jne     OVERFLOW        ; They are not equal – so the value will not fit in a BYTE.
mov     BYTE_VAR, al    ; Value will fit in byte – save it to a BYTE variable.
```

This technique will work for both signed and unsigned arithmetic.

STC, CLC and CMC

Sometimes you may want to directly control the value of the Carry flag when performing arithmetic operations.

The Set Carry (“STC”) allows you set the Carry flag.

The Clear Carry (“CLC”) allows you to clear the Carry flag.

The Complement Carry (“CMC”) allows you to inverts the current value of the Carry flag.

LEA

The Load Effective Address (“LEA”) instruction allows you to find the address offset of a memory variable. For example both the instructions below will yield the same address in the AX register.

```
lea     ax, MEM_VAR

mov     ax, OFFSET MEM_VAR
```

To use the “OFFSET” directive however, the address should be available at assembly time. If the address is only available at runtime, the “LEA” instruction must be used. In the example below, the contents of BX is added to the contents of SI and that result is added to 7 to give the offset that is placed in the AX register. Since the contents of BX and SI are only available at runtime, the “LEA” instruction must be used.

```
lea     ax, [BX+SI+7]
```

While the “LEA” instruction is generally used for address calculation, there is no reason why it can’t be used for arithmetic operations.

6.3 – Interrupt Operations

STI

The Set Interrupt (“STI”) instruction enables interrupts by setting the Interrupt Flag (IF) bit in the flags register.

CLI

The Clear Interrupt (“CLI”) instruction disables interrupts by clearing the Interrupt Flag (IF) bit in the flags register.

6.4 – String Operations

MOVSB

The Move Single Byte (“MOVSB”) instruction is a complex instruction that allows you to move a byte from a source address to a destination address. Effectively it does the following:

```
mov     al, BYTE PTR [si]       ; copy DS:SI to al
mov     BYTE PTR [di], al       ; move al to ES:DI
inc/dec si                      ; increment or decrement SI based on Direction flag
inc/dec di                      ; increment or decrement DI based on Direction flag
```

As you might guess, before using the “MOVSB” instruction, you need to ensure that ES, SI, DI and the Direction flag is set up correctly.

Note: If the Direction Flag is zero, DI and SI are incremented, else they are decremented.

STD and CLD

As noted in the “MOVSB” instruction, the value of the direction flag dictates if SI and DI are incremented or decremented by the “MOVSB” instruction.

The Set Direction (“STD”) instruction sets the Direction flag.

The Clear Direction (“CLD”) clears the Direction flag.

MOVSW

The “MOVSW” instruction is very similar to the “MOVESB” instruction but works on a WORD instead of a BYTE.

REP MOVSB

The Repeat MOVSB (“REP MOVSB”) instruction allows you to repeat the move instruction for a number of iterations defined by the CX register. Think of it as a loop instruction that decrements the CX register in each iteration and jumps out of the loop when CX hits zero.

REP MOVSW

Similar to “REP MOVSB”, but works on WORDs instead of BYTEs.

STOSB and **STOSW**

The Store String by Byte (“STOSB”) and Store String by Word (“STOSW”) allow you to store the value in a register into a memory address. Effectively, these instructions do half the work done by the MOVSB/MOVSW instructions – they move data from register to memory instead of memory to memory.

The STOSB instruction saves the byte in the AL register to the memory address defined by ES:[DI] and then increments or decrements the DI register by **one**, based on the Direction Flag.

The STOSW instruction saves the word in the AX register to the memory address defined by ES:[DI] and then increments or decrements the DI register by **two**, based on the Direction Flag.

REP STOSB and **REP STOSW**

The Repeat (“REP”) qualifier can also be used with the STOSB and STOSW instructions to repeat the instructions by a count defined by the CX register.

LODSB and **LODSW**

The Load String Byte (“LODSB”) and Load String Word (“LODSW”) allow you to load the value in a memory location into a register. These instructions do the other half of the work done by the MOVSB/MOVSW instructions.

The LODSB instruction loads a byte from DS:[SI] to the AL register and then increments or decrements the DI register by **one**, based on the Direction Flag.

The LODSW instruction loads a word from the DS:[SI] to the AX register and then increments or decrements the DI register by **two**, based on the Direction Flag.

CMPSB

The Compare String Byte (“CMPSB”) instruction compares two memory byte locations defined by DS:[SI] and ES:[DI]. It then sets the flags to indicate the result of the comparison and then increments or decrements the SI and DI registers, depending on the Direction Flag.

REPE and **REPNE**

The Repeat while Equal (“REPE”) and Repeat while Not Equal (“REPNE”) can be used in conjunction with the “CMPSB” instruction to compare a number of bytes defined by the CX register.

SCASB

The Scan String for Byte (“SCASB”), compares the BYTE in the AL register with the BYTE in ES:[DI] and sets the flags to reflect this comparison. It then increments or decrements the DI register, depending on the Direction Flag.

The REPE and REPNE qualifiers can also be used with “SCASB”.

6.5 – String Library Example

Sample 12 below uses the knowledge we have gained thus far to build a library of string functions.

As an exercise, first assemble the routines in this sample and then try to understand each routine.

