PART 6

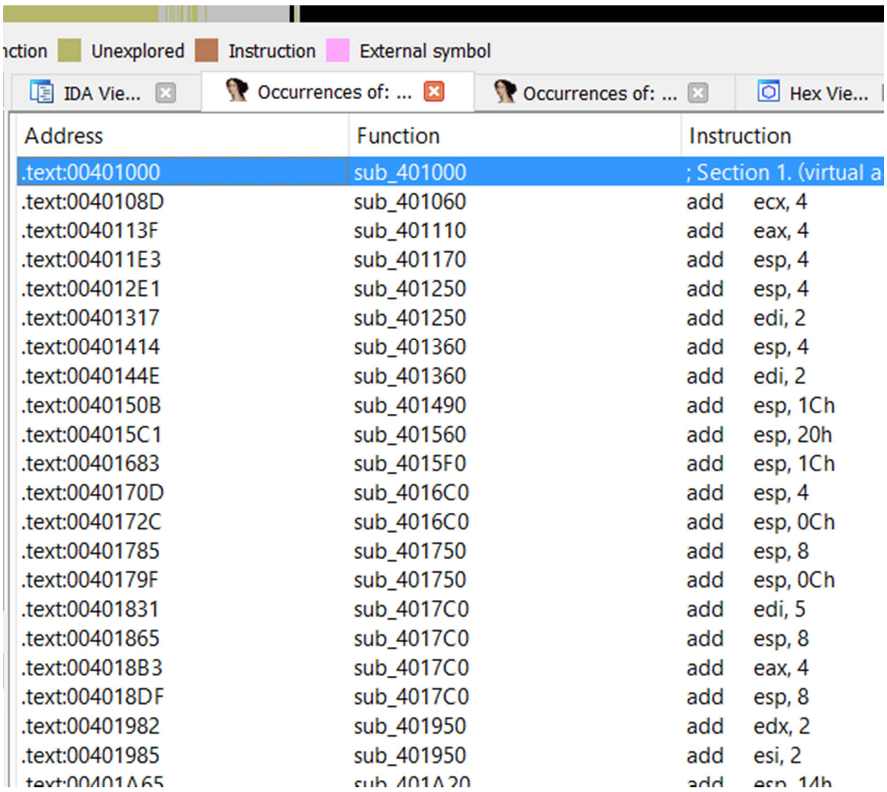
Let's continue with arithmetic and logical instructions.

ADD A,B

It adds the value from **B** to **A** and saves the result in **A**.

A can be a register or the content of a memory position. **B** can be a register, a constant or the content of a memory position. **A** and **B** can't be the content of a memory position at the same time in the same instruction.

Let's see some **ADD** examples searching for the text: **ADD** in VEVIEWER.



Address	Function	Instruction
.text:00401000	sub_401000	; Section 1. (virtual a
.text:0040108D	sub_401060	add ecx, 4
.text:0040113F	sub_401110	add eax, 4
.text:004011E3	sub_401170	add esp, 4
.text:004012E1	sub_401250	add esp, 4
.text:00401317	sub_401250	add edi, 2
.text:00401414	sub_401360	add esp, 4
.text:0040144E	sub_401360	add edi, 2
.text:0040150B	sub_401490	add esp, 1Ch
.text:004015C1	sub_401560	add esp, 20h
.text:00401683	sub_4015F0	add esp, 1Ch
.text:0040170D	sub_4016C0	add esp, 4
.text:0040172C	sub_4016C0	add esp, 0Ch
.text:00401785	sub_401750	add esp, 8
.text:0040179F	sub_401750	add esp, 0Ch
.text:00401831	sub_4017C0	add edi, 5
.text:00401865	sub_4017C0	add esp, 8
.text:004018B3	sub_4017C0	add eax, 4
.text:004018DF	sub_4017C0	add esp, 8
.text:00401982	sub_401950	add edx, 2
.text:00401985	sub_401950	add esi, 2
.text:00401A65	sub_401A20	add esp, 14h

