

A tutorial on
Assembly Language Programming

A beginner's guide to assembly language programming.

Author: Archis Gore

Introduction

This tutorial has been targeted towards an audience having substantial knowledge about programming a microcomputer. It is assumed that you are familiar with the 'C' language and hence a lot of concepts have been explained using terminology from C. Since C is very close to the machine level, it is very easy for someone familiar with C to catch the basics of assembly programming quickly. Please read the complete tutorial as later sections rely on the earlier sections for references.

This tutorial is not meant to be an electronics tutorial or a complete reference on assembly language. It assumes that you have read your references and texts and have done your homework before turning to actually writing assembly programs on a machine. It doesn't explain what op-codes are and how machine instructions are executed, etc. However it is crucial that you know these things before starting up your assembler unless you are trying to simulate the effects of a virus on the computer.

The assembly language instructions used in this tutorial are guaranteed to run on any Pentium class or higher processor. More than 95% of the instructions should work on any machine right down to the 8086. It would be a good idea to have with you a complete reference of the instruction set of the Pentium while reading this tutorial. A good place to find one would be any reference book preferably by 'Peter Abel'. The best place, however, is to get a printout of the complete instruction set from the documentation that accompanies NASM (explained later). It is also a good idea to know about the various registers and their use before attempting to begin programming in assembly.

This tutorial is intended to be an online tutorial meaning that the best use of this tutorial can be made if you sit in front of a machine and follow the instructions given in this tutorial. Before proceeding further, make sure you have a machine which will let you open up a DOS window (though our assembler supports anything from WinXP to Linux and HP-UX, you wont find them as forgiving as DOS). If you are using windows 9x, it is a good combination. You will need to make sure that your text editor works properly (I prefer the plain 'edit' provided by dos; though some prefer notepad). The main thing to have installed is an assembler. The recommended assembler is NASM (Netwide Assembler). It is a full-fledged assembler with a powerful macro preprocessor and supports advanced instruction sets right up to the P4, AMD Athlon, etc. This tutorial will try to explain the peculiarities of NASM as far as possible. The best way to learn assembly language programming is to practice, practice and yet practice.

The tutorial shall also explain how to choose alternatives from various instructions to optimize your programs for the Pentium architecture. It would be useful if you have read about the execution pipelines and can make your programs faster by orders of magnitude.

A crash-course on programming

Though you may have some programming experience, this section is designed to reorient you to some basic programming concepts as they will be very useful later. First and foremost, assembly language programming is no joke. It is not something you can read out of a thick book “Assembly for dummies in 1 hour” and go on impressing people about. Adding two numbers is not assembly language programming. The processor does that for you.

Next, assembly programming does not involve the “u and v pipelines” or the “translation lookaside buffers”. Though these things exist, you are never concerned with them. In fact, unless you are etching out the processor on the silicon substrate, writing the internal microcode at Intel or developing the preemptive multitasking code for Microsoft, you are rarely *allowed* to get concerned with them. So please stay away from complicated terminology which makes no sense to you as a programmer. Think about one instruction at a time and its effect on the processor.

Third, we take a detour to the complex terminology part for a while since this will help in simplifying the instructions later on. Contrary to popular belief, a complex instruction name does not indicate a complicated operation. In fact, a simple instruction is what makes life hell for most assembly programmers (or any programmers for that matter). To understand this concept, it is enough to know that a complicated instruction performs the complicated job for you which is why using it is simple. The simple instruction is the one which only performs your job partially and you have to “think” to get your work done. Consider the example of computing the square root of 2. The seemingly complicated instruction FSQRT is actually very simple since it gives us the square root of a number immediately. However, try computing the square root of any number accurately and in 10 clock cycles using any language of your choice using only addition, subtraction, multiplication and division which are seemingly simple operations.

And the final point is that assembly language is not magic. It does not make you a God of anything. It will neither do stuff for you that cannot be otherwise done, nor will it make you do otherwise simple stuff in a more complicated manner. Assembly language is a language like any other language. It will permit you whatever any other language permits you and will prevent you from doing whatever the processor cannot do. So do not expect to add two numbers in a very complex manner just because you are using assembly language. Also, do not expect that you can compete with Microsoft just because you know assembly.

This said, it is time to begin our adventures into assembly programming. Beware that like any other adventures, expect your machine to crash at least 10 times before you first program works. But this is what makes you the best programmer.

