Assembler : The Basics In Reversing

Indeed: the basics!! This is all far from complete but covers about everything you need to know about assembler to start on your reversing journey! Assembler is the start and the end of all programming languages. After all, all (computer LOL) languages are translated to assembler. In most languages we deal with relatively clear syntaxes. However, it's a completely other story in assembler where we use abbreviations and numbers and where it all seems so weird …

I. Pieces, bits and bytes:

- **BIT -** The smallest possible piece of data. It can be either a 0 or a 1. If you put a bunch of bits together, you end up in the 'binary number system'

  i.e. 00000001 = 1 00000010 = 2 00000011 = 3 etc.

- **BYTE -** A byte consists of 8 bits. It can have a maximal value of 255 (0-255). To make it easier to read binary numbers, we use the 'hexadecimal number system'. It's a 'base-16 system', while binary is a 'base-2 system'

- **WORD** - A word is just 2 bytes put together or 16 bits. A word can have a maximal value of 0FFFFh (or 65535d).

- **DOUBLE WORD -** A double word is 2 words together or 32 bits. Max value = 0FFFFFFFF (or 4294967295d).

- **KILOBYTE -** 1000 bytes? No, a kilobyte does NOT equal 1000 bytes! Actually, there are 1024 (32*32) bytes.

- **MEGABYTE -** Again, not just 1 million bytes, but 1024*1024 or 1,048,578 bytes.

II. Registers:

Registers are “special places” in your computer's memory where we can store data. You can see a register as a little box, wherein we can store something: a name, a number, a sentence. You can see a register as a placeholder.

On today’s average WinTel CPU you have 9 32bit registers (w/o flag registers). Their names are:

EAX: Extended Accumulator Register
EBX: Extended Base Register
ECX: Extended Counter Register
EDX: Extended Data Register
ESI: Extended Source Index
EDI: Extended Destination Index
EBP: Extended Base Pointer
ESP: Extended Stack Pointer
EIP: Extended Instruction Pointer

Generally the size of the registers is 32bit (=4 bytes). They can hold data from 0-FFFFFFFF (unsigned). In the beginning most registers had certain main functions which the names imply, like ECX = Counter, but in these days you can - nearly - use whichever register you like for a counter or stuff (only the self defined ones, there are counter-functions which need to be used with ECX). The functions of EAX, EBX, ECX, EDX, ESI and EDI will be explained when I explain certain functions that use those registers. So, there are EBP, ESP, EIP left:

EBP: EBP has mostly to do with stack and stack frames. Nothing you really need to worry about, when you start. ;)

ESP: ESP points to the stack of a current process. The stack is the place where data can be stored for later use (for more information, see the explanation of the push/pop instructions)

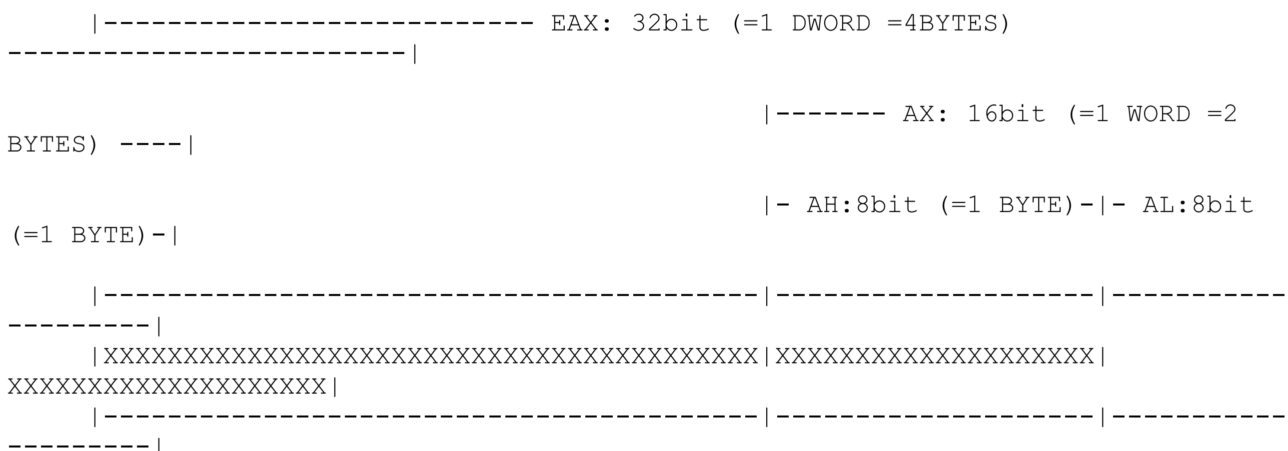
EIP: EIP always points to the next instruction that is to be executed.

