# Functions in Assembly

By Alexander Wang

42 Silicon Valley™

## A Good Second Program and a Design:

*“We’re not programmed. You have to learn to make your own decisions.”* – Captain Rex, Star Wars: The Clone Wars

If you have programmed in Python or any other language for any length of time then you will learn that the computer is stupid, unbelievably stupid. It needs exact and precise instructions to do anything useful. This next program is going to be simpler than your used to but keep with me:

When you ask somebody to pick the largest number, you can very quickly do it. But our brains are very good at skipping a lot of complex tasks. When your scanning for the largest number, you’re probably keeping track of the largest number you’ve seen so far. While your mind is very good at this, for the computer you must explicitly set up storage for holding the current position in the list and the current maximum number. When reading a piece of paper you know when you run out of numbers, but for a computer it doesn’t know when it runs out.

So let’s do a little bit of planning. Let’s assume that the last element is 0 so we know when it ends. Let’s also assume that the address where the list of numbers starts is **data_items**. We also need to hold the position in the list, a value to hold the current list element we’re looking at, and the current highest value on the list.

Let’s assign each of these things to a register:

- %eax will hold the current element being looked at
- %ebx will hold the current highest element
- %esi will hold the current position in the list.

When we first start the program, we don’t have a point of reference so the very first element will be the biggest element. Also we will set the current position in the list to be zero. Now let’s solve this in the following sequence of steps:

1. Check the current list element (%eax) to see if it’s a zero (our terminating element).
2. If it is zero, exit.
3. Increase the current position (%esi).
4. Load the next value in the list into the current value register (%eax). Question: What addressing is used here?
5. Compare the current value (%eax) with the current highest value (%ebx).
6. If the current value > current highest value, replace highest value with current.
7. Go back to Step 1.

You might notice that there are some “if” statements in the steps above. These are where decisions are being made. Depending on which if-statement is correct, the computer might do a totally different set of instructions. In more complicated programs, the number of jumping around will dramatically increase.

These if-statements are a class of instructions called **flow control** and are implemented by unconditional and conditional jumps. The **conditional jumps** change paths based on a previous comparison or a previous calculation. The unconditional jump will go to the path regardless of what happens. This may seem useless, but the unconditional jump is how you will return to the “original path”.

Another obvious use in this case are loops, if you have programmed in Python or C you’ve probably used them extensively. It is implemented by doing unconditional jumps back to the beginning of the loop. However, you must always have a conditional jump to exit out of the loop or else it will be an infinite loop. Now let’s implement the next program!

## Finding the Maximum Value in a List:

*“Next time, just tell me to jump.”* – Captain Rex to Anakin Skywalker

*"Now where's the fun in that?"* – Anakin Skywalker's reply

Type the following program as max.s:

```
#Purpose: This program finds the max number in a list of items

#VARIABLES:
# %eax current item
# %ebx largest item
# %edi holds the index of the item being examined
#FOLLOWING DATA ITEMS USED:
# data_items: contains the item data, A 0 is used as the last element
#
.section .data
      data_items:
         .long 15, 581, 51, 51, 2, 3, 51, 12, 1555, 4, 0
.section .text
.globl _start
_start:
      movl $0, %edi #move 0 into our index register
      movl data_items(, %edi, 4), %eax #load first byte of data; since
                                    #this is the first item, %eax is the
                                    #largest item found so far.
      movl %eax, %ebx
      start_loop:
          cmpl $0, %eax #check to see if we reached the end
          je exit_loop
          incl %edi #load the next value
          movl data_items(, %edi, 4), %eax
          cmpl %ebx, %eax #compare values
          jle start_loop #current value isn't bigger than our max go back.
          movl %eax, %ebx #move the value as largest
         jmp start_loop #repeat the loop
     exit_loop: #ebx is the status code for the exit call & largest number
         movl $1, %eax #1 is the syscall for exit
         int 0x80

```

Now compile it with the following command:

```
as max.s –o max.o
ld max.o –o max
```

Now run it and check its output:

```
./max
echo $?
```

Your output should be: 1555. Now let's go through this line by line.

