sensepost

A Crash Course in x86 Assembly for Reverse Engineers

1 TABLE OF CONTENTS

1.1 Introduction

The hardest part of learning x86 assembly in 2013 is finding good tutorials. As the popularity of low level languages decreases the incitement to create fresh up to date tutorials is limited. At the same time x86 assembly is critical in many security related fields such as malware analysis, vulnerability research and exploit development.

This document was created to address the lack of tutorials in a fast, pedagogical and cheap manner. While it is not a complete introduction, it addresses enough to prepare careful readers with the necessary knowledgebase to be able to decipher non-obfuscated assembly. And it does so within less than thirty pages.

For pedagogical reasons focus is kept to what the reader assumedly knows about C or C-based languages (like Java or C#). Hopefully this minimizes the confusion that usually appears when people are trying to learn a stack based language for the first time.

1.2 Bits, bytes, words, double words

The data “types” in 32 bits assembly are bits, bytes, words, and dwords.

The smallest of them is the bit, which can be either 0 or 1.

A byte is eight bits put together and can be between 0 and 255

A word is two bytes put together, or sixteen bits, and can have a maximum value of 65535.

A dword is two words (d in dword stands for double), four bytes or 32 bits. The maximum value is 4294967295.

1.3 Registers

A register is a small storage space available as part of the CPU. This also implies that registers are typically addressed by other mechanisms than main memory and are much faster to access. Registers are used to store values for future usage by the CPU and they can be divided into the following classes.

1.3.1 General purpose registers

Used by the CPU during execution. There are eight of them and the original idea of Intel engineers was that most of these registers should be used for certain tasks (their names hint the type of task intended). But during the years and the development of Intel architecture each and every one of these can be used as general purpose register. That is however not recommended. EBP and ESP should be avoided as much as possible, as using them without saving and restoring their original values means the functions stack frame will be messed up and the program will crash.

EAX)	Extended Accumulator Register
EBX)	Extended Base Register
ECD)	Extended Counter Register
EDX)	Extended Data Register
ESI)	Extended Source Index
EDI)	Extended Destination Index
EBP)	Extended Base Pointer
ESP)	Extended Stack Pointer

All the general purpose registers are 32-bit size in Intel's IA-32 architecture but depending on their origin and intended purpose, a subset of some of them can be referenced in assembly. Below is the complete list.

32 bits	16 bits	8 bit
EAX	AX	AH/AL
EBX	BX	BH/BL
ECX	CX	CH/CL
EDX	DX	DH/DL
ESI	SI	
EDI	DI	
EBP	BP	
ESP	SP	

AX to SP are the 16 bit registers used to reference the 16 least significant bits in their equivalent 32 bit registers. The eight bit registers reference the higher and lower eight bits of the 16 bit registers.

1.3.2 Segment registers

Segment registers are used to make segmental distinctions in the binary. We will approach segments later but in short, the hexadecimal value 0x90 can either represent an instruction or a data value. The CPU knows which one thanks to segment registers.

1.3.3 Status flag registers

Flags are tiny bit values that are either set (1) or not set (0). Each flag represent a status. For example, if the "signed" flag is set, the value of FF will represent a -1 in decimal notation instead of 255. Flags are all stored in special flag register, were many one bit flags are stored at once. The flags are set whenever an operation resulted in certain state or output. The flags we are most interested in for now are:

Z – zero flag, set when the result of the last operation is zero

S – signed flag, set to determine if values should be intercepted as signed or unsigned

O – overflow flag, set when the result of the last operation switches the most significant bit from either F to 0 or 0 to F.

C – carry flag, set when the result of the last operation changes the most significant bit

1.3.4 EIP - Extended Instruction Pointer

The instruction pointer has the same function in a CPU as the needle had in those old gramophones your grandpa used to have. It points to the next instruction to be executed.

1.4 Segments & offsets

Every program consists of several different segments. Four segments that each program must have are *.text*, *.data*, *.stack* and *.heap*. The program code is put in .text and global data is stored in .data. The stack is where, among many things, local variable and function arguments, are stored and the heap is an extendable memory segment that programs can use whenever they need more memory space.