There, we see many addition examples where the first member is a register and the second one is a constant. As we know, it will be added to the value that register has at that moment, the constant value, and will be saved in the register.

```
0040107D mov     large fs:0, eax
00401083 mov     [ebp+var_10], ecx
00401086 mov     [ebp+var_4], 0FFFFFFFFh
0040108D add     ecx, 4
00401090 call    ds:??1locale@std@@QAE@XZ
00401096 mov     ecx, [ebp+var_C]
00401099 mov     large fs:0, ecx
```

In this example, if ECX equals 10000, the 4 constant is added to it and the result is 10004 that is saved in the same ECX.

```
00413C80
00413C80
00413C80 ; Attributes: bp-based frame
00413C80
00413C80 sub_413C80 proc near
00413C80
00413C80 arg_0= dword ptr  8
00413C80
00413C80 push    ebp
00413C81 mov     ebp, esp
00413C83 mov     ecx, [ebp+arg_0]
00413C86 add     dword ptr [ecx+30h], 0FFFFFFFFh
00413C8A mov     eax, [ecx+30h]
00413C8D jnz     short loc_413C9A
```

It will add **0xFFFFFFFF** to the previous value in the content of the address that **ECX+30** points to if that address has write permission, it will add them and save the result there.

If ECX equals 0x10000 in 0x10030, for example, and the content is 1 when we add it 0xFFFFFFFF that -1, the result is 0 and it is saved in 0x10030.

In the CRACKME.EXE, there is some example of a two-register addition.

```
004013CC add     edi, ebx
004013CE inc     esi
004013CF jmp     short loc_4013C6
```

There, both registers will be added and saved in EDI. Of course, we can also add 16-bit and 8-bit registers.

ADD AL,8

ADD AX,8

ADD BX,AX

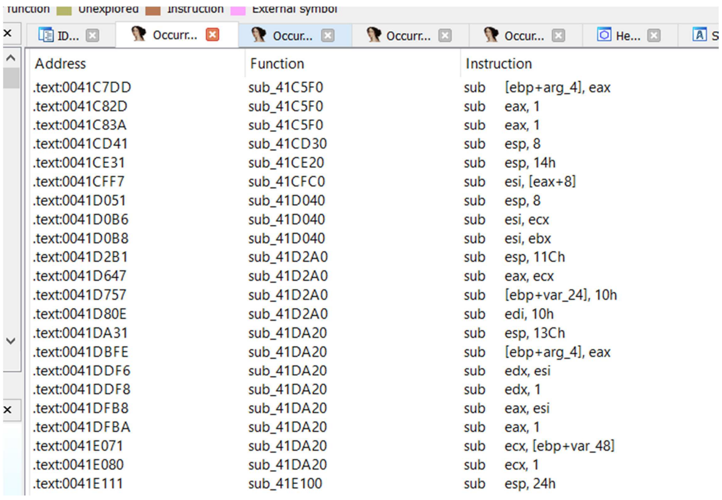
ADD byte ptr ds: [EAX],7

It will add **7** to the byte of the **EAX** content and it will be saved in the same place.

And all the possible combinations of additions with registers, memory position contents. All of them are valid, except if **A** is a constant and both are memory position contents at the same time in the same instruction.

SUB A,B

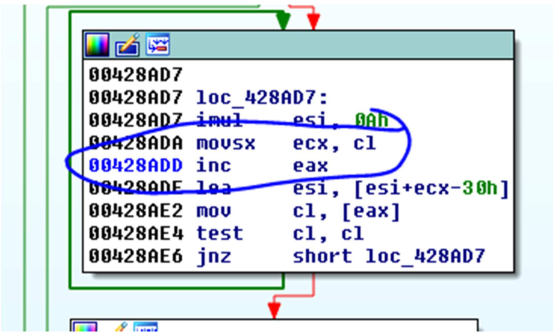
It's exactly the same as **ADD** but instead of adding it will subtract integers and save the result in **A**. The combinations are the same.