As you can see there are actually things in our data section now (Lines 10-12). Let's look at it a bit more carefully. The label name (data_items) is a label that refers to the location that immediately follows it. Then there is a an assembler directive following it: "*.long*" and a bunch of numbers. So, everywhere we use data_items the Assembler will substitute it with the memory address associated with it. There are several other directives other than .long and the main ones are:

- .byte – You should know what this is by now.
- .int – Not regular integers, but these take up 2 bytes and are limited to numbers 0 to 65535.
- .long – The traditional "int" value
- .ascii – enters the literal characters into memory. Examples will be shown later on.

In our example, we reserved 11 longs (or 44 bytes (11 * 4 bytes per long). Data_items here specifically refers to the very first number.

Take note in that I decided that the last number would be 0 as a terminator. Now I could have chosen any number of ways but this was the easiest way. Alternative methods: another label to indicate the end, hard-coded the number in. But regardless of the method chosen, the computer knows nothing about the data being processed.

Now normally, you would have variables and variable names associated. In our case, we have 3 variables:

- A variable for the current maximum number
- A variable for the current location
- A variable for the current value.

The program that was written above is nice enough to where we can fit all of them in our select few registers. In larger programs you will need to put them in memory when you run out of registers to play with, which we will cover later on.

In this program %ebx is the location of the largest item we've found. %edi is the *index* of the current data item we're looking at. Now, what is an index? When we read information from data_items, the first element then the second one then the one after. The data_item number is the *index* of data_items.

Now the first line with actual instructions (16). Since we are looking at the first item, we load the number 0 into %edi.

The next instruction is a bit more difficult, but you absolutely must understand it:

```
movl data_items(, %edi, 4), %eax
```

To understand it you need to know the following:

- data_items is the location number of the first element in the list.
- Each number is 4 bytes long
- %edi is holding 0 at this moment.

So, basically, what this says is start at the beginning of data_items, and take the first item (%edi = 0), and remember that each data is 4 bytes long and move it to %eax. This is *indexed addressing* and the general form is the following:

movl  STARTINGADDRESS(, %REGISTER, WORDSIZE).

Now the line immediately after says that we will move the contents of %eax into %ebx because %eax has the first element of the list (which is our starting point). However, even though it says "move" it actually copies the contents instead.

And now we finally reach the wonderful loop. We've marked the starting point with: start_loop. To reiterate the steps we need in our loop:

- Check to see if current value is zero, if it is exit, if not continue
- Load next value of list.
- Check if current value > biggest value.
- If it is, copy to the location with largest value.
- Go back to the beginning.

Now lets enter our loops for lines 22-23:

The cmpl instruction compares two values. Here we are comparing the 0 and the value inside of register %eax. This comparison instruction also affects another register, %eflags. Line 23 is when a jump may occur to the end_loop label if the two values were the same. However we used *je* here but there are many jump statements you can use. Please see Appendix A for a massive list of them.

In this situation, if %eax and 0 are equal then we will exit out of the loop otherwise we go to the next two lines:

```
incl %edi #load the next value
movl data_items(, %edi, 4), %eax
```

If you remember from our previous discussion, %edi contains the index of the list. *Incl* increments the value of %edi by one. Then the movl is the same as the one we discussed before.

```
cmpl %ebx, %eax #compare values
jle start_loop #current value isn't bigger than our max go back.
```

Here we compare our current value in %eax with our max value in %ebx. If the value in %eax is less than our current max value then we will return to the  top of the start_loop. Otherwise we record the current max value:

```
movl %eax, %ebx #move the value as largest
jmp start_loop #repeat the loop
```

This will repeat until the end when it jumps to end_loop. This part of the program calls the Linux Kernel to exit.

## Addressing Modes:

“*My address is like my shoes. It travels with me*” – Mary Harris Jones

In our previous chapter (chapter 2), we learned about different ways to address memory. This section will go into more depth about it in relationship to how Assembly would do it.

In general, the form for memory address references is the following:

ADDRESS_OR_OFFSET(%BASE_OR_OFFSET, %INDEX, MULTIPLIER)

All of the fields are optional. To calculate the final address it is quite simply the following:

$$Final = ADDRESS + \%BASE + MULTIPLIER * \%INDEX$$

ADDRESS_OR_OFFSET and MULTIPLIER must be constant values, while the other two must be registers. If any piece is left out, then it is assumed to be 0.

