**NZNWS + Kordia MARE workshop Cheat Sheets**

# **Registers**

## What is Register?

Registers are the variables in the assembly language. The size of each register in x86 architecture is 32 bits. Registers are located in the processor and each processor can have its own proprietary registers.

## General Purpose Registers

General purpose registers can be divided into 3 subcategories:

* Data Registers
* Pointer Registers
* Index Registers

## Data Registers

Data registers are the ones that contain data.

**EAX (Accumulator Register)**

The most basic register used in arithmetic operations is the "EAX" register. It stores the results of arithmetic operations and the return values of functions. For example, it is used for operations such as addition and multiplication.

**EBX (Base Register)**

It is the register that holds the base address of the program.

**ECX (Counter Register)**

The “**ECX**” is the register that is used as counter. It is used in loop and string operations.

**EDX (Data Register)**

It is a register that is generally used for holding data. It is also used in I/O (Input/Output) operations.

## Pointer Registers

Pointer registers are the ones that hold memory addresses.

**EBP (Base Pointer)**

EBP is the one that holds the lowest address of the Stack and is used for local variables.

**ESP (Stack Pointer)**

It is the register that holds the top address of the stack. Displays the last element that entered the Stack.

**EIP (Instruction Pointer)**

The EIP register may be the most important register as it holds the address of the next instruction to be executed in the program flow. In other words, if the value of this register is changed, the flow of the program can be interfered with.

**2.1.9. Index Registers**

Index registers are used for index information storage.

**ESI (Source Index)**

It is the register that holds the source index information for string operations. It holds the address of where the data will be read.

**EDI (Destination Index)**

It is the register that holds the destination index information for string operations. It holds the address of where the data will be written.

## Segment Registers

Segment registers are the registers used to hold addresses of specific segments in memory.

**Stack Segment (SS)**

It is the segment register that holds the base location address of the stack.

**Code Segment (CS)**

It is the register which is known as “**.text**” and it holds the address of the code segment used for data access.

**Data Segment (DS)**

It is the register which is also known as “**.data**” and it holds the address of the data segment which is the default variable location for data access.

**Extra Segment (ES)**

It is the register that holds the address of the extra segment used during string operations.

## EFLAGS Register - Status Flags

The EFLAGS register, unlike other registers, is a special register where each bit has a different meaning. The values of the bits of this register allow the CPU operations to be controlled and monitored.

**Adjust Flag (AF)**

It is the flag that is set when there is a transfer from the 3rd bit to the 4th bit in arithmetic operations.

**Carry Flag (CF)**

It is the flag that is set in arithmetic operations when the value of the register is more than the maximum value or if the value is less than the minimum value of the register. For example, the description over 4 bits is as follows:

**Example 1**:

Initially: “**CF = 0**”

1111 + 0001 = 0000 (CF = 1)

0000 - 0001 = 1111 (CF = 0)

**Example 2**:

Initially: “**CF = 0**”

0000 - 0001 = 1111 (CF = 1)

**Example 3**:

Initially: “**CF = 0**”

0111 + 0001 = 1000 (CF = 0)

1000 - 0001 = 0111 (CF = 0)

**Direction Flag (DF)**

It is the flag that determines the direction in the transport and comparison of string data. The string operation is performed from left to right in case of “**DF = 0**”, and from right to left in case of “**DF = 1**”.

**Interrupt Flag (IF)**

It is the flag that determines whether external interruptions are taken into consideration and therefore it is the flag that determines whether the necessary operation is implemented. Keyboard entry would be a good example to the external interruption. Interruptions are ignored in case of "**IF = 0**" and considered disabled, and are applied to the process in case the "**IF = 1**" state.

**Overflow Flag (OF)**

The overflow flag is set when a positive value for the signed integer is too large to be represented in the register, or when a negative value is too small.

**Parity Flag (PF)**

It is the flag that shows the total number of "**1**" bits in the results of arithmetic operations. If the total number of “1” bits is even, it becomes “PF = 1”. If the total number of “**1**” bits is odd, it becomes “**PF = 0**”.

**Example 1**:

10111100 + 00010001 = 11001101 (PF = 0)

**Example 2**:

10111100 + 00010000 = 11001100 (PF = 1)

**Sign Flag (SF)**

It is the flag that indicates whether the result of an arithmetic operation is negative or positive. If the result of the arithmetic operation takes a negative value, the sign flag is set as “**SF = 1**”. In case of a positive result, the sign flag will be in the state of “**SF = 0**”.