Preparing to program

This section explains what an assembly language program looks like and how to recognize one. Before writing any program, it is necessary that you know what an assembly program looks like, how to identify it, assemble it (assembly equivalent of compiling it), run it, etc. As mentioned previously, the assembler that we intend to use is NASM available freely with source-code at <http://nasm.sourceforge.net>. For our purposes, please unzip the nasm.exe (or nasmw.exe for a dos box in windows) to any directory. Create a batch file asm.bat in the same directory with the following line in it:
nasm(w).exe %1.asm -o %1.com

This will speed up our typing a little bit. It would be a good idea to know that while in C you have been preparing exe files, in assembly, we shall be making com files. The first reason is that exe files have a much more complex internal structure than assembly programs and have a lot of relocation stuff that has to be managed. This puts a lot of load on the programmer to write proper assembly code. It also creates the need to have an intermediate linker to convert the .obj files into .exe files. The second reason is that memory management for com files is easier. You have the whole memory space to yourself without worrying about where other programs are located. The OS virtualizes all the memory you need. And the OS also provides you with a stack which your program does not have to manage. Although NASM supports more complex output formats, it will be left to you to satisfy your curiosity by finding out sources of information for its use.

If you plan to store your programs in any directory other than the one in which nasm(w).exe is located, please add this line at the end of the file c:\autoexec.bat and reboot your machine:

```
PATH=%PATH%;<nasmdirectory>;
```

(<nasmdirectory> is the directory path in which the nasm(w).exe is located)

Whenever I use the phrase “assembling a program”, I simply expect you to type the following command:

```
asm filename
```

Please note that “filename” does not include the extension (remember we added the extension in our batch file above). After this, the assembler is fired up. It processes your program and provides output messages and warnings. This part is too extensive to be covered in the tutorial. This is where your instruction set reference comes in. Please read the instruction at the line having the error and check that the operands and other stuff on that line are valid. If the assembly went well, expect to have the file “filename.com” in your current directory. Simply execute it to run your program.

Please make sure you have done your homework on “little-endian” and “big-endian” formats. The little endian format is used by the Pentium to store bytes in memory (also known as inverted or swapped format) in which the LSB is stored at the lower memory address. This will be an important issue to be considered. Please access variables in terms of the bytes declared for them. Otherwise the program may be seriously messed

up. So if a variable is allocated one byte, please do not try to access two bytes from it as the machine will intrude on the space of some other variable.

One final thing is the issue of what mode we work in. As per Intel documentation, the Pentium can work in real mode or protected mode. Please note that this is independent of the number of bits we use in our programming. For example, both 16 bit and 32 bit programming can be done in real mode and in protected mode. For the course of this tutorial we shall be working in real mode with 16 bit addressing. NASM supports all types of modes and you can find them by reading its documentation.

Please note here that 16 bit mode does not imply the use of only 16-bit registers. It only implies 16-bit segment-offset type addressing in which a 20-bit address is used for memory access. All 32bit registers are supported in 16bit mode. So in our programs `eax`, `ebx`, etc. will work perfectly fine. All 32bit computations are supported. The FPU will support upto 80bit floating point numbers.

We shall not be working in protected mode since DOS works only in real mode and working in Windows or Linux is too complex a task in writing only the startup code for our programs. Writing system calls to the windows API (Application Programming Interface) is something even VC++ programmers avoid to a large extent. They are difficult to trace, very primitive and very unforgiving towards mistakes. Many C programmers used to programming in Turbo C find working with gcc on Linux a nightmare. Imagine the chaos created while writing assembly language programs without the standard ANSI C libraries.

Transition from C

In this section I have covered the differences assembly language programming has with higher level languages like 'C'. Though I have come up with ways to get around them, it doesn't mean that they don't exist.

The first and foremost difference and also the most annoying difference is the absence of the printf and scanf functions. Assembly language provides you directly what the processor provides you. The processor does not provide such functions like scanf and printf. So how the hell do I communicate with the user? The answer is simple, make system calls (or interrupt calls on DOS). Please be aware that though there are powerful services provided by dos, internally everything works at a character by character level. Even these services are not as powerful as printf or scanf. Their maximum potential is to display a static string from memory to the screen. So how do we display all those complex things like "%d %c", etc.? The only solution is the code them in explicitly into your program. To make this coding easier, we use a concept known as macros. These macros are similar to macros used in C. We simply put code used again and again into a specific place and ask the assembler to put it in wherever we want.