Address	Function	Instruction
.text:0041C7DD	sub_41C5F0	sub [ebp+arg_4], eax
.text:0041C82D	sub_41C5F0	sub eax, 1
.text:0041C83A	sub_41C5F0	sub eax, 1
.text:0041CD41	sub_41CD30	sub esp, 8
.text:0041CE31	sub_41CE20	sub esp, 14h
.text:0041CFF7	sub_41CFC0	sub esi, [eax+8]
.text:0041D051	sub_41D040	sub esp, 8
.text:0041D0B6	sub_41D040	sub esi, ecx
.text:0041D0B8	sub_41D040	sub esi, ebx
.text:0041D2B1	sub_41D2A0	sub esp, 11Ch
.text:0041D647	sub_41D2A0	sub eax, ecx
.text:0041D757	sub_41D2A0	sub [ebp+var_24], 10h
.text:0041D80E	sub_41D2A0	sub edi, 10h
.text:0041DA31	sub_41DA20	sub esp, 13Ch
.text:0041DBFE	sub_41DA20	sub [ebp+arg_4], eax
.text:0041DDF6	sub_41DA20	sub edx, esi
.text:0041DDF8	sub_41DA20	sub edx, 1
.text:0041DFB8	sub_41DA20	sub eax, esi
.text:0041DFBA	sub_41DA20	sub eax, 1
.text:0041E071	sub_41DA20	sub ecx, [ebp+var_48]
.text:0041E080	sub_41DA20	sub ecx, 1
.text:0041E111	sub_41E100	sub esp, 24h

INC A and DEC A

Increase or decrease 1 to a register or a memory position content. It is a special addition and subtraction case.



Both are used to increase or decrease 1 in counters at a time.

IMUL

It is the integer multiplication and there two ways.

IMUL A,B

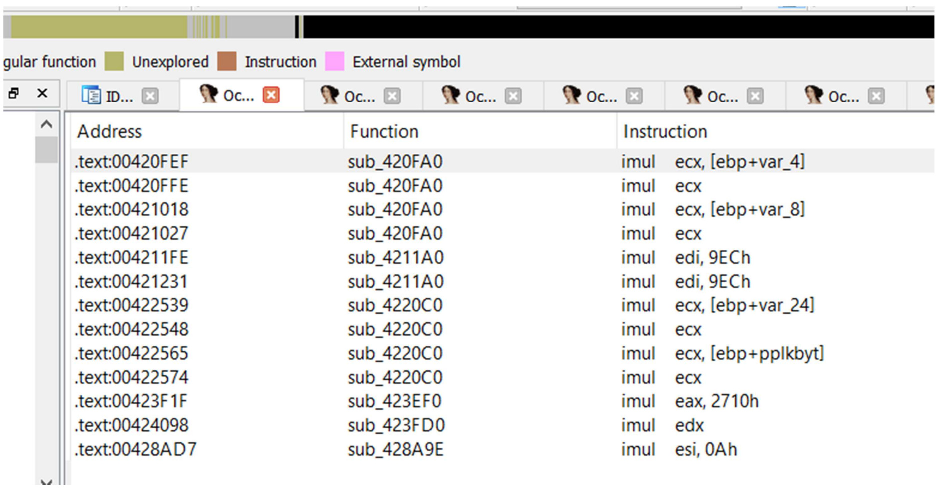
IMUL A,B,C

The first one does the integer multiplication of both **A** and **B** and the result is saved in **A** and the second one multiplies B and C and saves the result in **A**.

In both cases, **A** can only be a register, **B** can only be a register or memory position content and **C** can only be a constant.

imul eax, [ecx]
imul esi, edi, 25

Some example in VEVIEWER:



Address	Function	Instruction
.text:00420FEF	sub_420FA0	imul ecx, [ebp+var_4]
.text:00420FFE	sub_420FA0	imul ecx
.text:00421018	sub_420FA0	imul ecx, [ebp+var_8]
.text:00421027	sub_420FA0	imul ecx
.text:004211FE	sub_4211A0	imul edi, 9ECh
.text:00421231	sub_4211A0	imul edi, 9ECh
.text:00422539	sub_4220C0	imul ecx, [ebp+var_24]
.text:00422548	sub_4220C0	imul ecx
.text:00422565	sub_4220C0	imul ecx, [ebp+pplkbyt]
.text:00422574	sub_4220C0	imul ecx
.text:00423F1F	sub_423EF0	imul eax, 2710h
.text:00424098	sub_423FD0	imul edx
.text:00428AD7	sub_428A9E	imul esi, 0Ah

There are just examples of the first way. In both cases, it will multiply in an integer way both members and save the result in the first one.