There's one more thing you have to know about registers: although they are all 32bits large, some parts of them (16bit or even 8bit) can not be addressed directly.

The possibilities are:

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

A register looks generally this way:



So, EAX is the name of the 32bit register, AX is the name of the "Low Word" (16bit) of EAX and AL/AH (8bit) are the “names” of the "Low Part" and “High Part” of AX. BTW, 4 bytes is 1 DWORD, 2 bytes is 1 WORD.

REMARK: make sure you at least read the following about registers. It’s quite practical to know it although not that important.

All this makes it possible for us to make a distinction regarding size:

- **i. byte-size registers:** As the name says, these registers all exactly 1 byte in size. This does not mean that the whole (32bit) register is fully loaded with data! Eventually empty spaces in a register are just filled with zeroes. These are the byte-sized registers, all 1 byte or 8 bits in size:
    - AL and AH
    - BL and BH
    - CL and CH
    - DL and DH

- **ii. word-size registers:** Are 1 word (= 2 bytes = 16 bits) in size. A word-sized register is constructed of 2 byte-sized registers. Again, we can divide these regarding their purpose:
    - 1. general purpose registers:

      AX (word-sized) = AH + AL -> the '+' does *not* mean: 'add them up'. AH and AL exist independently, but together they form AX. This means that if you change AH or AL (or both), AX will change too!

      -> 'accumulator': used to mathematical operations, store strings,..

      BX -> 'base': used in conjunction with the stack (see later)

      CX -> 'counter'

      DX -> 'data': mostly, here the remainder of mathematical operations is stored

      DI -> 'destination index': i.e. a string will be copied to DI

      SI -> 'source index': i.e. a string will be copied from SI

    - 2. index registers:

      BP -> 'base pointer': points to a specified position on the stack (see later)
      SP -> 'stack pointer': points to a specified position on the stack (see later)

    - 3. segment registers:

      CS -> 'code segment': instructions an application has to execute (see later)
      DS -> 'data segment': the data your application needs (see later)
      ES -> 'extra segment': duh! (see later)
      SS -> 'stack segment': here we'll find the stack (see later)

    - 4. special:

      IP -> 'instruction pointer': points to the next instruction. Just leave it alone ;)

- **iii. Doubleword-size registers:**

2 words = 4 bytes = 32 bits. EAX, EBX, ECX, EDX, EDI…

If you find an 'E' in front of a 16-bits register, it means that you are dealing with a 32-bits register. So, AX = 16-bits; EAX = the 32-bits version of EAX.

III. The flags:

Flags are single bits which indicate the status of something. The flag register on modern 32bit CPUs is 32bit large. There are 32 different flags, but don't worry. You will mostly only need 3 of them in reversing. The Z-Flag, the O-Flag and the C-Flag. For reversing you need to know these flags to understand if a jump is executed or not. This register is in fact a collection of different 1-bit flags. A flag is a sign, just like a green light means: 'ok' and a red one 'not ok'. A flag can only be '0' or '1', meaning 'not set' or 'set'.