Another option, if you are up to it, is to call the printf function from a C library. This will require you to know the C calling convention and dealing with the call stack manually and explicitly. How to do it is out of the scope of this tutorial. We shall rely on macros throughout our discussion.

Another difference from C is that whenever you want to reference variables, you always have to refer to them by their addresses and not their value, since NASM stores only addresses of variables (those familiar with MASN and TASM will find this a bit annoying). This will become apparent when we write our first "Hello World" program. Also, you must put back any changed variables from the registers into memory before overwriting the register contents. For C programmers, this means that NASM only knows pointers. Furthermore, these pointers are of type void. So you have to typecast the pointers and then de-reference them before accessing the value of the pointed data.

Most macros discussed in this tutorial are available at my website <http://www.geocities.com/archisgore/asm/>. It not updated yet, they should be available soon. Else please mail me at archisgore@yahoo.com. How to use the macros in your programs is detailed below. Please refer to the file macros.txt for a complete listing explanation of each available macro.

Macros

Before proceeding further, let's address the one question in all our minds; where the heck did all those macros come from? If you haven't already guessed, they came from the two include files "header.inc" and "footer.inc" (described later in the program section). You may open up these files to see how a macro is declared. The NASM documentation provides all the details to declare macros and use macro-local labels, etc. Hence, this will not be covered in this tutorial.

A quick macro reference is given below.

A macro is declared by saying "%macro macroname <no of parameters>" on a single line. A macro ends by saying "%endmacro". Anything between these two lines is part of that macro. You may refer to each parameter in a macro by using "%1 %2 %3", etc. just like writing batch files.

Whenever you call a macro with those parameters, all the contents of the macro are directly placed in that part of your main program with the "%1, %2, etc" replaced with the parameters you provided in their place.

How do my macros work? Well, the file header.inc contains all the macro definitions needed for your program to work. They must be declared before they are used just like your C preprocessor. Then why do I need the file footer.inc? Because, if you went replacing each macro name by the amount of code needed for each macro, you could easily run out of the 64K limit for com programs. Hence, each macro uses supporting subroutines (functions for C programmers) to perform commonly required tasks. Since these subroutines must not interfere with your normal program execution, they are put at the end of the program and hence are contained in the file footer.inc.

Why the extension .inc? Because I liked it. Makes it easier for me to remember that they are include files and not to be compiled as individual programs. You can change it if you want.

Why not call the functions directly from the program? Well, the answer is simple enough. Because a programmer can't be trusted. Functions rely on the parameters passed to them and hence do not perform validations or checking of any kind. Also, functions have a tendency to change values in various registers. The macros make sure that all your registers are safely pushed on the stack, and that the function parameters are set up correctly before calling the function. They also restore your registers to their states before returning. Hence, you can call any macros without having your registers change their values.

An important issue to be addressed here is for input. If the macros do not change register values, then how do I get my input from the user? You take all your input into memory. Whenever you want to input something from the user, pass a memory location

instead of a register (though this is legal, it is pointless). All this is given in the sample programs below. Please read them carefully.

Also remember that though macros help, they do not perform validations or type-checking. If you pass wrong parameters to the macros, they will give you wrong output (the GIGO syndrome: Garbage In Garbage Out).

To sum up this section, please use the macros like any other machine instructions that you use. Only remember that you are using a macro and that it is going to expand into a complex set of instructions while compiling the program. This is especially important during examinations.

Please also note that no macros are provided for double word values (32 bit values). Below is a list of all macros provided in the header.inc and footer.inc files with a description of its syntax, etc. Please also keep it with you while writing your assembly language programs. They help a lot.

Hello World

Looks like all that philosophical discussion at the top is over and we are finally at a stage where we are writing our first program in assembly language. I have given my hello world program below. A detailed explanation of the program will suffice to give a basic idea of the general program structure of NASM.

```
org 100h

segment .data
    msg: db "Hello World$"

segment .text
    dispstr msg
end

%include "footer.inc"
```

The first line looks very familiar to C programmers. It performs exactly like its C counter part. It directly inserts the contents of the file "header.inc" into our program at that location. Similarly, the last line inserts the contents of the file "footer.inc" at the designated position.

