Introduction to Assembly

By Alexander Wang

42 Silicon Valley

Chapter 1: x86 Assembly:

"*I find your lack of faith disturbing*" – Darth Vader, Star Wars

Introduction:

"*It's the ship that made the Kessel run in less than twelve parsecs. I've outrun Imperial starships. Not the local bulk cruisers, mind you. I'm talking about the big Corellian ships, now. She's fast enough for you, old man*" – Han Solo

Welcome to Assembly, if you managed to get this far then you have at least a somewhat okay understanding of everything before this point. Assembly is not the greatest language ever invented despite what the author might be telling you. There are significant downsides to using and writing it most of which amounts to the simple fact that *it is very slow to program in*.

Higher level languages despite being much "slower" is significantly easier to be productive in them and modern day compilers will 99% of the time write better assembly than you ever will. So why would you want to learn it? It is *essential* for reverse engineering and exploiting vulnerabilities you might find in a binary.

Even better is that when your debugging through code you might pick up on certain areas that can be written better to either decrease the number of instructions or make it easier on the compiler.

But regardless of the reason for you choosing to learn Assembly, we will start with a basic introduction of the CPU that was barely touched upon in the last chapter.

Your First Program

"*Finally, strategy must have continuity. It can't constantly be reinvented*" – Michael Porter

In this section, you will finally be able to write your first assembly program. In addition you will learn the structure of a basic assembly program and a few instructions (*commands*).

These first few programs may overwhelm you at first. It is absolutely critical that you tinker with them and read through them multiple times as necessary. Even if your tinkering fails, every failure will force you to learn something new.

(Although the exercises will force you to tinker with them. Hint hint. Nudge, nudge).

Entering in the Program

This first program is going to be *very simple*. All it'll do is exit! It's short, but will show some basics about the Assembly Language and Linux. You will need to enter program in an editor exactly as written, with the following name: ***exit.s***. Do not worry about understanding it as we will describe it in-depth in the following section.

```
#Purpose: Simple program that exits and returns status code back
#         the Linux Kernel.
#INPUT: None
#

#OUTPUT: returns a status code. This can be viewed by typing
#                      echo $?
#        after running the program.

#VARIABLES:
# %eax holds the system call number
# %ebx holds the return status
.section data

.section .text
.globl _start
_start:
      movl $1, %eax #this is the Linux kernel command number
                    #(system call) for exiting a program

      movl $0, %ebx #This is the status number we will return to
                    #the operating system. Change this number
```

```
                        #and it will return different things to
                        # echo $?

        int $0x80 #this wakes up the kernel to run the exit comman

```

This is called the source code – human-readable format of a program. In order to transform it into a program that the computer can run we have to *assemble* and *link* it.

The first step is to assemble it. Assembling is the process that transforms what you typed into instructions for the machine. The machine itself only reads sets of numbers, but we prefer words. To assemble the following program type in the command:

as exit.s –o exit .o

as is the command which runs the assembler, exit.s is the source file, and –o exit.o tells the assembler to put its output in a file called exit.o. exit.o is an object file. This is incomplete instructions for a machine. In significantly larger programs you would have many .o's for multiple source files. The *linker* is the program that is responsible for putting the object files together into a program and adding information to it so the kernel knows how to run it. To link run the following command:

ld exit.o –o exit

ld is the command to run the linker, exit.o is the object file we want to link, and –o exit instructs the linker to create an executable called "exit". If any of these commands report an error, you have either mistyped the program, or the command. After correcting the program, you have to rerun all the commands again. *You must always reassemble and re-link programs after you modify the source file*. You can run the program by typing:

./exit

You'll notice that when you type it nothing seems to happen and you go to the next line. That's because this program does literally nothing and exits immediately. However if you run the following command immediately after running the program:

echo $?

It will say 0. Why this happens? It is because whenever a program exits it sends an exit status code that tells the kernel if it exited correctly/fine. If everything is fine, then it will return 0. Typically, 0 means everything is okay, and any other number is an error. It is up to the programmer to decide what the numbers mean when planning out the program.

And now let's go through the program line-by-line.

Outline of a Program:

Take a look at our program above. You might notice a lot of hashes (#), these are comments. Comments are not translated by the assembler. They are used strictly for your own benefit (and for others who might look at your code later on). For Assembly, it is absolutely necessary that you write comments for large blocks of code at a time as reading Assembly even a few hours later is painful. It is good practice to do at a minimum these comments near the beginning:

- Purpose of the code.
- Overview of the processing involved.
- Interesting quirks of the program.

Copied and pasted for your benefit we will go line by line:

```
.section data

.section .text
.globl _start
_start:
      movl $1, %eax #this is the Linux kernel command number
                    #(system call) for exiting a program

      movl $0, %ebx #This is the status number we will return to
                    #the operating system. Change this number
                    #and it will return different things to
                    # echo $?

      int $0x80 #this wakes up the kernel to run the exit comman
```

Anything that begins with a period (.) isn't directly translated into a machine instruction. Instead it's an instruction to the assembler itself called an *assembler directive*. In particular the ***.section*** tells the assembler to split up your program into sections.

