Let's Learn Assembly!

by Narue

Prerequisites:

Since I can't cover every possible combination of assembler and hardware, I'll make a few assumptions. My assumptions are that you're using the 32-bit NASM assembler on a Windows machine sporting an x86 processor. Further, I'll assume that you have GCC installed on your machine as that's what we'll be using to link our object files with the C library. Finally, I'll assume that you have at least a passing familiarity with C so that you can use the C library without too much trouble.

Notes about the Tutorial

This tutorial is neither authoritative nor complete. It is written for a beginner on the assumption that the tutorial will get you started and from there you can find your way to more complete resources. The reason I decided to write this is because all of the beginner material I've found has either been too cryptic to be of any use (written by experts who can't teach), too old to be of any use (DOS really *is* dead), or overly complicating the syntax when assembly should be small and beautiful (Randall Hyde's HLA language).

Here are some handy links that will be of use inconjunction with and after getting your feet wet here. They include the NASM assembler itself and relevant documentation. Further resources can be easily found through Google, and I hesitate to list them here because links are so transient. With any luck, the two links I do give will remain comfortably constant, but if not, Google is your friend.

NASM Assembler
NASM Documentation

Basic Components (.data section)

An assembly source file consists of several parts that are divided into sections. The sections that we will be using are .data, .bss, and .text. The .data section holds your initialized global variables, and you start the .data section with the section keyword followed by the name of the section. It's not required, but square brackets help the section header to stand out.

```
[section .data]

; Initialized variables
```

If you haven't already figured it out, comments start with a semicolon and end at the first newline. To actually declare a variable in the .data section, you first need to give it a name. Names in assembly are nothing more than labels, like case labels or goto labels in C. A label is an identifier followed by a colon.

```
[section .data]

myvar:          ; Declare a variable
```

Naturally, simply giving a variable a name doesn't allocate memory for it or assign it a value. To allocate memory in the .data section, you use the allocation keywords: db, dw, and dd. They allocate one byte, two bytes (one word), and four bytes (one double word), respectively. There are two others that extend beyond four bytes and are typically used with floating-point, dq and dt, but we won't use them in this tutorial. The allocation keyword follows the label for a variable.

```
[section .data]

myvar:          db      ; Initialize the variable
```

Now that we have a size for our memory, we can initialize it to something. Let's make a string. Strings in assembly can be delimited using either single or double quotes. My preference is single quotes unless the string contains a single quote, in which case I use double quotes. NASM also allows us to follow the string with a comma separated list of characters or whatnot that add to the string. This can be used to create non-printing characters like a newline (ASCII code 10) or a null character for C-style string handling. Initialization constants are so similar to C that you shouldn't have any issues.

```
[section .data]

myvar:          db      'Hello, world!',10,0    ; C-style string
```

Basic Components (.bss section)

Following the .data section is the .bss section. .bss is just a cryptic name for uninitialized data, so don't worry. It's really no harder than the .data section. The allocation keywords for the .bss section are resb, resw, and resd (also with resq and rest that we won't use). Like the .data section, variables are given a name using a label, but because there is no initialization, you now need to explicitly tell NASM how much of something you want.

```
[section .bss]
```

```
myvar:      resb    64      ; 64 bytes
myint:      resd    1       ; 1 dword
mywords:    resw    5       ; 5 words
```

Notice that the number is scaled to the size you specify, so mywords is actually 10 bytes because a word is 2 bytes, and 5 x 2 = 10. :-) This is especially important because you'll be used to scaling the stack pointer later on, and that uses a strict byte count. If you accidentally give a byte count, you could be wasting a lot of space, so be careful.

Basic Components (.text section)

The next section is the .text section. This is where all of your actual code will reside, and where it really starts to get interesting. Because we'll be interfacing with C to handle a lot of functionality, such as I/O, we need to follow GCC's rules for program startup and shutdown. That basically means a global _main function that returns an integer. NASM lets you specify a global function with the global keyword, then the function itself is just a label.

```
[section .text]

        global _main
_main:
        mov     eax,0
        ret
```

Whoa, wait up a second! What's with the stuff after the label? That's the second part of the requirement, where _main returns an integer. The return value is stored in a register called eax, and control is returned from a subroutine using the ret instruction. The mov instruction is basically assignment, where the second operand is assigned to the first operand. So this code assigns 0 to eax, and returns eax to the calling process. This program is directly equivalent to the following C program.

```
int main ( void )
{
  return 0;
}
```