Now here are the various types of addressing Assembly to deal with:

- **Direct Addressing Mode**
    - Done only by using ADDRESS_OR_OFFSET part
        - Example: movl ADDRESS, %eax
    - This loads the address into %eax

- **Indexed Addressing Mode**
    - Done using ADDRESS_OR_OFFSET and %INDEX portion.
    - You can use constants for the %INDEX part as well.
        - Example: movl ADDRESS(, %ebx 1), %eax
- **Indirect Addressing Mode**
    - Loads a value from the address indicated **by a register**.
    - For example if %eax held an address and we wanted what was inside of it:
        - movl (%eax), %ebx
- **Base Pointer Addressing Mode**:
    - Similar to indirect addressing but adds a constant value to the address held in a register.
        - Example: movl 4(%eax), %ebx
        - Address at %eax + 4 and value at that point moved to %ebx
- **Immediate Mode**:
    - Very simple; loads a value directly into a register.
        - movl $12, %eax
        - %eax now has the value 12
    - Note the “$” it is crucial. Without it, it is assumed you are using direct addressing mode.
- **Register Addressing Mode:**
    - Simply moves data between two registers.

For our next program we will be dealing with functions (oooh, yay…)

## Functions:

*"But in my opinion, all things in nature occur mathematically"* – Rene Decartes

No, not a mathematical function, although very similar to one. We will do some very basic algebraic review as you'll be implementing this function as an illustration. Yay math isn't totally useless!

$$f(x,y,z) = \begin{cases} x * y \; if \; z > 0 \\ x + y \; if \; z < 0 \\ x - y \; if \; z = 0 \end{cases}$$

Now for those who haven't taken algebra yet. This is a relatively simple function: it takes in 3 parameters (whole numbers) named x, y, z and do different things to them depending on the value z has.

Examples:

$$f(1,2,3) = 1 * 2 \; (because \; z > 0) = 2$$
$$f(1,2,-1) = 1 + 2 \; (because \; z < 0) = 3$$
$$f(1,2,0\,) = 1 - 2 \; (because \; z = 0) = \; -1$$

You will be implementing this function. But before that we will discuss how functions work in Assembly.

## How Functions Work:

Functions are composed of a few different pieces:

Function name:

The name is a symbol that represents the address where the function code's starts. In Assembly, the symbol is defined as a label right before the functions code. This is exactly like the code you have used for jumping around.

Function parameters:

A function parameters are the items that are explicitly given to a function for processing. Using the above example, the parameters would be 3 numbers (x, y, z). Function parameters can be many or even none.

Local Variables:

Data storage that a function uses while processing kind of like a piece of scratch paper. They get a new piece of paper every time the function is called. Local variables are not accessible by functions outside of the function it originated from.

Global Variables:

Data storage that a function used for processing **except** it can be used by other functions outside of where it originated from.

Return Address:

An "invisible parameter" that isn't directly used during the function. It is a parameter that tells the function where to resume execution after the function is completed. This is needed because eventually the program needs to go back to its original path. In Assembly, the *call* instruction handles passing the return address for you, and the *ret* instruction handles using that address to return back to where you were when you called the function first.

Return Value:

The return value is the main method of transferring data back to the main program. Most programming languages allow a single return value (Python being a notable exception).

Now, the way variables, parameters and return values are stored and returned varies from computer to computer and language to languages. This variation is called a *calling convention* because it describes how functions expect to get and receive data when they are called.

For the purpose of these assignments you will be using the "C calling convention" because it is the standard in Linux. Do note however you can do whatever you want in terms of calling conventions just note that it will be painful if you try to interface it with other languages that expect a specific kind of calling convention.

## Assembly Using the C Calling Convention and the Stack:

"*Ah.. more bugs, more kills*" – Delta 07 (Sev), Star Wars: Republic Commando

The following section is critical if you want to do the other projects past this initial one with full understanding of the how and why. There is no reiteration of any information in this section beyond this point. **You have been warned**.

You cannot write any program in Assembly without *completely and fully understand* the **stack** on a CPU. To copy-and-paste our earlier analogy:

*"Think of an empty room. Now fill that empty room with a stack of books. Let's say you want the top most book, how would you grab it? What about one in the middle without creating chaos by ripping it out directly?"*

That is how a stack works on a CPU as well. Your computer's "stack" lives at the very top address of memory. You can push values to the top of the stack through an instruction called **pushl**, which pushes a register or a value to the top of a stack.