```
;*****************************************************************************
; Sample12.asm
;*****************************************************************************
;This program demonstrates the use of subroutines to write re-usable code
;as well as the use of the string, bit and arithmetic mnemonics.
;
;Notes: The following subroutines are used
;              1) StrLen - Gets the length of a NULL terminated string
;              2) StrGet - Gets DOS user input
;              3) StrPut - Display output to DOS
;              4) StrUpr - Converts string to upper case
;              5) StrCmp - Compares two strings
;              6) StrCpy - Copies strings
;
;              Using the .286 mode to allow the use of "pusha" and "popa" mnemonics
;              to save registers.
;*****************************************************************************


.286
;*****************************************************************************
; STACK Segment
StackSg        Segment Para Stack 'Stack'
                      db             200     dup(?)
StackSg        EndS

;*****************************************************************************
; DATA Segment
DataSg         Segment Para 'Data'
dstart         equ $
CR                    equ            13      ; Return
LF                    equ            10      ; Line Feed
DOSRead        equ             3Fh            ; DOS function code to read
DOSWrite       equ             40h            ; DOS function code to write
StdInput       equ             0              ; Standard input file number
StdOutput      equ             1              ; Standard output file number
UPPER_MASK     equ             0DFh           ; Mask with 5th bit set to 0

InputBuffer    db              80 dup(?)      ;Input Buffer

CompareStr     db              'ABORT',0      ;NULL terminated ABORT string

OutputBuf      db              80 dup(?)      ;Output Buffer

DataSg         EndS

;*****************************************************************************
; CODE Segment
CodeSg         Segment Para 'Code'
                      Assume  cs:CodeSg,ss:StackSg,ds:DataSg
Main           Proc    Far
                      mov             ax, DataSg
                      mov             ds,ax

                      ;Test StrGet
                      lea             di, InputBuffer
                      call    StrGet

                      ;Test StrUpr
                      lea             di, InputBuffer
```

```
                        call    StrUpr

                        ;Test StrCmp
                        mov             ax, ds
                        mov             es, ax
                        lea             si, InputBuffer
                        lea             di, CompareStr
                        call    StrCmp
                        je              FINISH

                        ;Test StrCpy
                        mov             ax, ds
                        mov             es, ax
                        lea             si, InputBuffer
                        lea             di, OutputBuf
                        call    StrCpy

                        ;Test StrPut
                        lea             si, OutputBuf
                        call    StrPut


FINISH:
                        mov             ax, 4c00h
                        int             21h
Main            EndP



;*************************************************************************
;StrLen
;*************************************************************************
;Function: This routine determines the length of a NULL terminated string.
;
;Entry: DS:SI holds offset to the string buffer
;
;Exit: CX holds the string length
;
;Modifies:  CX and flags registers
;               All other registers left unchanged
;
;Notes: Allows for string up to 65535 chars
;*************************************************************************
StrLen          Proc            Far

                        ;Save Registers
                        push    es
                        push    di
                        push    ax
                        push    bx

                        ;Initialize cx to maximum possible value so that repne scasb can
                        ;keep counting until it finds the scanned char
                        mov             cl, 0FFh
                        mov             ch, 0FFh