What if we want to print something, like the hello world program? Just like the global keyword, NASM has an extern keyword that lets you tell the assembler that you're using a subroutine defined elsewhere. Then the subroutines can be called with the call instruction. Here's the hello world program in NASM assembly.

```
[section .data]

hello:  db      'Hello, world!',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    hello
        call    _printf
        add     esp,4

        mov     eax,0
        ret
```

Okay, that's not so simple. Well, you shouldn't expect it to be since assembly is a 1:1 correspondence to the machine instructions of the processor. All of the stuff that higher level languages hide from you, like pushing arguments and fixing stack pointers, has to be done manually. In this case, you need to push the address of the string onto the stack, which is how assembly passes arguments to subroutines, call _printf, and then remove the address of the string from the stack.

The key element here is that the stack remains pristine. Anything you push onto the stack needs to be removed either by using the pop instruction or by adding to the stack pointer, esp. We'll cover stack manipulation shortly.

To run this program, you first need to assemble it into an object file that GCC understands. Then you run GCC on the object file to link with the C library. To assemble the file to a proper object file, you can use the –f switch to nasmw.exe using the win32 option.

```
C:\>nasmw –f win32 prog.asm –o prog.obj
```

This gives you the prog.obj file in the root directory. The –o switch is identical to GCC's –o switch and it lets you specify the name of the object file. Now you can call GCC with the newly created object file and make an executable file with your chosen name.

```
C:\>gcc prog.obj –o prog.exe
```

That's all there is to it. Call prog.exe and watch your masterpiece run. Congratulations, you've made the first step toward being an assembly programmer!

C:\>prog.exe
Hello, world!
C:\>

Registers

The real work of moving stuff around is done with registers. Registers are really really really fast, really really tiny blocks of memory that sit right on top of the CPU, so access is blazing fast. Any data movement will involve a register at some point, if only for the speed benefits. The processor also uses instructions that work on registers because it's faster. Each register has (or had) a well defined job, and it's best to use them for those jobs because the instructions are optimized for it, and it helps make your code self-documenting. There are eight general purpose registers that we'll look at, each named with the acronym suggesting their jobs:

eax - Extended Accumulator Register
ebx - Extended Base Register
ecx - Extended Counter Register
edx - Extended Data Register
esi - Extended Source Index
edi - Extended Destination Index
ebp - Extended Base Pointer
esp - Extended Stack Pointer

The extended part means that the register is 32-bit. Remove the e and you have all eight of the 16-bit general purpose registers. The four registers break down into a union of smaller registers, where eax breaks down into ax, which is the lower two bytes of eax. Correspondingly, ax breaks down into ah and al, which are the upper and lower byte, respectively. This applies for eax, ebx, ecx, and edx.

eax is the accumulator simulator for the x86 processor. ;-) Several instructions have optimized opcodes for operations that assume eax. These are add, adc, and, cmp, or, sbb, sub, test, and xor. You can use these to your advantage. Also be aware that some operations require you to use eax in such a way, which helps explain the usage of multiplication, division, and sign extension. Movement to eax is highly optimized, so if you perform as much work with it as possible, you'll probably have faster code.

ebx is the only register without a well defined purpose. Feel free to use it for extra storage space, but keep in mind that it's an untouchable register that needs to be restored to its original value when you're done with it.

ecx is a loop counter, plain and simple. Prefer ecx when working with loops, but be aware that the native direction that ecx moves is down. That is, it's decremented

rather than incremented. However, changing the direction of an incrementing loop is a simple exercise most of the time.

edx is sort of an extension to eax. Extended size items can be stored by overflowing eax into edx. That's why some instruction documents talk about [edx:eax]. You can think of [edx:eax] as a 64-bit pseudo-register.

esi and edi are the read and write registers for string operations, and several instructions use them. Generally, if you don't use any instructions that expect edi or edi, you can use the two registers for string traversal stuff.

esp is the top-of-stack pointer. ebp is an offset onto the stack that refers to either arguments or local variables in a subroutine. esp shouldn't be used for anything but its designed purpose, but ebp is designed to help you avoid working too much with esp.