1.4.1 The stack

The stack is the part of memory where a program stores local variables and function arguments (among many things) for later use. It is organized as a “Last In First Out” data structure. When something is added to the stack, it is added on top of it and when something is removed, it is removed from the top. Another very important feature about the stack is that it grows backwards, from the highest memory address to the lowest, more about that in a moment.

Two registers that are customized to work closely with the stack are the ESP and EBP. The ESP is the stack pointer and always points to the top of the stack. When something is added to the stack, the stack grows. This means the ESP needs to be corrected to point to the new “top” of the stack, which is done by *decrementing* ESP. Again, this is because the stack grows backwards, from highest address to lowest.

1.4.2 Stack frames

The EBP is the base pointer but what does base mean? Well, every process has at least one thread, and every thread has its own stack. And within the stack of every thread, each function has its own stack frame. The base is the beginning of a stack frame. The main function in every program has its stack, when it calls a function the called function creates its own stack frame which is marked out by the EBP that points to the beginning of the functions stack frame and the ESP that points to the top of the stack. More about this subject later.

1.4.3 The Heap

The heap is memory space that can be allocated by a process when it needs more memory. Each process has one heap and it is shared among the different threads. All the threads share the same heap.

The heap is a Linked-List data structure, which means each item only knows the position of the immediate items before and after it. When the process does not need the memory anymore, it is custom to “free” the allocated heap. This is done by de-referencing the no longer required portion and allowing other processes to use it.

1.5 Instructions

Intel instructions vary in size from one to fourteen bytes. The opcode (short for operation code) is mandatory for them all and can be combined with other optional or mandatory bytes to create advanced instructions. This is a vast topic and further reading is done at the links below for those who want. If not, the disassembler will do the job for you, but it can be good to know why opcode 83 sometimes is disassembled as an **add** and other times as an **and** instruction when you look in your disassembler. Below links will explain that indirectly.

http://www.swansontec.com/sintel.html
http://ref.x86asm.net/coder32.html

Most instructions have two operators (like **add eax**, **ebx**), but some have one (**not eax**) or even three ("**imul eax, edx, 64**"). Instructions that contain something with "dword ptr [eax]" reference the double word (4 byte) value at memory offset [XXX]. Note that the bytes are saved in reverse order in the memory as Intel uses Little Endian representation. That means the most significant bit of every byte is the most left bit.

1.5.1 Arithmetic operations - ADD , SUB, MUL, IMUL, DIV, IDIV…

ADD, syntax: add dest, src

Destination and source can be either a register like eax, a memory reference [esp] (anything surrounded by square brackets is an address reference). The source can also be an immediate number. Noteworthy is that both destination and source cannot be a memory reference at the same time. Both can however be registers.

```
add eax, ebx        ; both dest and src are registers
add [esp], eax      ; dest is a memory reference to the top of the stack, source
                    ; is the eax register
add eax, [esp]      ; like the previous example but with the roles reversed
add eax, 4          ; source is an immediate value
```

The **sub** instruction works exactly as the **add** instruction.

SUB, syntax: sub dest, src

The *division* and *multiplication* instructions are a little different, let's go through division first.

DIV/IDIV, syntax: div divisor

The dividend is always **eax** and that is also were the result of the operation is stored. The rest value is stored in **edx**.

```
mov eax, 65         ; move the dividend into eax
mov ecx, 4          ; move the divisor into ecx
div ecx             ; divide eax by ecx, this will result in eax containing 16 and
                    ; edx
                    ; containing the rest, which is 1
```

IDIV is the same as DIV but signed division.

MUL/IMUL, syntax:	**mul value**
	mul dest, value, value
	mul dest, value

mul/imul (unsigned/signed) multiply either **eax** with a value, or they multiply two values and put them into a destination register or they multiply a register with a value.

1.5.2 Bitwise operations – AND, OR, XOR, NOT

AND, syntax:	**add dest, src**
OR, syntax:	**or dest, src**
XOR, syntax:	**xor dest, src**
NOT, syntax:	**not eax**

Bitwise operations are what their name suggests. Two pieces of data are being compared bit by bit and depending on the operation, the outcome is either a 0 or a 1. Consider below two values:

value 1:	10011011
value 2:	11001001
output:	????????

If the operation is AND the output would be 10001001 since only the 1st, 5th and 8th bits in both value 1 and 2 are set. That is what AND means, it checks for equally positioned bits that are both set.