- The Z-Flag:
  - The Z-Flag (zero flag) is the most useful flag for cracking. It is used in about 90% of all cases. It can be set (status: 1) or cleared (status: 0) by several opcodes when the last instruction that was performed has 0 as result. You might wonder why "CMP" (more on this later) could set the zero flag, because it compares something - how can the result of the comparison be 0? The answer on this comes later ;)
- The O-Flag:
  - The O-Flag (overflow flag) is used in about 4% of all cracking attempts. It is set (status: 1) when the last operation changed the highest bit of the register that gets the result of an operation. For example: EAX holds the value 7FFFFFFF. If you use an operation now, which increases EAX by 1 the O-Flag would be set, because the operation changed the highest bit of EAX (which is not set in 7FFFFFFF, but set in 80000000 - use calc.exe to convert hexadecimal values to binary values). Another need for the O-Flag to be set, is that the value of the destination register is neither 0 before the instruction nor after it.
- The C-Flag:
  - The C-Flag (Carry flag) is used in about 1% of all cracking attempts. It is set, if you add a value to a register, so that it gets bigger than FFFFFFFF or if you subtract a value, so that the register value gets smaller than 0.

IV. Segments en offsets

A segment is a piece in memory where instructions (CS), data (DS), the stack (SS) or just an extra segment (ES) are stored. Every segment is divided in 'offsets'. In 32-bits applications (Windows 95/98/ME/2000), these offsets are numbered from 00000000 to FFFFFFFF. 65536 pieces of memory thus 65536 memory addresses per segment. The standard notation for segments and offsets is:

SEGMENT : **OFFSET** = Together, they point to a specific place (address) in memory.

See it like this:

A segment is a page in a book : **An offset is a specific line at that page.**

V. The stack:

The Stack is a part in memory where you can store different things for later use. See t as a pile of books in a chest where the last put in is the first to grab out. Or imagine the stack as a paper basket where you put in sheets. The basket is the stack and a sheet is a memory address (indicated by the stack pointer) in that stack segment. Remember following rule: the last sheet of paper you put in the stack, is the first one you'll take out! The command 'push' saves the contents of a register onto the stack. The command 'pop' grabs the last saved contents of a register from the stack and puts it in a specific register.

VI. INSTRUCTIONS *(alphabetical)*

Please note, that all values in ASM mnemonics (instructions) are **always** hexadecimal.

Most instructions have two operators (like "add EAX, EBX"), but some have one ("not EAX") or even three ("IMUL EAX, EDX, 64"). When you have an instruction that says something with "DWORD PTR [XXX]" then the DWORD (4 byte) value at memory offset [XXX] is meant. Note that the bytes are saved in reverse order in the memory (WinTel CPUs use the so called "Little Endian" format. The same is for "WORD PTR [XXX]" (2 byte) and "BYTE PTR [XXX]" (1 byte).

Most instructions with 2 operators can be used in the following ways (example: add):

```
add eax,ebx                          ;; Register, Register
add eax,123                          ;; Register, Value
add eax,dword ptr [404000]           ;; Register, Dword Pointer [value]
add eax,dword ptr [eax]              ;; Register, Dword Pointer [register]
add eax,dword ptr [eax+00404000]     ;; Register, Dword Pointer [register+value]
add dword ptr [404000],eax           ;; Dword Pointer [value], Register
add dword ptr [404000],123           ;; Dword Pointer [value], Value
add dword ptr [eax],eax              ;; Dword Pointer [register], Register
add dword ptr [eax],123              ;; Dword Pointer [register], Value
add dword ptr [eax+404000],eax       ;; Dword Pointer [register+value], Register
add dword ptr [eax+404000],123       ;; Dword Pointer [register+value], value
```

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 **):

Jump		Description	Condition
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 ;)

VII. Logical Operations

Here follow the most used in a reference table.

Reference Table

operation	src	dest	result
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0
NOT	0	N/A	1
	1	N/A	0

Yep, indeed, this is already the END ☺ Sorry if there are mistakes in this text.