                        mov             ax, ds          ;Since scasb works on ES:DI
                        mov             es, ax          ;set up ES to be DS
                        mov             di, si          ;and set up DI to SI
                        xor             ax, ax          ;set ax to zero - search for NULL
                        cld                             ;clear direction flag
                        mov             bx, di          ;save current value of DI
                        cli                             ;Disable interrupts (bug in 8086)
                        repne   scasb                   ;Scan for NULL in string
                        sti                             ;Enable interrupts
                        mov             cx, di          ;Evaluate index of NULL
                        sub             cx, bx          ;Get string length including NULL
                        dec             cx              ;remove NULL

                        ;Restore registers
                        pop             bx
```

```
                        pop             ax
                        pop             di
                        pop             es

                        ret

StrLen          EndP

;*************************************************************************
;StrGet
;*************************************************************************
;Function: This routine gets a string from the DOS input stream
;
;Entry: DS:DI holds offset to input buffer
;               CX holds the input buffer size
;
;Exit:  Input buffer holds NULL terminated string.
;
;Modifies: Registers left unchanged on exit
;
;Notes:
;*************************************************************************
StrGet          Proc    Far

                        ;Save registers
                        pusha

                        ;Read user input and save it in caller's buffer (DI)
                        xor             ax, ax          ;Initialize ax to zero
                        mov             ah, DOSRead     ;Set AH to DOS Read function code
                        mov             bx, StdInput    ;Set BX to Standard Input handle
                        mov             dx, di          ;Set DX to user input buffer address
                        int             21h             ;Issue DOS interrupt
                        mov             cx, ax          ;Save number of chars read

                        sub             cx, 2           ;Remove LF and CF chars
                        add             di, cx          ;Move DI to point to last char
                        mov             BYTE PTR[di],0 ;NULL terminate string

                        ;Restore registers
                        popa

                        ret

StrGet          EndP



;*************************************************************************
;StrPut
;*************************************************************************
;Function: This routine puts a string to the DOS output stream.
;
;Entry: DS:SI holds offset to output buffer
;
;Exit:
;
;Modifies: Registers left unchanged on exit.
;
;Notes:
;*************************************************************************
StrPut          Proc    Far

                        ;Save registers
                        pusha

                        ;Write the string pointed by SI on the screen
                        xor             ax, ax          ;Initialize AX to zero
                        mov             ah, DOSWrite    ;Set AH to DOS Write Function code
                        mov             bx, StdOutput   ;Set BX to Standard Output handle
```

```
                        call    StrLen                  ;Set up CX to number of chars in
string
                        mov             dx, si          ;Set up DX to point to first char
                        int             21h

                        ;Restore registers
                        popa

                        ret

StrPut          EndP



;*************************************************************************
;StrUpr
;*************************************************************************
;Function: This routine converts a string to upper case
;
;Entry: DS:DI holds offset to string buffer
;
;Exit: String changed to upper case
;
;Modifies: Registers left unchanged
;
;Notes: Only characters in the range 'a' to 'z' are altered to avoid affecting
;               non-alphabet ascii charaters
;*************************************************************************
StrUpr          Proc    Far

                        ;Save registers
                        pusha

UprStart:
                        cmp             BYTE PTR[di], 0         ;if ([DI] == 0 )
                        je              UprEnd                  ;then return
                        cmp             BYTE PTR[di], 'a'       ;if ([DI] < a )
                        jb              NotChar                 ;then move to next char
                        cmp             BYTE PTR[di], 'z'       ;if ([DI] > z )
                        ja              NotChar                 ;then move to next char
                        and             BYTE PTR[di], UPPER_MASK ;Set 5th bit to zero

NotChar:
                        inc             di                      ;Increment DI
                        jmp             UprStart                ;Jump to beginning of loop

UprEnd:
                        ;Restore registers
                        popa

                        ret

StrUpr          EndP


;*************************************************************************
;StrCmp
;*************************************************************************
;Function: This routine compares a string with another
;
;Entry: DS:SI holds offset to string 1 buffer
;               ES:DI holds offset to string 2 buffer
;
;Exit:  Flags register set
;
;Modifies: All other registers left unchanged
;
;Notes:
;*************************************************************************
StrCmp          Proc    Far
```