Unless stated otherwise, any of the general purpose registers can be used for other things if the need arises, but be aware that instructions are designed and optimized for the registers' jobs. Not using them as they were designed can result in awkward or confusing code.

Because we're interfacing with C, we need to remember that C expects any subroutine to leave certain registers in the same state that it got them. You can use them internally, but you need to restore their values before you're done. Likewise, any C functions that you call will follow this rule as well. The untouchable registers that are relevant for now are ebx, esi, edi, and ebp.

You'll notice that whenever we use one of the untouchable registers, like ebp, we'll make sure to push it onto the stack before changing its value, and pop it off when we're done. Likewise, any register that isn't untouchable (ie. ecx), we'll push onto the stack before calling a C function and then pop it off when the function returns, because we don't want to lose the value and the function isn't required to restore it after using the register internally. This stack magic is required learning for any assembly programmer. :-)

Memory Addressing

NASM is very consistent in how it addresses memory. For any name, *name* is the address of the memory and *[name]* is the contents of the memory. Square brackets around a name are like the indirection operator in C. You can think of every label as a pointer, if you want. Let's see how it works with a "simple" program.

```
[section .data]

mystr:  db      'ABCD',10,0
fmt1:   db      'eax   = %d',10,0
fmt2:   db      '[eax] = %d',0
```

```
[section .text]

        global _main
        extern _printf
_main:
        ; Print the string
        push    mystr
        call    _printf
        add     esp,4

        ; Print the address of the string
        push    mystr
        push    fmt1
        call    _printf
        add     esp,8

        movzx  eax,byte [mystr]

        ; Print the first character
        push    eax
        push    fmt2
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

Don't run away screaming, it's not that bad. Remember that each time you call a subroutine, you first push the arguments to that function in reverse order (we're working with a stack, remember, so the first to pop off of the stack is the first argument). Then when the subroutine returns, you have to fix the stack pointer so that it points to the same place it did before you did all of the pushing. Multiply 4 times the number of arguments, and, remembering that there needs to be zero impact on esp when you're done with it, you can easily see why I added 4 and 8 to esp after the three printf calls. That's the bulk of the program, so let's remove those parts and get right to the meat.

```
mystr:  db      'ABCD',10,0

; Print the string
push    mystr

; Print the address of the string
push    mystr
```

```
movzx  eax,byte [mystr]

; Print the first character
push    eax
```

Printing the string is obvious enough; we push the address (a pointer to) the string, which is what printf expects as its first argument. Then we print the address of the string, remembering that an unadorned name for any variable is the address of that variable. So in C, mystr would be equivalent to &mystr[0]. Then comes the wacky stuff. As you'll learn shortly, the stack only allows dwords. If you just go ahead and push *dword [mystr]*, you'll get a big number because the other three bytes of the dword aren't empty. Not quite what we wanted.

So what we need to do is make sure that only the lowest byte of the dword has the value of the character, and the rest of the bytes are all zero. This is done with the movzx instruction. movzx means mov with zero extension, which means it copies a byte into the low order byte of the dword and copies zeros into the rest of the dword. There's also a movsx that fills the rest of the dword with the sign of the character, so you can correctly copy a negative value. After the movzx, the eax register looks like this, and that's the value we want.

[0][0][0][65]

Now, it's not always as simple as *name* is the address and *[name]* is the contents. Well, it is, but it doesn't always seem that way. You can also use an offset into the contents of the address. For example, *movzx eax,byte [mystr + 3]* would give us the value of 'D' instead of 'A' because we're offsetting the address by 3 bytes. These addresses can be complicated, but most often you'll find yourself using the *[base + offset]* or *[base + multiplier * offset]* forms. The first is good for a straight byte count offset and the second is good for a number of items of size N offset.

Finally, sometimes you need the address of an offset. For example, let's say that instead of simply printing a single character, we want to print a slice of the string, starting at 'C'. Okay, what do we push? If you said *dword [mystr + 2]*, you get a cookie and then a slap on the wrist. :-) *[mystr + 2]* is the character 'C' in the string, not the address of the character 'C' in the string. That means that printf will be looking at address 67, which is not quite what we wanted.

What we need is something like the address-of operator in C, where by applying the operator, you get the address of some piece of data. Fortunately, assembly offers you the lea instruction. lea stands for load effective address, and it's roughly equivalent to the address-of ('&') operator in C. The syntax is much like mov, where the right operand is a data reference and the left operand is a register to store the address of the data. Here's how lea is used to print the slice.