**Trap Flag (TF)**

It is the flag that allows the processor to set the operating mode as single-step mode. The debugger program sets this flag to run each command one by one. In this way, each command executed by the processor at the assembly level can be executed step by step.

**Zero Flag (ZF)**

It is the flag set depending on whether the result of the arithmetic or the comparison operations is "**0**" (zero). If the result of the operation is “**0**”, then the  zero flag is set as “**ZF = 1**”. If the operation result is other than zero, then the zero flag is not set “**ZF = 0**”.

# **X86 Assembly Language and CPU Instructions**

## What is X86 Assembly Language?

Assembly language can be described as a machine language which is different for each processor. Assembly language is more difficult to understand than other programming languages. Running programs can be analyzed at the processor level only through the assembly language. This training describes the x86 assembly language.

## Addressing Modes

The assembly language has many different addressing modes. Some of these are as follows:

**Register Addressing**

In this addressing, registers are used as operands. For example:

MOV EAX, EBX

Above, the operation is applied between two registers.

**Immediate Addressing**

A fixed value is used as the operand in this addressing. For example:

MOV EAX, 0x0

The operation is applied between the fixed value and the register in the above operation.

**Memory Addressing**

The memory address is used as the operand in this addressing. For example:

MOV EAX, DWORD PTR [ESP]

The operation is applied between the memory address and the register in the above operation.

## CPU Instructions - 1

Instructions are the commands in the assembly language. There are a lot different type of instructions for different tasks and purposes. Some may have similar duties. It is vital to know the instructions in order to follow the program flow in the assembly language. Below are some of the instructions and their tasks:

## Arithmetic Instructions

Arithmetic instructions are those that perform arithmetic operations between operands. For example, the four arithmetic operations are examples of such transactions.

**ADD**

The “**ADD**” is the instruction that enables to perform the addition. For example:

ADD    ESP, 0x8

With the above instruction, 8 is added to the value of the ESP register.

**SUB**

The "**SUB**" is the instruction that enables the subtraction. For example:

SUB    ESP, 0x4

With the above instruction, 4 is subtracted from the value of the ESP register.

**MUL**

The "**MUL**" is the instruction that enables multiplication to be performed.

**DIV**

The "**DIV**" is the instruction that enables division to be performed.

**INC**

The “**INC**” is an instruction that allows you to increment the value of the operand by 1. For example:

INC    EBX

With the above instruction, the value of the EBX register is incremented by 1.

**DEC**

The “**DEC**” is an instruction that allows you to decrement the value of the operand by 1. For example:

DEC    EBX

With the above instruction, the value of the EBX register is decremented by 1.

**Branch Instructions**

Branch instructions are those that performs the comparison and/or branching. They are vital with respect to follow the program flow in the assembly language. Below are some of these instructions:

**JMP**

The “**JMP**” is the instruction that allows branching unconditionally. It takes the memory address as an operand. For example:

JMP    0x56556020

With the above instruction, the program flow branches to the memory address given as operand (branching).

**JZ/JE**

The “**JZ**” and “**JE**” instructions are among the conditional branching instructions. They stand for:

JZ = Jump if Zero

JE = Jump if Equal

JE and JZ instructions take memory address as operand. For example:

JE     0x5555555551b5 <main+277>

With the above instruction, the condition for branching to the memory address in the operand is that the zero flag is set to "1" in the form of "**ZF=1**".

**JNZ/JNE**

The “**JNZ**” and “**JNE**” instructions are also among the conditional branching instructions. They stand for:

JNZ = Jump if not Zero

JNE = Jump if not Equal

JNE ve JNZ instructions take memory address as operand. For example:

JNE    0x565561e7 <main+78>

The condition for branching to the memory address in the operand with the above instruction is that the zero flag is set to "0" in the form of "**ZF=0**".

**CALL**

The “**CALL**” instruction is the instruction used for function call. It takes a function address as an operand. For example:

CALL   0x56556199 <function1>

With the above instruction, the function named “**function1**” is called. There are 2 basic operations with this instruction:

* The address of the instruction after the CALL instruction in the program flow is pushed to the stack (Return address)
* The value of the EIP register is set as the function address, so that the program flow branches to the corresponding function.