If the operation would be OR, it would check for any set bites and as long as a bit is set in either value 1 or value 2, it would set the equivalent bit in the output. Hence the result of an OR would be 11011011.

The XOR is like the OR but with one very important distinction. It will not set bits in the output were both bits are set, instead it will only set bits that are exclusively set in either value 1 but not 2, or the other way around. The above example would give the following output: 01010010.

The way XOR works brings an interesting feature, any value XOR:ed with itself will become 0. Many compilers are making use of this feature of the XOR operation by XOR:ing a register with itself instead of moving the value 0 into it, as the XOR operation will go faster.

The NOT operation is different to the other bitwise operations as it only takes one value and inverses every bit. For example the value 11011110 would become 00100001 when NOT'd.

1.5.3 Branching – JMP, JE, JLE, JNZ, JZ, JBE, JGE…

JMP/JE/JLE…etc syntax: **jmp address**

In assembly, branching is made through the use of jumps and flags. A jump is just an instruction that under certain circumstances will point the instruction pointer (EIP) to another portion of the code (much like the "goto" keyword in C). Flags are, as mentioned previously, tiny one bit values that can be set (1) or not set (0). Most instructions set one or more flags. Let's revisit some of the instructions we already looked at and see which flags they set

ADD can set all of the Z, S, O, C flags (and some more that are of no interest to us right now) according to the result. Same is true for the **SUB** instruction.
The **AND** instruction however always clears the O and C flags, but sets Z and S flags according to the result.

Depending on which flags are set, a jump will either happen or not. As you see, there are always only two options in assembly branches and if you think about it, this is also true in all the more complex type of branches that higher level languages offer. A switch statement in C for example will always perform or not perform a case, then move on to the next case and once again decide whether to perform or not perform that case.

Two notes! First of all, most of the time you will see an instruction called **CMP** (which stands for compare) being used before a jump. CMP is the ideal pre-branch instruction as it can set all the status flags and is really fast. The syntax for CMP is: **cmp dest, src**
This does not mean the other instructions cannot be used before a jump, for example **XOR** occurs frequently but the most common is the **CMP** instruction.

The other important note is about the jump instructions. There are a lot of jump instructions and nobody can memorize them all. Often there are several jumps that look very much alike. For example, **JLE** stands for "Jump Less or Equal". In C this would be:

```
if (x <= y)        { do this }
```

At the same time, **JBE** stands for "Jump Below or Equal". Which in C would be:

```
if (x <= y)        { do this }
```

So why these different jumps that looks exactly the same in C, one wonders? The answer is "due to signed and unsigned comparisons". **JLE** is used to check the flags after a comparison between signed variables and **JBE** for unsigned comparisons. This was just an example, unless you memorize them all, you always need to read in the Intel Developer's Guide to see which flags a jump checks for.

1.5.4 Data moving – MOV, MOVS, MOVSB, MOVSW, MOVZX, MOVSX, LEA…

MOV, syntax:	**mov dest, src**
MOVSB, syntax:	**movzx dest, src**
MOVZX, syntax:	**movzx dest, src**

MOV moves data from source into destination. Both source and destination can be register, or one of them register and the other one a memory reference. Both cannot be a memory reference however.

The mov instructions come in many flavours, just like the jump instructions, and partly for the same reason. **MOVS/MOVSB/MOVSW/MOVSD** for example copy a byte, word or dword from source to destination.

The mov instructions that have an X in their name are used for variable extension. In C it would for example be like a typecast from char to integer, like this

```
char a = 'h';
int b;
b = (int)a;
```

The instructions work like this

```
MOVSX)      DEST ← Signextend[SRC]
MOVZX)      DEST ← Zeroextend[SRC]
```

Where signed means the extension bits will hold the value of one.
Another instruction that can be used for data moving is the **LEA** instruction. LEA stands for "Load Effective Address" and the syntax looks like this:

```
lea eax, dword ptr[ecx+edx]          ; This will store ecx+edx in eax
```

1.5.5 Loops – LOOP, REP…

Although one can create neat loops using jumps, Intel's x86 assembly also provides instructions specifically tailored to create iterating sequences of code. Like many of the other instructions we looked at, they come with many flavours depending of the size and sign of the variables they work with. For simplicity reasons, I will only show the easiest cases, **LOOP** first:

```
mov ecx, 5      ; remember ecx stands for extended counter register
_proc:
dec ecx         ; decrements ecx
loop _proc      ; loops back to _procs, second row
```

REP instructions work like LOOP instructions, but are specifically customized to handle strings (this is where IA-32 assembly almost becomes a high level language in my world ☺).