And now this is where things get confusing, the "top of the stack" is actually the bottom of the stack and grows downwards as you push things onto it. You can also remove things from the stack using ***popl.*** This removes the top element and places it in the memory address or register you chose.

When we push an element onto the stack, the top of the stack moves to accommodate the new value. We can actually keep pushing things onto the stack until we hit our data or code (in which case undefined behavior happens. Yay!). So how do we know where this "top" is? Through one of our special registers (remember those?), the **stack register, %esp**. The stack register will always contain a pointer to the current top of the stack, wherever it is.

Every time we push something to the stack with **pushl**, the stack register, %esp, gets subtracted by 4 (remember word size and the stack grows downwards?). If we remove something from the stack then 4 gets added to %esp, and the value gets added to whatever it is we want.

Now what if we want the top element without modifying the stack with popl? Use indirect addressing mode like so:

```
movl (%esp), %eax
```

Remember the "()" or else you will copy the address, %esp, into %eax. Now if we want to access the value right below the top you would do something like this:

```
movl 4(%esp), %eax
```

Likewise if you wanted to access anything further down it would be:

```
movl 4 * indexOfElement(%esp), REGISTER/MEMORY
```

In the C calling convention, the stack is key to implementing a function's local variables, parameters, and return address.

Before executing a function, a program pushes all of the parameters for the function onto the stack in the reverse order that they are documented in. then the program issues a **call** instruction indicating which function it wishes to start. The call instruction does two things. First it pushes the address of the next instruction, which is the return address, onto the stack. Then it modifies the instruction pointer (%eip) to point to the start of the function. So, when the function starts, the stack looks like this (the "top" of the stack being the bottom):

| |
|---|
| Parameter #n |
| … |
| Parameter 2 |
| Parameter 1 |
| Return Address <- %esp |

Each of the parameters have been pushed onto the stack, and finally the return address is stored there where the stack pointer will point to. Now the function itself has some things it needs to do first.

The first thing it needs to do is save the current base pointer, %ebp, by doing pushl %ebp. The base pointer's purpose is to access a functions parameters and local variables. Next it copies the stack pointer into %ebp, with movl %esp, %ebp. This allows you to access the function parameters as fixed indices from the base pointer. Why? Because you might do other things to the stack pointer like push on more parameters for another function call.

Copying the stack pointer into the base pointer will always allow you to know where your parameters are and also your local variables as we will see later on even when you are pushing and popping things off of the stack for other reasons. So it is more or less a constant reference to the stack frame (all of the stack variables used within a function, including parameters, local variables, and the return address).

Next the function reserves space on the stack for any local variable that it needs. This is done by simply moving the stack pointer out of the way. Let's say that we need two words of memory to run a function. We can simply move the stack pointer down (up) two words to reserve the space simply with:

```
subl $8, %esp
```

This subtracts 8 from %esp. This way we can use the stack for variable storage without worrying about clobbering them with pushes that we may make for function calls. Also since it is allocated on the stack, it will disappear when we return from the function which is why they are called local variables.

And now our stack will look like this:

| |
|---|
| Parameter #n ← n * 4 + 4 (%ebp) |
| … |
| Parameter 2 ← 12(%ebp) |
| Parameter 1 ← 8(%ebp) |
| Return Address ← 4(%ebp) |
| Old ebp ← (%ebp) |
| Local Var 1 ←-4(%ebp) |
| Local Var 2 ← -8(%ebp) and (%esp) |

So we can now access any of these all with base pointer addressing mode. This was the original purpose of the base pointer register, %ebp. You can use other registers for this purpose, but in x86, %ebp was built specifically for this purpose.

Global variables and static variables are accessed similarly to how we have been accessing memory in the previous program. The only difference being that static variables are used for only one function, and globals are used by many functions.

Now when a function is done executing it does 3 things:

1. It stores the return value in %eax if it needs one

2. It resets the stack to how it was previously (gets rid of the current stack frame and puts the stack frame of the calling code back into effect).
3. It returns control back to wherever it was called from. This is done using the **ret** instruction, which pops whatever value is at the top of the instruction and sets the instruction pointer, %eip, to that value.