There are no examples of the first way:

IMUL EAX,EDI,25

It multiplies EDI by 25 and saves the result in EAX. It is easy.

IDIV A

In this case, **A** just marks the divider of the operation. As the dividend as the quotient are not specified because they are always the same.

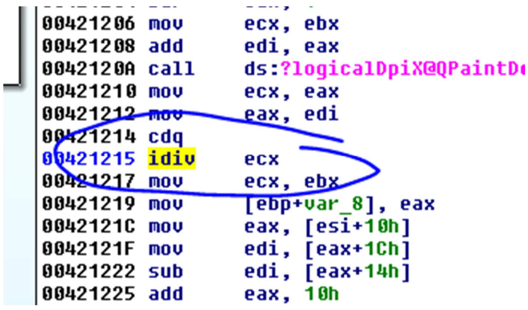
D : d = c

The **dividend** (D) is the number to be divided by other.

The **divider** (d) is the number divided by the dividend.

The **quotient** (q) is the result of the division.

This operation creates a bigger 64-bit number with EDX as a high part and EAX as a low part. It divides that by **A**, saves the result in EAX and the rest in EDX



If EAX = 5, EDX = 0 and ECX = 2, it will make an integer division. The result of 5/2 will be 2. It will be saved in EAX and the rest 1 in EDX.

The same will happen if A is the content of a memory position content.

EDX:EAX will be divided by that value and it will save the result in EAX and the rest in EDX.

LOGICAL OPERATIONS

AND, OR or XOR

AND A,B

It will AND both values and save the result in A. The same happens with OR or XOR. Each one has its truth table. It is applied to each member and the result is saved in **A**.

A and **B** can be registers or memory address contents, but it is illegal that both are memory contents at the same time in the same instruction.

The most used cases are XOR of a same register to change it into 0 easily.

XOR EAX,EAX or any other value will be 0 because the XOR truth table is:

ENTRADAS A	B	AND	NAND	OR	NOR	EXOR
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0
funciones		$A \cdot B$	$\overline{A \cdot B}$	$A+B$	$\overline{A+B}$	$A \oplus B$

In this case, the result is the last column and we see that if we xor a number by itself, the result will be 0. This operation is done in binary mode.

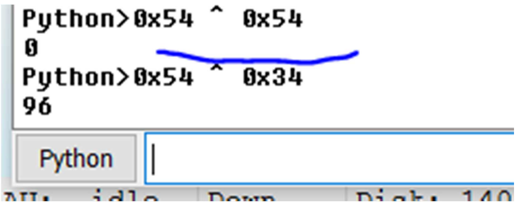
ENTRADAS A	B	AND	NAND	OR	NOR	EXOR
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0
funciones		$A \cdot B$	$\overline{A \cdot B}$	$A+B$	$\overline{A+B}$	$A \oplus B$

```
Python>bin(0b111101 ^ 0b111101)
0b0
```

Python

Writing it as binary in the Python bar and using ^ that is xor in Python,   we see that xoring two equal numbers the result is always 0.

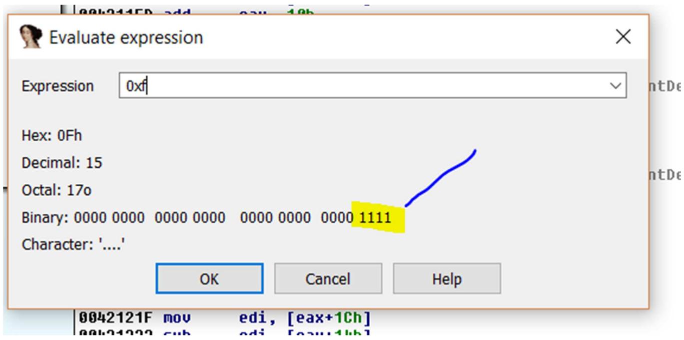
Of course, we can do that with hex and decimal numbers. I just used binary to see how it affects each bit.