```
[section .data]

mystr:  db      'ABCD',10,0

[section .text]

        global _main
        extern _printf
_main:
        ; Print the string
        push    mystr
        call    _printf
        add     esp,4

        ; Print a slice
        lea     eax,[mystr + 2]
        push    eax
        call    _printf
        add     esp,4

        mov     eax,0
        ret
```

The Stack

Every program is allocated a stack to use for scratch data, local variables, subroutine parameters, return values, and so on. The stack is your friend, even if it causes you untold frustration. :-) When your program starts, you have one register, called esp (for extended stack pointer) devoted solely to telling you the top of the stack at any given point. Any changes to esp that you make must be reversed or you'll quickly feel the pain of an inaccurate stack pointer. Here are the rules:

1) Anything you push onto the stack gets popped when you're done
2) When you add to esp, you subtract the same amount when you're done
3) When you subtract from esp, you add the same amount when you're done

The stack pointer must remain pristine or you'll have problems. Consider this: When the program starts, esp points to memory that contains the return address of the program. If you push 0 but forget to pop it, then you've effectively changed the return address to 0, because that's what the program will use. That's nasty stuff, don't do it. ;-)

Single items are added to the stack (by first subtracting 4 from *esp* and then mov'ing the data to *[esp]*). You can do this manually without too much trouble by using the sub and mov instructions.

```
[section .data]

fmt:    db      '%d',10,0

[section .text]

        global _main
        extern _printf
_main:
        sub     esp,4
        mov     dword [esp],123
        sub     esp,4
        mov     dword [esp],fmt
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

That's awkward, and I'm sure you've noticed by now that there's an instruction that does just that. The push instruction takes some form of data as its operand, subtracts 4 from esp, and mov's the data into the contents of the address that esp points to.

```
[section .data]

fmt:    db      '%d',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    123
        push    fmt
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

The push instruction has a corresponding pop instruction that mov's the contents of the address that esp points to its operand, and then adds 4 to esp. Let's say that we want to take the two values that we pushed and save them in two registers for later use.

```
[section .data]
```

```
fmt:    db      '%d',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    123
        push    fmt
        call    _printf
        pop     eax
        pop     edx

        push    edx
        push    eax
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

For subroutine arguments, it's rare to use pop because simply adding the appropriate number of dwords to esp is sufficient unless you want to save the contents of the stack elsewhere. Push and pop are commonly used together to save untouchable registers, or to save registers that will be modified by a subroutine when you want to keep their value.

The stack has two important uses: First, it stores subroutine parameters, as you've seen when calling C functions. Inside of a subroutine, the parameters are accessed by using memory addressing and lea. For example, let's look at a program that prints out the number of command line arguments to main (through argc), and the first of those arguments (through argv).

```
[section .data]

fmt1:   db      'argc    = %d',10,0
fmt2:   db      'argv[0] = %s',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    dword [esp + 4]
        push    fmt1
```

```
        call    _printf
        add     esp,8

        mov     eax,dword [esp + 8]

        push    dword [eax]
        push    fmt2
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

Upon entering a subroutine, the value of *[esp]* is always the return address of the subroutine. So it stands to reason that the first argument is stored at *[esp + 4]*, the second at *[esp + 8]*, and so on. Therefore, *[esp + 4]* is argc, and we can use it directly as that's the data we want. However, *[esp + 8]* is argv, which is defined as a pointer to a pointer, or an array of pointers. If we try to print *[esp + 8]*, we won't get the output that we want. So the previous program performs a double dereference trick by saving the contents of the memory address *[esp + 8]* in a register. That gives us the address of the array. Then the register is dereferenced to give us the address of the first pointer in the array, which is the one we really want.

We can take this trick all of the way down to the bottom, where the we get to a single character in the string. The process is the same down the line, which is good because consistency begets good coding practices. :-)

```
[section .data]

fmt1:   db      'argc      = %d',10,0
fmt2:   db      'argv[0]   = %s',10,0
fmt3:   db      'argv[0][0] = %c',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    dword [esp + 4]
        push    fmt1
        call    _printf
        add     esp,8

        mov     eax,dword [esp + 8]

        push    dword [eax]
```

```
        push    fmt2
        call    _printf
        add     esp,8

        mov     eax,dword [esp + 8]
        mov     edx,dword [eax]

        push    dword [edx]
        push    fmt3
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

Wait, why did we repeat the mov from *[esp + 8]* to eax? Remember that eax is not an untouchable register, and it's very popular, so you can expect any subroutine that you call to trash it and not restore the value. So we need to make sure that eax still has the value we want by recalculating it and storing it back in eax.