So before a function returns control to the code that called it, it must restore the previous stack frame. Without doing this, the ret instruction will not work. Therefore, before we return we have to reset the stack pointer %esp and base pointer %ebp to what they were before they began. Therefore, in order to return you must do the following:

```
movl %ebp, %esp
popl %ebp
ret
```

At this point, you should consider all your local variables as being deleted. The reason being that future stack pushes will most likely overwrite that data.

Control has now been handed back to the calling code, which can now return %eax for the return value, if it has one. The calling code will then pop off all of the parameters passed into it in order to get the stack pointer back to how it was previously. Alternatively you could add 4 * number of

parameters using the addl instruction, if you don't need those parameters anymore.

Critical:
When you call a function, assume that everything currently in your registers is erased. The only register guaranteed to remain the same is %ebp. %eax is guaranteed to be overwritten. If there are registers you want to save, push them before pushing on the parameters and pop them off in reverse order after popping over the parameters. Even if you know a function won't overwrite anything, do it anyways because future versions might.
Other languages calling conventions may not require you to do this. But since you are reading this, you will probably need to.

## A Functional Example:

*"The difference between gods and daemons largely depends upon where one is standing at the time."* – Aurelian Lorgar, Primarch of the Word Bearers, Horus Rising

Let's take a look at how a function call works in a real program. The function that we are going to write is a simple function that computes the power of two numbers. For the sake of simplicity we will assume that both parameters must be greater than 1.

```
#Purpose: This program finds the power of 2^3 + 3^3
#Everything is stored in registers, but for completeness; keep data sect.
.section data

.section .text
.globl _start
_start:
      pushl $3 #Push second argument
      pushl $2 #Push first argument
      call power #call the function
      addl $8, %esp #Move the stack pointer back (remove the parameters)
      pushl %eax #save the answer of the first call
      pushl $3 #push second argument
```

```
     pushl $3 #push first argument
     call power #call the function
     addl $8, %esp #move the stack pointer back
     popl %ebx #Second answer is in %eax, first answer is on the stack
               #only need to pop it now.
     addl %eax, %ebx #Add them, store result in %ebx
     movl $1, %eax #exit (%ebx is returned)
     int 0x80

#Purpose: This function is used to compute the value of a
#         number raise to a power
#INPUT: First Argument - the base number
#       2nd Argument - the power to raise it to
#Output: Will give the result as a return value
#Notes: Power must be greater than 1

#VARIABLES:
# %eax = temporary storage
# %ebx = base number
# %ecx = power
# -4(%ebp) = current result.

.type power, @function
power:
  pushl %ebp #save old base pointer
  movl %esp, %ebp #make stack pointer the base pointer
  subl $4, %esp #make room for local storage
  movl 8(%ebp), %ebx #Get first parameter
  movl 12(%ebp), %ecx #Get 2nd parameter
  movl %ebx, -4(%ebp) #store the current result.
  loop_start:
    cmpl $1, %ecx #is power 1? We are done else continue
    je loop_end
    movl -4(%ebp), %eax #get current result into %eax.
    imull %ebx, %eax #multiply current result into %eax
    movl %eax, -4(%ebp) #store the current result.
    decl %ecx #decrease the power by 1
    jmp loop_start #go back to beginning!
```

```
86   loop_end:
87     movl -4(%ebp), %eax #return value in %eax
88     movl %ebp, %esp #restore the stack pointer
89     popl %ebp #restore the base pointer
90     ret
```

Type the program in, assemble, run it. Now try calling power with different functions. Also try subtracting the results of the two computations. Try adding a third call to the power function and add its results in.

The main program is simplistic. All it does is push the parameters on, call the function, and restore the stack pointer. Note that between the two calls we store our result into the stack. This is because the only register that is guaranteed to be saved is %ebp.

Now let’s look at our function. Immediately you’ll notice comments, these comments are incredibly useful for those who have to read your code later on and for your own sake.

Now the line immediately following: .type power, @function, tells the linker to treat the symbol **power** as a function. However, since everything is in one file this doesn’t matter as much.

After that we have the label power: which defines the address where the instructions following the label begins. This is how call power works. It transfers control to this point in the program.

Now you might be wondering what the difference between call and jmp is. The difference is that call also pushes a return address onto the stack and a jmp doesn’t do that.