Syntax:

```
mov esi, str1
mov edi, str2
mov, ecx, 10h
rep cmps
```

What happens here is that the strings to be compared are loaded into ESI and EDI and then a comparison is performed for 16 bytes (hexadecimal value 0x10 = 16 in decimal notation). If at some point the source and destination are not equal, a flag will be set and the operation will be aborted.

1.5.6 Stack management – POP, PUSH

POP, syntax: **pop dest**

PUSH, syntax: **push var/reg**

The **POP** and **PUSH** instructions are probably the easiest instructions in assembly. They both do what their names suggest to the stack. The **POP** instruction pops a value or memory address (which also is a value) from the stack and stores it in the destination. Additionally it also *increments* the stack pointer (ESP) to point to the new top of the stack (remember the stack grows backwards which means higher addresses when it shrinks).

The **PUSH** instruction pushes a value to the stack and *decrements* the stack pointer to point to the new top.

1.5.7 Functions – CALL, RET

Wake up, this is going to become a bit heavy as I've seen many people make mistakes here!

CALL is like a jump with several differences. A jump instruction loads an address into EIP and continues execution from there. A CALL however stores the current EIP on the stack, with the expectation to reload it once the calleé (that is the called function) is done. A jump instruction has no way to do that as the current position of the EIP is not stored.

CALL, syntax: CALL _function

When the above instruction is reached, following steps occur.

1. EIP is stored to the stack ; This is done by the CALL instruction
2. **EBP is stored to the stack ; This is where the heavy will start as this is**
3. **EBP is made to point to ESP ; actually a "calling convention" abstraction**
4. ESP is decremented to, among several things, contain the local variables of _function
5. EIP is loaded with the address of _function

Point **1** is performed by the CALL instruction, while **point 2** and **3** are performed…by what?

So here comes a very important distinction between assembly and binaries. Assembly is the instruction set of the architecture. It can be x86, ARM, PowerPC, RISC, etc…
A binary program is however

1) assembly instructions for a certain architecture meant to run on…
2) …a certain operating system…
3) …compiled by a certain compiler

For example, Windows binary files follow the PE format while Linux binary files follow the ELF format. And different compilers will give different output binary for the same C-code.
In the five step ladder above, point **1** is performed by the **CALL** instruction while there is no specific instruction for point **2** and **3**. These are instead written into the binary, as separated push and and mov instructions and referred to as a "function prolog", which look like this:

```
push ebp
mov ebp, esp
sub esp, 10h            ; The value 10h is an example value
```

When a calleé has finished executing, the caller's EBP is popped back into the EBP.
Then the **RET** instruction removes the stack-frame of the calleé by incrementing the ESP (again, remember the stack shrinks when incremented) and pop the old saved EIP into EIP so that execution can continue where it left of.
RET, syntax: RET / RET num

Mindful readers might now be thinking: *"But hey, when I write functions in C I sometimes choose to declare return values, what happens to those?"*

I'm glad you asked and the answer is that the return value, as we know it in C, is stored in EAX before the execution is returned to the caller.

1.5.8 Interrupts, Debugger traps – INT, trap flag

INT, syntax: int num ; were “num” represents an interrupt handler

Interrupts are used to tell the CPU to halt the execution of a thread. They can be hardware based, software based or exception based (for example unauthorized memory access attempt). When the INT instruction is hit, the execution is moved to an exception handler, which is defined by **num**. Some INT flavours do not require a **num** value, **INT3** for example.

When a software based breakpoint is set in an assembly level debugger like OllyDBG the instruction where the breakpoint is supposed to hit is exchanged to an int3 instruction, which has the hexadecimal value of 0xCC. And when the interrupt is hit, the control of the thread is handed back to the debugger. At the same time, the trap flag is set. When a program is single stepped in a debugger, the CPU is checking for the **trap flag**. If the trap flag is set, the CPU will execute one instruction and give control of the thread back to the debugger.