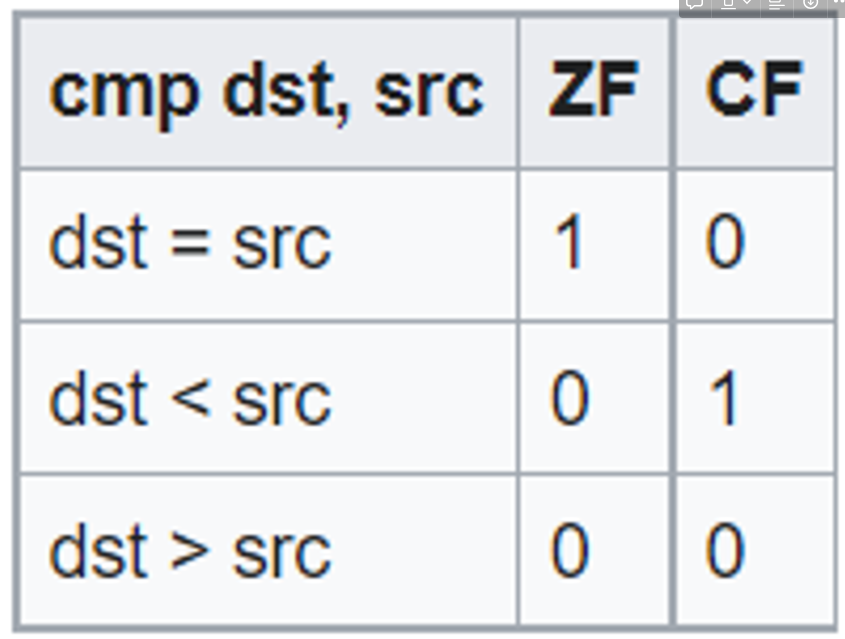
**CMP**

The “**CMP**” instruction is used for comparison operations. Takes 2 values to compare as operands. For example:

CMP    EDX, EAX

The above instruction compares the EAX and EDX registers. Depending on the result of the comparison, “**Zero Flag(ZF)**” and “**Carry Flag(CF)**” may be changed. The below table shows the conditions that will cause these changes.

(Image Source: <https://www.aldeid.com/wiki/X86-assembly/Instructions/cmp>)



## Data Transfer Instructions

There are many data transfer instructions in assembly language that are used for different purposes. Some of these are listed below:

**MOV**

The “**MOV**” is the most basic data transfer instruction used to assign a value to a register or to an address in the memory. For example:

MOV    EAX, 0x0

With the above instruction, the value “**0**” (zero) is assigned to the EAX register.

**LEA**

The “**LEA**” is the instruction used to assign a memory address to the target. It stands for "**LEA: Load Effective Address**". For example:

LEA    ECX, [esp+0x4]

With the above instruction, the memory address is assigned to the ECX register.

**XCHG**

The "**XCHG**" instruction allows to exchange values in 2 registers. For example:

EAX = 0x2

EBX = 0x3

Let the register values be as above.

XCHG    EAX, EBX

After the above instruction is executed, the updated values of the registers are as follows:

EAX = 0x3

EBX = 0x2

**PUSH**

The "**PUSH**" is the instruction that allows adding data to the stack. For example:

PUSH    EDX

**POP**

The "**POP**" is the instruction that extracts data from the stack. For example:

POP    EDX

## Logical Instructions

Assembly language has many instructions that are used for logical operations. Some of these instructions are described below:

**AND**

The “**AND**” is the instruction that enables to implement the logical AND operation. For example:

AND    ESP, 0xfffffff0

With the above instruction, the fixed value and the ESP register are ANDed.

**OR**

The “**OR**” is the instruction that enables to implement the logical OR operation. For example:

OR    EAX, 0xfffffff0

With the above command, the fixed value and the EAX register are ORed.

**XOR**

The “**XOR**” is the instruction that enables to execute the logical XOR (exclusive OR) operation. For example:

XOR    EBP, EBP

With the above instruction, the EBP register is XORed with itself.

**NOT**

The "**NOT**" is the instruction that enables to implement the bitwise inversion operation. For example:

EAX = 0x626c7565 (Hexadecimal)

Let the EAX register have the above value.