Next we have the lines that set up our function Lines 38-40. At this point our stack will look like the following:

| |
|---|
| Base Number ← 12(%ebp) |
| Power ← 8(%ebp) |
| Return Address ← 4(%ebp) |
| Old %ebp ← (%ebp) |
| Current Result ← -4(%ebp) and (%esp) |

There was an alternative instead of using a local variable, another register since we have a couple that we could use. However, this way was chosen as an example to see how setting up local variables would have been done. More often then not you will run out of registers and will need to keep things on the stack. Other times, your function will call another function and send it a pointer to some of your data. Now this isn't possible with a register.

Basically, the initial steps is to store the base number as a multiplier and the current value (%ebx and -4(%ebp) respectively). It also stores the power in %ecx. Then it loops through and multiplies the current value by the multiplier until the power reaches 1 and exits.

By this point in time you should be able to read this without too much help. Perhaps the only thing you need help with is imull which is integer multiplication and it *stores the result in the* ***2nd operand***. Decl decreases the given register by 1.

## Appendix A Assembly Instructions:

[NOTE: SOME OF THESE YOU WILL NOT NEED UNTIL LATER ON]

**ADD** (Addition)
Syntax: ADD destination, source

The ADD instruction adds a value to a register or a memory address. It can be used in these ways:

These instruction can set the Z-Flag, the O-Flag and the C-Flag (and some others, which are not needed for cracking).

------------------------------------------------------------------------------------------

**AND** (Logical And)
Syntax: AND destination, source

The AND instruction uses a logical AND on two values.
This instruction *will* clear the O-Flag and the C-Flag and can set the Z-Flag.
To understand AND better, consider those two binary values:

```
1001010110
0101001101
```

If you AND them, the result is 0001000100
When two 1 stand below each other, the result is of this bit is 1, if not: The result is 0. You can use calc.exe to calculate AND easily.

------------------------------------------------------------------------------------------

**CALL** (Call)
Syntax: CALL something

The instruction CALL pushes the RVA (Relative Virtual Address) of the instruction that follows the CALL to the stack and calls a sub program/procedure.

CALL can be used in the following ways:

```
    CALL        404000                      ;; MOST COMMON: CALL ADDRESS
    CALL        EAX                         ;; CALL REGISTER - IF EAX WOULD BE 404000 IT
WOULD BE SAME AS THE ONE ABOVE
    CALL        DWORD PTR [EAX]             ;; CALLS THE ADDRESS THAT IS STORED AT
[EAX]
    CALL        DWORD PTR [EAX+5]                   ;; CALLS THE ADDRESS THAT IS STORED
AT [EAX+5]
```

------------------------------------------------------------------------------------------

**CDQ** (Convert DWord (4Byte) to QWord (8 Byte))
Syntax: CQD

CDQ is an instruction that always confuses newbies when it appears first time. It is mostly used in front of divisions and does nothing else then setting all bytes of EDX to the value of the highest bit of EAX. (That is: if EAX <80000000, then EDX will be 00000000; if EAX >= 80000000, EDX will be FFFFFFFF).

-----------------------------------------------------------------------------------------

**CMP** (Compare)
Syntax: CMP dest, source

The CMP instruction compares two things and can set the C/O/Z flags if the result fits.

```
    CMP         EAX, EBX                        ;; compares eax and ebx and sets z-flag if they
are equal
    CMP         EAX,[404000]                    ;; compares eax with the dword at 404000
    CMP         [404000],EAX                    ;; compares eax with the dword at 404000
```

-----------------------------------------------------------------------------------------

**DEC** (Decrement)
Syntax: DEC something

dec is used to decrease a value (that is: value=value-1)

dec can be used in the following ways:

```
    dec eax                                     ;; decrease eax
    dec [eax]                                   ;; decrease the dword that is stored at [eax]
    dec [401000]                                ;; decrease the dword that is stored at [401000]
    dec [eax+401000]                            ;; decrease the dword that is stored at
[eax+401000]
```

The dec instruction can set the Z/O flags if the result fits.

-----------------------------------------------------------------------------------------

**DIV** (Division)
Syntax: DIV divisor

DIV is used to divide EAX through divisor (unsigned division). The dividend is always EAX, the result is stored in EAX, the modulo-value in EDX.