Another simple use is:

AND EAX, 0F

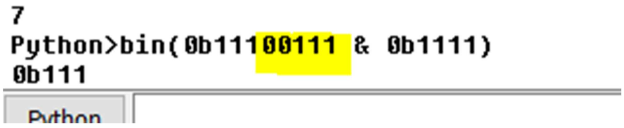
As 0F is 1111 in binary.



ENTRADAS A	B	AND	NAND	OR	NOR	EXOR
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0
funciones		$A \cdot B$	$\overline{A \cdot B}$	$A+B$	$\overline{A+B}$	$A \oplus B$

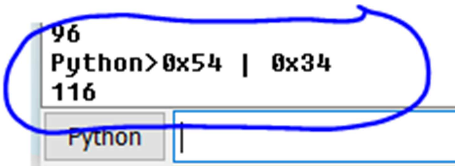
As the second bit is 1, the result won’t change while the other bits will be 0.

This way, I zero out all the bits of a number and leave the last 4 bits intact.



AND is **&** in Python and the result is 0B0111 that was the last original four bits.

OR is the vertical bar **|** in Python.

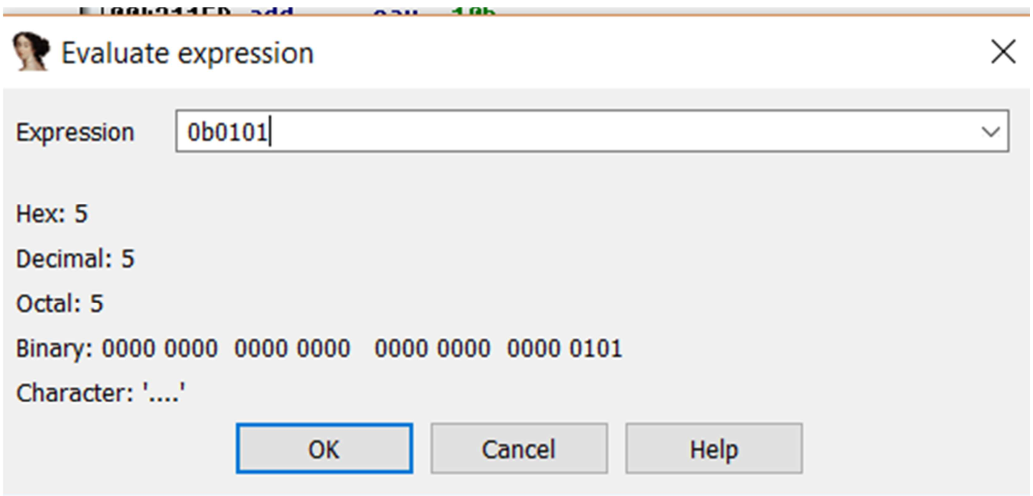


We can always use a scientific calculator or Python to solve the operation without changing both numbers into binary and seeing the bit to bit process that is a heavy task.