The stack is also used for local variables in a subroutine. You can "allocate" local variables by subtracting from esp and then using memory addressing to look at the block you want. However, when you subtract from esp, you need to add the same amount when you're done and want to "free" the local variables.

```
[section .data]

fmt:    db      'local var = %d',10,0

[section .text]

        global _main
        extern _printf
_main:
        sub     esp,4

        mov     dword [esp],123

        push    dword [esp]
        push    fmt
        call    _printf
        add     esp,8

        add     esp,4
        mov     eax,0
        ret
```

There are two schools of thought on how to access local variables. Because it's really best to avoid working directly with esp except when doing the necessary adjustments, ebp is preferred for the actual addressing. The first school of thought is to keep track of esp like we did above, but mov the value of esp to ebp such that ebp refers to the first local variable, and *[ebp + n]* will give you further local variables.

```
[section .data]

fmt:    db      'First  = %d',10,'Second = %d',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    ebp
        sub     esp,8
        mov     ebp,esp

        mov     dword [ebp],123
        mov     dword [ebp + 4],456

        push    dword [ebp + 4]
        push    dword [ebp]
        push    fmt
        call    _printf
        add     esp,12

        add     esp,8
        pop     ebp
        mov     eax,0
        ret
```

ebp is an untouchable register, so we save it before mov'ing the value of esp into ebp. Subtracting esp by 8 gives us two dwords to work with as local variables, and then the starting address of those two dwords on the stack is assigned to ebp. Now, no matter how much esp changes through the course of the program, the two local variables will always be at *[ebp]* and *[ebp + 4]*. At the end, esp is fixed by adding the same amount, then ebp is restored by popping the saved value from the stack into ebp. Notice that everything we did at the start is undone in reverse order, that's how stacks work.

The second school of thought on local variables is the one supported by the assembly language itself. This does the reverse, first saving the current value of esp, and then subtracting esp to allocate local variables. With this method, the local variables are at a negative offset from ebp rather than a positive offset.

```
[section .data]

fmt:    db      'First  = %d',10,'Second = %d',10,0

[section .text]

        global _main
        extern _printf
_main:
        push    ebp
        mov     ebp,esp
        sub     esp,8

        mov     dword [ebp - 4],123
        mov     dword [ebp - 8],456

        push    dword [ebp - 8]
        push    dword [ebp - 4]
        push    fmt
        call    _printf
        add     esp,12

        add     esp,8
        pop     ebp
        mov     eax,0
        ret
```

Okay, that looks more awkward and confusing that the first method, so why use it? Well, because there are instructions that make it easier for you. :-) The enter and leave instructions handle the saving and assignment of ebp, and the adjustment of esp. It's slower than the manual way, but much cleaner.

```
[section .data]

fmt:    db      'First  = %d',10,'Second = %d',10,0

[section .text]

        global _main
        extern _printf
_main:
        enter   8,0

        mov     dword [ebp - 4],123
        mov     dword [ebp - 8],456
```

```
        push    dword [ebp - 8]
        push    dword [ebp - 4]
        push    fmt
        call    _printf
        add     esp,12

        leave
        mov     eax,0
        ret
```

The enter instruction takes two operands: The number of bytes to subtract from esp, and the nesting level for nested subroutines. For us, we'll keep the nesting level at 0 to keep things simple, so only the first operand really matters. The leave instruction has no operands; it just does the right thing. :-)

Arithmetic

Assembly, of course, supports all of the arithmetic operations that higher level functions support. In fact, some of them are easier in assembly than in languages like C. Let's start with addition and subtraction since we've already used them. The add and sub instructions take two operands: The first operand is the data to add to or subtract from, and the second operand is the amount to add or subtract.