```
                        ;Save registers
                        push    si
                        push    di
                        push    cx

                        call    StrLen                  ;Calculate string 1 length
                        inc             cx              ;Add NULL character
                        cld                             ;clear direction flag
                        cli                             ;Disable interrupts (bug in 8086)
                        repe    cmpsb                   ;Perform comparison
                        sti                             ;Enable interrupts (bug in 8086)

                        ;restore registers
                        pop             cx
                        pop             di
                        pop             si

                        ret

StrCmp          EndP


;*****************************************************************************
;StrCpy
;*****************************************************************************
;Function: This routine copies one string to another.
;
;Entry: DS:SI holds offset to source string buffer
;               ES:DI holds offset to destination string buffer
;
;Exit: copies string contents of DS:SI to ES:DI
;
;Modifies: Registers left unchanged
;
;Notes: The algorithm to determine if an incremental or decremental copy is
;               required is
;               if( Source Offset < Destination Offset )
;               {
;                       if( Destination Offset < (Source Offset + Source Length) )
;                       {
;                               Use decrement method
;                       }
;               }
;               else
;               {
;                       Use increment method
;               }
;
;               Also note that the comparisons require multi-word arthmetic operations
;               (DS:SI and ES:DI), and in these calculation the Zero flag can be
;               misleading and hence the algorithm uses only the Carry flag in making
;               its determinations.
;*****************************************************************************
StrCpy          Proc    Far

                        ;Save registers
                        pusha

                        ;Calculate the 20-bit real address of the source string
                        ;and store it in DX:AX
                        mov             ax, ds
                        shl             ax, 4
                        mov             dx, ds
                        shr             dx, 12
                        mov             cx, si
                        add             ax, cx
                        adc             dx, 0

                        ;Calculate the 20-bit real address of the destination string
                        ;and store it in BP:BX
```

```
                        mov             bx, es
                        shl             bx, 4
                        mov             bp, es
                        shr             bp, 12
                        mov             cx, di
                        add             bx, cx
                        adc             bp, 0

                        ;Save DX:AX
                        push    ax
                        push    dx

                        ;Perform multi-word comparison
                        ;if(Src offset (DX:AX) >= Dest offset (BP:BX))
                        ;Note: Do not rely on Zero flag
                        ;Ust Carry flag only
                        sub             ax, bx
                        sbb             dx, bp

                        ;Restore DX:AX
                        pop             dx
                        pop             ax
                        jae             IncStyle

                        ;Calculate Source offset + Source length
                        call    StrLen
                        inc             cx
                        add             ax, cx
                        adc             dx, 0
                        add             ax, si
                        adc             dx, 0

                        ;Perform multi-word comparison
                        ;if (Dest offset < (Src offset + Src Length))
                        ;Note: Do Not rely on Zero flag
                        ;Use Carry flag only
                        sub             bx, ax
                        sbb             bp, dx
                        jb              DecStyle

IncStyle:
                        call    StrLen
                        inc             cx
                        cld
                        cli
                        rep             movsb
                        sti
                        jmp             CopyDone

DecStyle:
                        call    StrLen
                        mov             bx, cx
                        inc             cx
                        lea             si, [si + bx]
                        lea             di, [di + bx]
                        std
                        cli
                        rep             movsb
                        sti

CopyDone:
                        popa

                        ret

StrCpy          EndP


CodeSg          EndS
End             Main
```

7.0 – Conclusion

Every programming language has a set of constructs and a set of instructions. In the x86 assembler programming language we studied the following five constructs:

- Sequential processing
- Jumps
- Loops
- Procedure calls
- Addressing Modes

Along the way, we also came across several x86 instructions that allow us to do bit manipulations, arithmetic operations and string operations.

The constructs of a programming language are the equivalent of grammar, and instructions are the equivalent of vocabulary, in a spoken language. As with spoken languages, fluency in constructs and instructions come with continued use of the language. Hence it is recommended that the student practice the concepts discussed in this set of notes by writing many more applications.