An example:

```
    mov eax,64                                  ;; EAX = 64h = 100
    mov ecx,9                       ;; ECX = 9
    div ecx                                     ;; DIVIDE EAX THROUGH ECX
```

After the division EAX = 100/9 = 0B and ECX = 100 MOD 9 = 1

The div instruction can set the C/O/Z flags if the result fits.

-----------------------------------------------------------------------------------------

**IDIV** (Integer Division)
Syntax: IDIV divisor

The IDIV works in the same way as DIV, but IDIV is a signed division.
The idiv instruction can set the C/O/Z flags if the result fits.

--------------------------------------------------------------------------------------------

**IMUL** (Integer Multiplication)

```
Syntax:    IMUL value
           IMUL dest,value,value
           IMUL dest,value
```

IMUL multiplies either EAX with value (IMUL value) or it multiplies two values and puts them into a destination register (IMUL dest, value, value) or it multiplies a register with a value (IMUL dest, value).

If the multiplication result is too big to fit into the destination register, the O/C flags are set. The Z flag can be set, too.

--------------------------------------------------------------------------------------------

**INC** (Increment)
Syntax: INC register

INC is the opposite of the DEC instruction; it increases values by 1.
INC can set the Z/O flags.

--------------------------------------------------------------------------------------------

**INT**
Syntax: int dest

Generates a call to an interrupt handler. The dest value must be an integer (e.g., Int 21h).
INT3 and INTO are interrupt calls that take no parameters but call the handlers for interrupts 3 and 4, respectively.

--------------------------------------------------------------------------------------------

**JUMPS**
These are the most important jumps and the condition that needs to be met, so that they'll be executed (Important jumps are marked with * and very important with **):

| | | | |
|---|---|---|---|
| JA* | - | Jump if (unsigned) above | - CF=0 and ZF=0 |
| JAE | - | Jump if (unsigned) above or equal | - CF=0 |
| JB* | - | Jump if (unsigned) below | - CF=1 |
| JBE | - | Jump if (unsigned) below or equal | - CF=1 or ZF=1 |
| JC | - | Jump if carry flag set | - CF=1 |
| JCXZ | - | Jump if CX is 0 | - CX=0 |
| JE** | - | Jump if equal | - ZF=1 |
| JECXZ | - | Jump if ECX is 0 | - ECX=0 |

| | | | |
|---|---|---|---|
| JG* | - | Jump if (signed) greater | - ZF=0 and SF=OF (SF = Sign Flag) |
| JGE* | - | Jump if (signed) greater or equal | - SF=OF |
| JL* | - | Jump if (signed) less | - SF != OF (!= is not) |
| JLE* | - | Jump if (signed) less or equal | - ZF=1 and OF != OF |
| JMP** | - | Jump | - Jumps always |
| JNA | - | Jump if (unsigned) not above | - CF=1 or ZF=1 |
| JNAE | - | Jump if (unsigned) not above or equal | - CF=1 |
| JNB | - | Jump if (unsigned) not below | - CF=0 |
| JNBE | - | Jump if (unsigned) not below or equal | - CF=0 and ZF=0 |
| JNC | - | Jump if carry flag not set | - CF=0 |
| JNE** | - | Jump if not equal | - ZF=0 |
| JNG | - | Jump if (signed) not greater | - ZF=1 or SF!=OF |
| JNGE | - | Jump if (signed) not greater or equal | - SF!=OF |
| JNL | - | Jump if (signed) not less | - SF=OF |
| JNLE | - | Jump if (signed) not less or equal | - ZF=0 and SF=OF |
| JNO | - | Jump if overflow flag not set | - OF=0 |
| JNP | - | Jump if parity flag not set | - PF=0 |
| JNS | - | Jump if sign flag not set | - SF=0 |
| JNZ | - | Jump if not zero | - ZF=0 |
| JO | - | Jump if overflow flag is set | - OF=1 |
| JP | - | Jump if parity flag set | - PF=1 |
| JPE | - | Jump if parity is equal | - PF=1 |
| JPO | - | Jump if parity is odd | - PF=0 |
| JS | - | Jump if sign flag is set | - SF=1 |
| JZ | - | Jump if zero | - ZF=1 |

--------------------------------------------------------------------------------------------

**LEA** (Load Effective Address)
Syntax: LEA dest,src

LEA can be treated the same way as the MOV instruction. It isn't used too much for its original function, but more for quick multiplications like this:

```
lea eax, dword ptr [4*ecx+ebx]
```
which gives eax the value of 4*ecx+ebx

--------------------------------------------------------------------------------------------

**MOV** (Move)
Syntax: MOV dest,src

This is an easy to understand instruction. MOV copies the value from src to dest and src stays what it was before.