Multiplication is less intuitive, because it operates implicitly on eax, with a register as the operand that specifies the amount to multiply by. So for any multiplication, you need to mov the original value to eax, then the multiplier to another register of your choice, and then call either mul (for unsigned multiplication) or imul (for signed multiplication). The result is stored in eax, or if there's overflow, [edx:eax].

```
[section .data]

fmt:    db      'eax = %d',10,0

[section .text]

        global _main
        extern _printf
_main:
        mov     eax,2

        push    eax
        push    fmt
        call    _printf
        add     esp,8
```

```
        mov     eax,2
        mov     edx,2
        mul     edx

        push    eax
        push    fmt
        call    _printf
        add     esp,8

        mov     eax,0
        ret
```

Division follows similar rules as multiplication (for both div and idiv), with the added problem of division by zero. The quotient is stored in eax and the remainder in edx, so there's no remainder instruction in assembly, since the remainder just falls out of division. :-)

```
[section .data]

fmt1:   db      'eax = %d',10,10,0
fmt2:   db      'eax = %d',10,'edx = %d',10,0

[section .text]

        global _main
        extern _printf
_main:
        mov     eax,13

        push    eax
        push    fmt1
        call    _printf
        add     esp,8

        mov     eax,13
        cdq
        mov     ebx,2
        div     ebx

        push    edx
        push    eax
        push    fmt2
        call    _printf
        add     esp,12
```

```
mov     eax,0
ret
```

Wait a second, what's that cdq instruction? Well, div doesn't just work on eax, it works on [eax:edx], which means that if the dividend is small enough to fit just in eax, edx needs to be zero. It also means that we can't use edx as the divisor. We could simply mov 0 to edx and call it good, but it's easier to sign extend eax into [eax:edx] with the cdq instruction. It's basically the same as movsx except without the mov, and you're forced to use eax. :-)

Arithmetic negation can be performed with the neg instruction. It simply gives you the two's complement negation (invert all bits then add one) of some data. There's also a not instruction for the one's complement negation (just invert all of the bits). neg and not take one operand that they work on.

If you find yourself saying *add x,1* or *sub x,1* often, you've independently discovered the need for the inc and dec instructions that add one to their operand. These instructions are likely to be optimized for adding 1, so you should favor them over add and sub.

Loops, Conditionals, and Jumping

Those coming from a higher level language are usually shocked to learn that assembly does not have any advanced branch control features. The only thing you can do to control the flow of execution in an assembly program is jump to a label, similar to goto in C. The instructions to perform a jump are rather extensive, ranging from the unconditional jmp instruction to a number of conditional instructions that take their que from a mysterious flags register that we won't look at in this tutorial. The jump instructions we'll use are as follows.

jmp – Unconditional jump

jz – Jump if zero
jnz – Jump if not zero

je – Jump if equal
jne – Jump if not equal

jl – Jump if less
jle – Jump if less than or equal

jg – Jump if greater
jge – Jump if greater than or equal

The jl and jg instructions, and friends, work on signed values. The corresponding unsigned instructions use the same variations except with ‘a’ for ‘above’ instead of ‘g’ for ‘greater’, and ‘b’ for ‘below’ instead of ‘l’ for ‘less’. For now, we’ll assume that the magic comes from the cmp instruction, which compares two values and places the result somewhere that the conditional jump instructions can find it. The details don’t really matter at this point.

A simple two way conditional test (aka. if..else) can be simulated using labels and conditional jumps. Let’s write a quick program to enter an age and determine whether the user is an old fogey or a young’un.

```
[section .data]

prompt:             db      'How old are you? ',0
fmt:        db      '%d',0
oldmsg:             db      "You're old enough for assembly",10,0
youngmsg:   db      "You're too young for assembly",10,0

[section .text]

        global _main
        extern _scanf, _printf
_main:
        enter   4,0

        push    prompt
        call    _printf
        add     esp,4

        lea     eax,[ebp - 4]

        push    eax
        push    fmt
        call    _scanf
        add     esp,8

        cmp     dword [ebp - 4],30
        jge     old

        push    youngmsg
        call    _printf
        add     esp,4

        jmp     end
old:
        push    oldmsg
```

```
        call    _printf
        add     esp,4
end:
        leave
        mov     eax,0
        ret
```