The ***.section data*** will hold any memory storage you need for data. It isn't needed in this case because we don't have anything. But, for all later projects this will be necessary. It's just here for completeness.

The ***.section text*** is where the program instructions live.

.globl _start instructs the assembler that *_start* is important to remember. ***_start*** is a *symbol* or label, meaning that it's going to be replaced by something else either during assembly or during linking. Symbols are, in general, used to mark locations of data or addresses that are important so you don't have to memorize memory addresses. Can you imagine how painful it would be to memorize or look up an address for every piece of code or data? In addition, every time you changed something you would have to change all your addresses. Symbols are used so you can concentrate on working functioning code.

.globl is another *assembler directive*, meaning that the assembler should not throw away the symbol that comes immediately afterwards. **_start** is special and always needs to be marked as **.globl** because it marks the starting location of your program. It is imperative that you have that line.

The next line: **_start:** *defines* the value of the _start label. A label is a symbol followed by a colon. As mentioned earlier, if a label's value changes it doesn't matter for you because it will automatically adjust itself when it gets compiled so you don't have to memorize it.

And now we can finally get to an actual instruction for assembly:

movl $1, %eax

Quite literally, it means transfer the number 1 into register eax. In assembly, most instructions have an *operand*. In this case it has two operands, a source and a destination. Operands can be numbers, registers, or memory addresses.

On the x86 processor there is 7 general-purpose registers:

- %eax
- %ebx
- %ecx
- %edx
- %edi
- %esi

In addition there are 4 special registers:

- %ebp
- %esp
- %eip
- %eflags

We'll discuss the special ones later, just know that those ones exist. Some of these registers, namely eip and eflags, can only be accessed with special instructions. The others can be accessed in a similar manner as the general-purpose registers but they have special meanings or uses typically.

So, the movl instruction here moves the number 1 into %eax register. This is called *immediate mode* (refer back to the previous chapter for an explanation). Without the dollar sign, it would be *direct addressing*, i.e. it would go to the memory address 1 and take whatever is inside of it.

The reason for why we are *moving* the number 1 into %eax is that we are preparing to call the Linux Kernel. The number 1 is reserved for the system call: *exit*. We will talk about system calls when we reach our discussion on operating systems and vulnerabilities, but basically they are requests for the operating system’s help. When you make a system call, the number that you want must be stored in %eax. And depending on the system call, other registers might be needed before you can do the system call.

Do note that system call isn’t actually the main use of a register, the next program will actually use it for something else.

Now for a system call, other *parameters* might be necessary. The next line: **movl $0, %ebx**.

For exit, in particular, it requires an extra **parameter** – the exit status code that is seen when you echo $?. In this case we are moving the number 0 into %ebx. We will talk about the other ones as they arise.

The next instruction is where the magic happens: **int 0x80**

The **int** part specifically means an **interrupt**. The 0x80 is the interrupt to be used. You might be wondering why 0x80 instead of “80”. This is because “0x” means a hexadecimal number.

An interrupt interferes with the normal program flow and transfers control from our program to Linux so that it can do a system call. Technically, it transfers control to the *interrupt handler* which then looks up the interrupt in a table. Now if we didn’t signal for an interrupt, then no system call would have been presented.

What’s a Loop?

Now this is a good question. A loop is simply code that is repeated multiple amount of times. I’ll introduce code then explain it:

```
#Purpose: Example of an infinite loop!
#INPUT: None

#OUTPUT: Nothing because infinite loop!

.section data

.section .text
.globl _start
_start:
      startOfLoop:
        cmp $0, $0   #compare zero with zero
        je  startOfLoop #if they are equal, go to top
        jne endofLoop   #if they aren’t equal, exit

      endOfLoop:
        movl $1, %eax #this is the Linux kernel command number
                    #(system call) for exiting a program

        movl $0, %ebx #This is the status number we will return to
                    #the operating system. Change this number
                    #and it will return different things to
                    # echo $?

        int $0x80 #this wakes up the kernel to run the exit command
```

The equivalent Python code for the above Assembly code is:

```
while(1):
    # Don’t do anything

exit(0)
```

Now let’s get to the explanation in particular the important bits. Copying the relevant parts for you here:

```
_start:
      startOfLoop:
        cmp $0, $0   #compare zero with zero
        je  startOfLoop #if they are equal, go to top
        jne endofLoop   #if they aren’t equal, exit

      endOfLoop:
...
```

startOfLoop is yet another “label”. What is it labeling? A memory address. This memory address that is called “*startOfLoop*” will be the start of our loop. It is to maintain a point of reference (meaning that we can go back to it by name instead of memorizing a memory address.

Cmp $0, $0. This is a new operator and it stands for **comparison** (neat naming sense right?). What does it do? An unsigned subtraction (remember what those were from our previous chapter?) and compares the results.