There are some variants of MOV:

MOVS/MOVSB/MOVSW/MOVSD EDI, ESI: Those variants copy the byte/word/dword ESI points to,
to the space EDI points to.

MOVSX: MOVSX expands Byte or Word operands to Word or Dword size and keeps the sign of the value.

MOVZX: MOVZX expands Byte or Word operands to Word or Dword size and fills the rest of the space with 0.

--------------------------------------------------------------------------------------------

**MUL** (Multiplication)
Syntax: MUL value

This instruction is the same as IMUL, except that it multiplies unsigned. It can set the O/Z/F flags.

--------------------------------------------------------------------------------------------

**NOP** (No Operation)
Syntax: NOP

This instruction does absolutely nothing
That's the reason why it is used so often in reversing ;)

--------------------------------------------------------------------------------------------

**OR** (Logical Inclusive Or)
Syntax: OR dest,src

The OR instruction connects two values using the logical inclusive or.
This instruction clears the O-Flag and the C-Flag and can set the Z-Flag.

To understand OR better, consider those two binary values:

```
                        1001010110
                        0101001101
```

If you OR them, the result is 1101011111

Only when there are two 0 on top of each other, the resulting bit is 0. Else the resulting bit is 1. You can use calc.exe to calculate OR. I hope you understand why, else write down a value on paper and try ;)

--------------------------------------------------------------------------------------------

**POP**
Syntax: POP dest

POP loads the value of byte/word/dword ptr [esp] and puts it into dest. Additionally it increases the stack by the size of the value that was popped of the stack, so that the next POP would get the next value.

-----------------------------------------------------------------------------------------------

**PUSH**
Syntax: PUSH operand

PUSH is the opposite of POP. It stores a value on the stack and decreases it by the size of the operand that was pushed, so that ESP points to the value that was PUSHed.

-----------------------------------------------------------------------------------------------

**REP/REPE/REPZ/REPNE/REPNZ**
Syntax: REP/REPE/REPZ/REPNE/REPNZ *ins*

Repeat Following String Instruction: Repeats ins until CX=0 or until indicated condition (ZF=1, ZF=1, ZF=0, ZF=0) is met. The ins value must be a string operation such as CMPS, INS,
LODS, MOVS, OUTS, SCAS, or STOS.

-----------------------------------------------------------------------------------------------

**RET** (Return)
Syntax: RET
RET digit

RET does nothing but return from a part of code that was reached using a CALL instruction.
RET digit cleans the stack before it returns.

-----------------------------------------------------------------------------------------------

**SUB** (Subtraction)
Syntax: SUB dest,src

SUB is the opposite of the ADD command. It subtracts the value of src from the value of dest and stores the result in dest.

SUB can set the Z/O/C flags.

-----------------------------------------------------------------------------------------------

**TEST**
Syntax: TEST operand1, operand2

This instruction is in 99% of all cases used for "TEST EAX, EAX". It performs a Logical AND(AND instruction) but does not save the values. It only sets the Z-Flag, when EAX is 0
or clears it, when EAX is not 0. The O/C flags are always cleared.

-----------------------------------------------------------------------------------------------

**XOR**
Syntax: XOR dest,src

The XOR instruction connects two values using logical exclusive OR (remember OR uses inclusive OR).

This instruction clears the O-Flag and the C-Flag and can set the Z-Flag.
To understand XOR better, consider those two binary values:

```
1001010110
0101001101
```

If you OR them, the result is 1100011011

When two bits on top of each other are equal, the resulting bit is 0. Else the resulting bit is 1. You can use calc.exe to calculate XOR.
The most often seen use of XOR is “XOR, EAX, EAX”. This will set EAX to 0, because when
you XOR a value with itself, the result is always 0. I hope you understand why, else write down a value on paper and try ;)