Are these programs beginning to look less overwhelming now? :-) The trick to a conditional test in assembly is to make sure that you jump over the code that isn't used while executing the code that *is*. This often results in several labels and several jumps. For example, an if..else is often set up like the following.

```
        cmp     a,b     ; Is a equal to b?
        jne     else

        ; a is equal to b

        jmp     end
else:
        ; a is not equal to b
end:
```

Notice the careful logic that skips over the blocks that don't match the condition. If a is equal to b, no jump is made initially, but after the processing of the true condition, a jump is made over the false condition. Likewise, if a is not equal to b, a jump is made over the true condition and the false condition is processed. An if without an else can skip the part where the else code is jumped over.

```
        cmp     a,b
        jne     false

        ; a is equal to b
false:
        ; Moving on
```

Loops follow a similar pattern, except instead of a one-time compare and jump downward, you repeat the same block over and over by jumping upward. There are two kinds of loops: Test at the top, and test at the bottom. A test at the top loop is like a while loop or a for loop in C, and a test at the bottom loop is like a do..while loop in C. Here's are the two loop styles.

```
        ; Test at the top loop
loop:
        cmp     [count],10
        jge     done
```

```
        ; Test at the bottom loop
        jmp     cmp
loop:
        ; Code to execute
```

```
        ; Code to execute                   inc     [count]
                                        cmp:
        inc     [count]                     cmp     [count],10
        jmp     loop                        jl      loop
done:
```

The test at the top loop should be avoided in favor of the test at the bottom loop. Why? Because the number of instructions is smaller in the test at the bottom loop. The two are side by side so that you can more easily see that there's only one jump instruction in the test at the bottom loop, but two in the test at the top loop. In code where every little bit counts, the test at the bottom loop will be faster. On the other hand, when learning, you should keep these things in mind, but write whatever you feel comfortable with. :-)

Doing Stuff (printing a string)

Easy I/O is critical when learning a new language. If you can't read data or output data, figuring things out with test code is much more difficult. So the rest of this tutorial will cover basic I/O to help you test your newfound skills with assembly.

Up to now we've been using printf. That's fine, but printf is often overkill if all you need is to print a string with no formatting. An alternative is the puts function, which prints a C-style string and a newline character.

```
[section .data]

msg:    db      'This is a message',0

[section .text]

        global _main
        extern _puts
_main:
        push    msg
        call    _puts
        add     esp,4

        mov     eax,0
        ret
```

Okay, now what if you want to print a string without the newline? The fputs function is not a simple alternative because using the standard streams (stdin, stdout, and stderr) is awkward. C defines those names to be macros, so you can't simply say *extern _stdout* and expect the code to work. You would need to know the internal name and the implementation details. For example, GCC uses _imp__iob as the internal name, but

_imp__iob is an array. So as well as knowing the name, you need to know which elements of the array correspond to which streams, and the size of the FILE structure so that you know how far to offset into the array. FILE streams are to be avoided unless you define them yourself using fopen, or unless you like pain. ;-)

Anyway, an alternative that is supported by both the POSIX standard and the Windows msvcrt.dll is the write function. As a stream it takes a file descriptor (a simple number where 0 is stdin, 1 is stdout, and 2 is stderr), the string, and the number of characters to write. This greatly improves string I/O when you consider the alternatives. The only problem is that write requires the length of the string. For static strings in the .data section, you can use a trick.

The equ mnemonic defines a named constant. So to create a constant called pi, you could say *pi: equ 3.14159* and use pi to mean 3.14159. A nice feature is that you can use an offset of a label to calculate the length of a string. Couple this with equ and you have a nice way to get the length of a static string in the .data section.

```
[section .data]

msg:    db      'This is a message'
len:    equ     $-msg

[section .text]

        global _main
        extern _write
_main:
        push    len
        push    msg
        push    1
        call    _write
        add     esp,12

        mov     eax,0
        ret
```

Strings that are defined in the .bss section are different, because they don't have a set size, only a set capacity. For those strings you need to calculate the length either through something like strlen, a loop of your own, or an input function that returns the number of characters that were actually read.