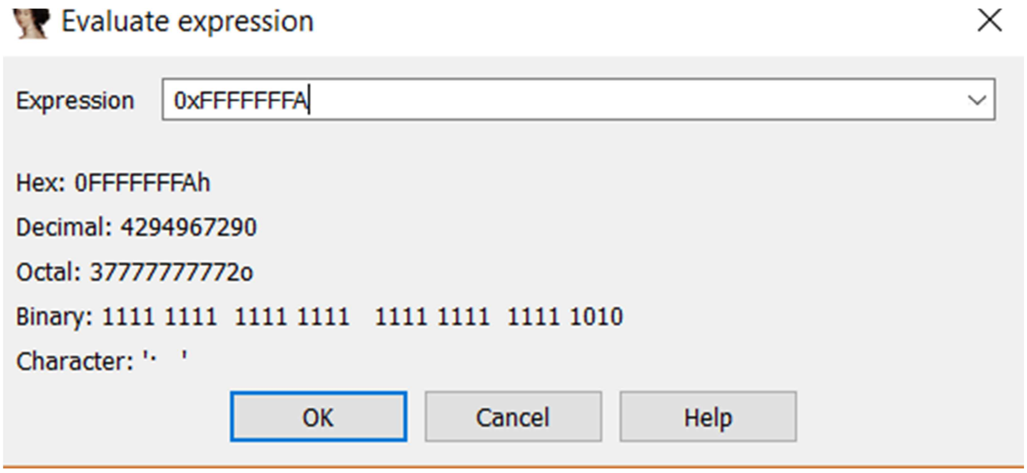
NOT A

It inverts all the bits of **A** and saves them in **A**.

Python doesn't have the **NOT** instruction, but it is easy if you have 0101 and you apply NOT.



The result will be the inversion of each bit.



All the 0's change into 1's and viceversa.

NEG A

NEG A changes **A** into **-A**.

It's not exactly like ~ in Python because this one subtracts 1.

~ x

Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as -x - 1.

To **NEG** in Python, we need to add **1** to the result.

```
Python>hex(~ 0x45+1)
-0x45
Python>hex(~ -0x45+1)
0x45
Python
```

SHL, SHR

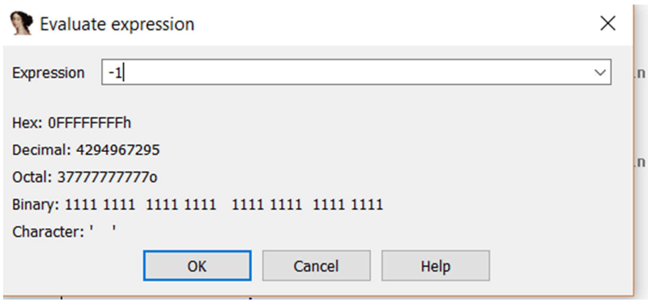
SHL A,B

SHR A,B

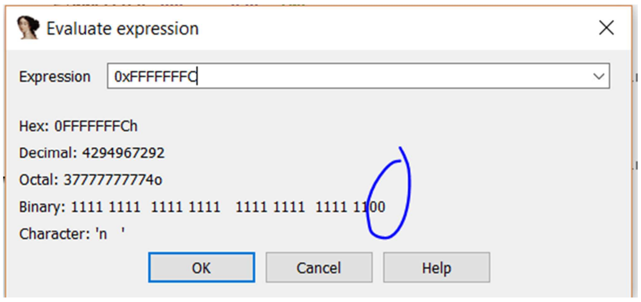
A can be a register or a memory position and **B** a constant or an 8-bit register.

This instructions shift to left (SHL) or right (SHR). The bytes that are missing are replaced by 0. Let's see some example.

If I have **-1**…



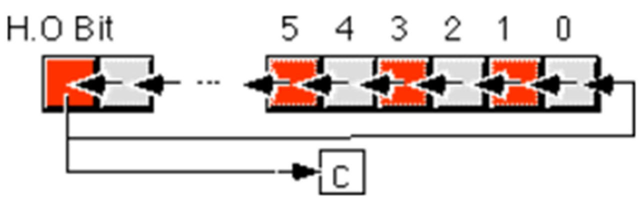
And I do **SHL 2**, it will be:



When moving the bits to the left, two bits fell down in the left side and they are filled with two 0's in the right side.

The same happens if we use SHR. The bits will be move to the right and the bits that fall down in the right side will be replaced with 0's in the left side.

We also have the **ROL** and **ROR** instructions that are similar. They shift certain bits, but the bits that fall down for one side return with their same value for the other side. It is a pure rotation because no bit is modified. They are just rotated.



Ricardo Narvaja

Translated by: @IvinsonCLS