Again, there are other flavours of breakpoints like conditional breakpoints, memory breakpoints and hardware breakpoints. This was just a detailed explanation of software breakpoints to demonstrate the idea of breakpoints.

1.6 Calling conventions

The previous chapter discussed the CALL and RET instructions and some in-detail description off what happens on the stack during a function call. The problem with that description is that it depends on the compiler and hence is not always true.

1.6.1 stdcall

The calling convention we described before is named **stdcall**. In the stdcall, function arguments are passed from right to left and the calleé is in charge of cleaning up the stack. Return values are stored in EAX. The stdcall is a combination of two other calling conventions, pascal and the cdecl.

1.6.2 cdecl

The **cdecl** (short for c declaration) is a calling convention that originates from the C programming language and is used by many C compilers for the x86 architecture. The main difference of cdecl and stdcall is that in a cdecl, the caller, not the calleé, is responsible for cleaning up the stack.

1.6.3 pascal

The **pascal** calling convention origins from the Pascal programming language and the main difference between it and **stdcall** is that the parameters are pushed to the stack from left to right.

1.6.4 fastcall

The **fastcall** is a non-standardized calling convention. It is usually recognized through the way it sends function arguments. While all the above conventions use the stack to store the function arguments, the fastcall convention tends to load them into registers. This results in less memory interaction and increases the performance of a call, hence the name.

1.6.5 Others calling conventions

This was just a fast introduction to some of the most common calling conventions. Both Wikipedia and Intel have great summaries on this topic, for those who favour more in-deep knowledge.

http://en.wikipedia.org/wiki/X86_calling_conventions

1.7 C to x86 assembly

1.7.1 Single-Branch Conditionals

1.7.1.1 C

```
if (var == 0) {
        aFunction();
}
// AfterCondition
…
```

1.7.1.2 x86 assembly

```
mov     eax, [var]
test    eax, eax
jnz     AfterCondition
call    aFunction
AfterCondition:
...
```

1.7.2 Two-way Conditionals

1.7.2.1 C

```
if (var == 7)
        function();
else
        anotherFunction();
…
```

1.7.2.2 x86 assembly

```
cmp     [var], 7
jz      ElseBlock
call    function
jmp     AfterConditionalBlock
ElseBlock:
call    anotherFunction
AfterConditionalBlock:
...
```

1.7.3 Logical operations - AND

1.7.3.1 C

```
if (var1 == 100 && var2 == 50)
        bla bla;
...
```

1.7.3.2 x86 assembly

```
cmp     [var1], 100
jne     AfterCondition
cmp     [var2], 50
jne     AfterCondition
bla bla
AfterCondition:
...
```

1.7.4 Logical operations - OR

1.7.4.1 C

```
if (var1 == 100 || var2 == 50)
        function();
...
```

1.7.4.2 x86 assembly

```
cmp     [Variable1], 100
je      ConditionalBlock
cmp     [var2], 50
je      ConditionalBlock
jmp     AfterConditionalBlock
ConditionalBlock:
call    function
AfterConditionalBlock:
...
```

1.7.5 Loops

1.7.5.1 C

```
c = 0;
while (c < 1000) {
        array[c] = c;
        c++;
}
```

1.7.5.2 x86 assembly

```
mov ecx, DWORD PTR [array]
xor eax, eax
LoopStart:
mov DWORD PTR [ecx+eax*4], eax
add eax, 1
cmp eax, 1000
jl LoopStart
```

1.7.6 Function calls

1.7.6.1 C

```
function (int x, char y );
```

1.7.6.2 x86 assembly

```
mov eax, y
push eax
mov eax, x
push eax
call function
```

1.8 Reverse engineering tools

The most common reverse engineering tools are a debugger, a disassembler and a hex editor. Branch standard for the former two are **OllyDBG** and **IDA Pro**. **HxD** is an easy, lightweight and powerful hex editor. Although a hex editor rarely requires an introduction, Olly and IDA certainly do.

1.8.1 OllyDBG tutorials

http://thelegendofrandom.com/blog/archives/31

http://thelegendofrandom.com/blog/archives/115

1.8.2 IDA Pro tutorials

https://www.hex-rays.com/products/ida/support/tutorials/