Doing Stuff (reading a string)

Scanf is typically a poor choice for string input, and fgets isn't a good option due to the awkwardness of using stdout as an external name. Fortunately, the write function

has a corresponding read function that's just as widely available. Conveniently enough, it returns the length of the string it reads, so pairing read and write is a good thing. :-)

```
[section .text]

        global _main
        extern _read, _write
_main:
        enter   36,0
        push    ebx

        lea     ebx,[ebp - 32] ; The string

        push    32
        push    ebx
        push    0
        call    _read
        add     esp,12

        mov     dword [ebp - 36],eax

        push    dword [ebp - 36]
        push    ebx
        push    1
        call    _write
        add     esp,12

        pop     ebx
        leave
        mov     eax,0
        ret
```

Now, at this point we're beginning to feel the pain of pushing arguments and cleaning up the stack pointer for every subroutine call we make. Fortunately, NASM offers macro facilities that can wrap all of this away into a more conventional function call. Unfortunately, it's not a standard macro, so we have to write it ourselves. Fortunately, I've written it for you. :-) What you need to do is create a new file, I called mine 'macros.inc', and paste this code into it.

```
%macro invoke 2-*
        %define _func %1

        %assign __params %0 - 1
        %assign __params __params * 4

        %rep %0 - 1
```

```
                %rotate -1
                push %1
        %endrep

        call _func
        add esp,__params
%endmacro
```

How it works is beyond the scope of this tutorial, as are the macro facilities of NASM. For now, we can take advantage of the macro without knowing how it does what it does. And what it does is lets you use invoke as an instruction, with the first operand being the subroutine to call, followed by the arguments to the subroutine in first to last order. You can include the macro file with the %include directive. So the read/write program above can be rewritten like so.

```
%include 'macros.inc'

[section .text]

        global _main
        extern _read, _write
_main:
        enter   36,0
        push    ebx

        lea     ebx,[ebp - 32] ; The string

        scall   _read,0,ebx,32
        mov     dword [ebp - 36],eax

        scall   _write,1,ebx,dword [ebp - 36]

        pop     ebx
        leave
        mov     eax,0
        ret
```

That's much more pleasant to work with, especially if you're more familiar with the C function call syntax than the reverse pushing of assembly. If you prefer to list the arguments in reverse order, change *%rotate -1* to *%rotate 1* and while the subroutine to call will still be the first operand to scall, any further arguments will be listed from right to left instead of left to right.

Doing Stuff (reading a number)

Numeric input is difficult if you need to perform the conversion from a string to a number on your own. The good news is that the C library supports several functions that help you to either read a number directly (with scanf) or easily convert a string to a number (strtol and friends). Here is a age program rewritten to use strtol instead of scanf.

```
%include 'macros.inc'

[section .data]

prompt:             db      'How old are you? ',0
fmt:        db      '%d',0
oldmsg:             db      "You're old enough for assembly",10,0
youngmsg:   db      "You're too young for assembly",10,0

[section .text]

        global _main
        extern _read, _printf, _strtol
_main:
        enter   32,0

        scall   _printf,prompt
        lea     ebx,[ebp - 32]
        scall   _read,0,ebx,32
        mov     byte [ebx + eax - 1],0

        scall   _strtol,ebx,0,0

        cmp     eax,30
        jge     old

        scall   _printf,youngmsg

        jmp     end
old:
        scall   _printf,oldmsg
end:
        leave
        mov     eax,0
        ret
```

Notice that no error checking has been made with either scanf or strtol. It's easy to check the return value of scanf because it's stored in eax, but strtol gets its error checking from the second argument, a pointer to a pointer. This is where lea really shines, but I'll refrain from showing you how to do it since you have the knowledge and the tools to error check strtol, and it's a good exercise. :-)

The End

That's all folks! By now you should be pleasantly surprised at the ease with which you can now read NASM syntax assembly language. You should also be surprised that you can actually understand what's going on as well, and maybe even be comfortable with writing what, at the beginning of the tutorial, was nothing more than gobbledygook.

Remember that you've only scratched the surface of assembly. There are hundreds of instructions and we only looked at some of the commonly used ones. Further, there are facets to NASM that extend and enhance assembly that are well beyond the scope of this tutorial. We didn't look at anything more than simple command line programs, but assembly can be used for everything from bare metal bios routines to the latest and greatest GUI application. Finally, NASM is only one of many assemblers, all with a different syntax.

I hope you think that assembly isn't as hard as you thought before, and that it might even be fun. It is! You just have to get over that first hurdle, which is what this tutorial was written to help you do. Happy programming! :-)