JE startOfLoop – stands for jump if equal to *startOfLoop*. So essentially, it will “move” where you are located to the label that is called “*startOfLoop*”

JNE endOfLoop – stands for jump if **not** equal to endOfLoop, which is similar to the above but instead goes to the memory address that is labeled *endOfLoop*.

endOfLoop is another *label*. This time we are labeling another memory address as the end of our loop in which after we fulfilled our condition we will exit out of our loop. But in this situation we will never get out.

For Loops and If Statements:

Look at the following Python Code:

```
For i in range(0, 10, 1)
    # Don't do anything

x = 0
if x is 0:
    x += 2
else if x is 1:
    x = 1
else:
    x = 2
```

It does the following things:

1. Loop i = 0 to 10; incrementing each time by 1
2. Set variable x = 0
3. Check if variable x = 0
    a. True:
        i. X += 2
    b. False:
        i. Check if variable x = 1
            1. True:
                a. X = 1
            2. False:
                a. X = 2

Now what does this look like in Assembly? Let's go to the next page (ran out of space on this one rip):

```
#Purpose: Example of a for loop, and if-else statement
#INPUT: None

#OUTPUT: Nothing because infinite loop!

#variables:
 #%eax = i
 #%ebx = x

.section data

.section .text
.globl _start
_start:
      movl %eax, 0 #Initialize our starting number for our For Loop
      movl %ebx, 0 #initialize “x” to 0

      startOfForLoop:
        cmp %eax, $10   #compare “i” with 10
        je  endOfLoop   #if I = 10; for-loop is done, exit out
        incl %eax       #increment “i” by 1
        jmp startOfLoop #Go back to top of the loop

      endOfLoop:
         cmp %ebx, $0:      #x == 0?
         jne checkIfX_Is1:  #Go to our 2nd possibility
         addl $2, %eax      #x += 2
         jmp endOfCheckX    #Go to the end or else we might end up going
                            #to another check by accident.
      checkIfX_Is1:
         cmp %ebx, $1       #x == 1?
         jne elseX          #Go to our remaining possibility (else)
         movl $1, %ebx      #X = 1
         jmp endOfCheckX    #Go to the end or else we might end up going
                            #to another check by accident.
      elseX:
         movl $2, %ebx      #x = 2 because didn’t satisfy above criteria
         jmp endofCheckX
```

```
    endofCheckX:
        #do nothing
```

This time I will not go through it entirely line by line. As the explanation is extremely similar to the one above with the infinite loop. Instead I would like to introduce the two new operations that makes for-loops possible and by extension another 2:

addl and incl

Addl – add meaning that it will add two things together (pretty simple).

INCL – Increment a value/register by 1.

Similarly, there is an equivalent for subtraction:

subl and decl

For subtraction and decrement by 1.

Appendix B Flag Register:

The above sections probably was not very clear about why a “jump” occurs. Those two were the easiest, but what about ones that are like:

jnz (Jump not Zero)
jz (Jump Zero)
jle (Jump less than or equal to)
jge (Jump Greater than or Equal to)
jg (Jump greater than)

And so on. Now the last three are somewhat self-explanatory; however, the top two is not. But they all have something to do with this special register called the “*flag register*” – it determines whether or not a “jump” will occur.

Like all other registers, it is 32-bits long. But each individual bit (binary digit) has a special purpose. And here is a nifty chart of all of them (don’t worry you don’t have to memorize all of them):

Bit #	Abbreviation	Description
0	CF	Carry Flag
1		RESERVED always has value 1
2	PF	Parity Flag
3		Reserved
4	AF	Adjust Flag
5		Reserved
6	ZF	Zero Flag
7	SF	Sign Flag
8	TF	Trap Flag (Single step)
9	IF	Interrupt Enable Flag
10	DF	Direction Flag
11	OF	Overflow Flag
12-13	IOPL	I/O Privilege Level (usually always 1)
14	NT	Nested Task Flag (usually always 1
15		RESERVED (usually 0)
16	RF	Resume Flag (386+ only)
17	VM	Virtual 8086 Mode flag (386+ only)
18	AC	Alignment Check (486SX+ only)
19	VIF	Virtual Interrupt Flag (Pentium+)

20	VIP	Virtual Interrupt Pending (Pentium+)
21	ID	Able to use CPUID instruction (Pent+)
22-31		RESERVED

Of these 32-bits you need to memorize at least the following/acronyms: carry (CF), parity (PF), zero (ZF), sign (SF), and overflow (OF). Do not worry though this chart will be in the Appendix when we get to reverse engineering (isn't your author so nice? If you did not notice, some of the information that was introduced previously were added as Appendices and gradually this will turn into a textbook).

Note: If you want to see all the jumps possible please see Appendix C of this book for 99% of the most common operations you will see. We will go in-more depth as we approach Reverse Engineering on this *very special register*.

Appendix C 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 ;)