The second line says "org 100h". It simply tells the assembler where in memory our program should start. Com programs start at 100h. Sys and device driver files start at location 0h. The 100h is necessary for the program to work, though you may experiment to simulate the effects of a virus on your computer.

The "segment" lines are assembler directives telling the assembler of the beginning of a new segment. The assembler fills in the values of the various base segment registers with the starting addresses of these segments. Segments always start on a paragraph boundary i.e on an address which is a multiple of 16.

The "segment .data" line tells the assembler that this is a data segment and contains static or initialized data. Hence DS points to the starting address of this segment.

The next line is where some actual work gets done, although it is not an executable statement. The 'msg:' part of the line indicates the label of the line (basic fans should find this very interesting). As you may see, the label declaration is identical to that used in C. The next word 'db' (define or declare a byte) tells the assembler to store a series of bytes which are given in quotes later. The '\$' at the end of the string is required by DOS. It is the string termination character required by the string services provided by DOS.

You may also reserve other types such as 'dw' for declaring a word, 'dd' indicates double word, while 'dq' indicates quad word. These will be used when we look at the FPU.

The cumulative effect is that the string “Hello World\$” gets stored in the data segment.

Next comes the “.text” segment. This is equivalent to the “.code” segment but a standard on the newer OSes. This segment is only readable and contains the executable code of your program. This is where the actual assembly instructions you write should appear.

The first line in the code segment is “dispstr msg1”. This is our first encounter with a macro. Dispstr macro displays a ‘\$’ terminated string on the screen. Msg1 is the address of (or pointer to) that string. Please note that all addresses are accessed from the data segment.

The next line “end” is also a macro which calls the DOS program termination service.

And our program ends. Please assemble the program and run it (discussed above) to verify the output.

Basics of NASM

Finally, after a lot of beating around the bush I shall now begin all the details on NASM and assembly language programming. I hope by now you know about macros and other stuff. Here, I shall not focus on writing programs or explaining them but I shall focus only on specific parts of programs that I have directly given

```
%include "header.inc"

org 100h

segment .data
    no1: db 0
    no2: db 0
    no3: db 0
    result: db 0

segment .text
    newline
    inpedcbyte [no1]
    newline
    inpedcbyte [no2]
    newline

    call func_add

    dispbytedec [result]

end

func_add:
    mov al,[no1]
    add al,[no2]
    mov [result],al
ret

%include "footer.inc"
```

The example above details all that is left of out assembly language tutorial. There are only two things to note in the above example that are different from MASM or TASM. If you never used MASM or TASM, you should not have much of a problem with this. This first thing is, in MASM or TASM, the assemblers remember how much space is allotted for which variable. In NASM, there no concept of a variable. NASM understands only labels. Hence, in MASM, to move contents of variable no1 to al, you would say something like *mov al,no1*. However in NASM, you have to give the same instruction as *mov al,[no1]*. This is because NASM knows only the value of no1 and not the value at the memory location of no1. So only no1 will give the address of the variable and not the contents of the variable itself. To access contents of no1, you must use [no1].

In C terminology, all the labels defined in the data segment are pointers to the data. Further, these pointers are void pointers, i.e. they do not know what type of value they point to. The value may be a string, a byte, a word, a double word, etc. The square brackets are the de-reference operator. Hence 'no1' is the address of data contained in the first number while '[no1]' is the actual value of the data. This is something like saying *no1 in C. In fact, if you ever try to access this assembly code from C, you will be writing in exactly this way. Now about the typecasting part. In C, you have to say something like *((char *)c) to tell the compiler that the pointer is to a character. Similarly in NASM you say 'byte [no1]' to tell it that you are pointing to a byte. Similar statements would be 'word [no1]', 'dword [no1]'.

The second difference in NASM is the way functions are declared. As I mentioned earlier, NASM knows only labels or addresses. It doesn't know what those addresses mean. Hence, you will find that in the above code sample, there is no elaborate definition for a function declaration. Only a label is sufficient. A call statement calls the function and a ret instruction returns.

The third difference to note if you have been working with other assemblers, is that NASM does not have any elaborate syntax for indexing. For example, instead of something like bp[si] in other assemblers, NASM uses only [bp+si]. So anything between square brackets is an address. Any arithmetic or indexing to be done should be done within the square brackets only. For example [ebx*2+eax] is allowed in some cases.

A typical program line in NASM has the following format:

Label: Op Code Operands ;comments

Please note that the ':' after the label is optional.

Now let us start analyzing the program given above line by line. I shall start directly in the code segment. The first line contains the macro 'newline'. This macro simply moves us to a new line. The next line is also a macro 'inpdecbyte' which inputs a decimal integer from the user into a byte of memory. Its argument is a one-byte memory location. Note here that I have not used 'no1' since I don't want to change the address of no1 (which I could neither do) but want to fill in the value of byte referenced by [no1] with user input. Why haven't I specified 'byte [no1]'? Just because I'm lazy. Though it is the recommended practice, the macro implicitly understands that the value being referenced is a byte as the macro name itself suggests.

Now let's jump down by two lines. The next line is the first actual core assembly language instruction which will be assembled directly into corresponding machine code. This line "call func_add" makes the processor jump to the label func_add. Then why not just say "jmp func_add"? Because we do not want to jump there permanently. The call instruction will ask the processor to remember the position from where we jumped. Once the function is done doing its job, it can tell the processor to "ret"(urn) and the processor

then takes care of putting us back on track. This saves us a lot of trouble in remembering from where or when we jumped.

Please refer to the discussion above to see how functions are declared. Unlike other assemblers, NASM knows only the address of the function. There is no elaborate format for declaring a function. Actually, you can call any label as if it were a function. This would not give you proper results unless that label has a corresponding ‘ret’ to get out of the function call.

Hence, a function in NASM is simply a self-sufficient “label: .. ret” block of code.

The almost-last line is a macro which tells the processor to display a byte in the form of a decimal integer.

Finally there is the ‘end’ macro just like the “hello world” program. This one, we already know.

A Floating Point example

Let us now write a program to use the FPU (Floating-Point Unit). This is the fun part of assembly programming. The FPU does a lot of things for us very easily. In fact, programming the FPU can be simpler than using C for doing certain jobs.

Again, I am going to provide a sample program and walk you through it instruction by instruction.

```
%include "header.inc"
org 100h
segment .data
    no1: dd 45
    no2: dd 90
    no3: dd 144
    result1: dw 0
    result2: dw 0
    result3: dw 0
    result4: dw 0
segment .text
    finit
    fild dword [no1]
    fsincos
    fistp word [result1]
    newline
    dispworddec [result1]
    fistp word [result2]
    newline
    dispworddec [result2]
    fild dword [no2]
    fcos
    fistp word [result3]
    newline
    dispworddec [result3]
    fild dword [no3]
    fsqrt
    fistp word [result4]
    newline
    dispworddec [result4]
    end
%include "footer.inc"
```

I have never personally tested the program above. I just wrote it, so you may have to make some slight modifications to it before it will work fine. I wont be explaining how

the FPU works, etc. Please read up on stack based CPU's. The FPU is an accumulator based processor i.e. it reads its operands from its stack and then pushes the result back in on the top of the stack. FPU registers have names st0, st1, ... and so on. All registers are 80 bit floating point registers. They form a stack internally with st0 at the top.

I believe that the first instruction needs no introduction. 'finit' simply initializes the floating point unit.

The next instruction loads an integer from a 32bit double word given by location [no1] into the floating point stack.

Next, the instruction 'fsincos' computes both the sine and cosine of the value in st0 (the top of the FPU stack) and pushes the values back on to the FPU stack. Hence, st1 contains the sine and st0 contains the cosine of the value.

The fourth instruction 'fistp' loads a word of memory with the integer part of the value at the top of the FPU stack. It then pops that value of the top of the stack, hence the value in st1 now comes in st0.

We get this value again. Notice that I am using only words while retrieving values. The reason is very obvious. I do not have macros to display the contents of a double word.

These popped values are now displayed in decimal format to the user. Please note that I have entered angles in degrees and their sines and cosines will be computed in radians. Just too lazy to get accurate values. There are certain FPU instructions to load certain in-build constants such as PIE into the stack. Find them and use them.

I have chosen the value for the square such that its root will be an integer, so that we wont lose any information in the floating-point to integer truncation.

I guess this program is self obvious once you have become comfortable with assembly language. All references to these instructions can be found in the NASM instruction set reference.