0x626c7565 => Binary => 01100010 01101100 01110101 01100101

NOT    EAX

New EAX Value = 10011101100100111000101010011010 (Binary)

New EAX Value = 0x9d938a9a (Hexadecimal)

With the above instruction, the bitwise NOT operation has been executed successfully and with that, the EAX register has a new value.

## NOP Instruction

The "**NOP**" instruction means "**no operation**" that allows to move on to the next instruction without any operation being executed. It is used alone without the operand.

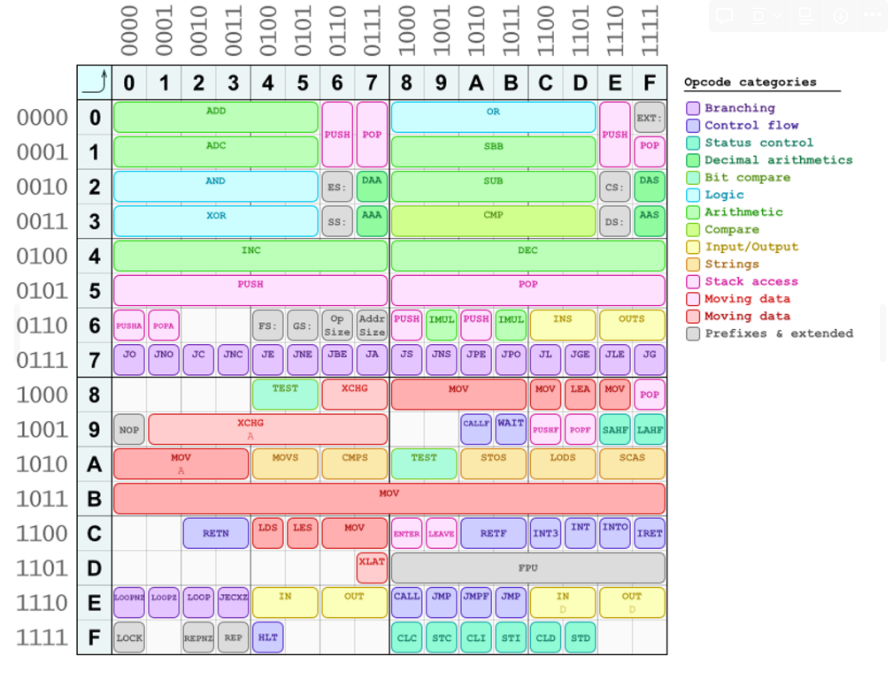
**Note**: More instructions for X86 Assembly architecture can be found at:

**X86 Assembly Instructions**: <https://www.aldeid.com/wiki/X86-assembly/Instructions>

**X86 Assembly Instructions(Wikipedia)**: <https://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086/8088_instructions>

## What is Opcode(Operation Code)?

Opcode(Operation Code) is a unique value that belongs to each instruction. Thanks to this value, the machine understands which instruction to execute. The following image shows the opcodes of the instructions according to the x86 architecture:



Opcodes are usually expressed in hexadecimal notation. For example, let's see the opcode of the "**NOP**" instruction according to the image above:

A screenshot of a computer

Description automatically generated

After finding the "**NOP**" instruction on the image, we should first look at its equivalent on the left lines: "**9**", then, check the column equivalent at the top: “**0**”. The combination of these hexadecimal values creates the opcode: “**0x90**”.

**NOP Instruction Opcode**: 0x90

## What is Assembler and Disassembler?

Assembler is a compiler that converts the source code written in assembly language to machine code.

A green line with purple text

Description automatically generated

“**The Netwide Assembler (NASM)**”, is an example of an assembler:

**The Netwide Assembler (NASM)**: <https://www.nasm.us/>

Disassembler is a tool that helps acquire the assembly code of the executable binary file.

A green line with purple text

Description automatically generated

Examples of disassembler include:

* IDA Pro (Windows & Linux) -

<https://www.hex-rays.com/ida-pro/>

* Hopper Disassembler (Linux) -

<https://www.hopperapp.com/>

* Binary Ninja -

<https://binary.ninja/>

* Objdump (Linux) -

<https://man7.org/linux/man-pages/man1/objdump.1.html>

## What is Debugging and Debugger?

Debugging is the in-depth analysis process in order to see the detailed and step-by-step operations of the programs during operation. Debugging can be done for many different purposes. For example, it can be used to test the program. If debugging is applied within the scope of reverse engineering, the aim is to learn the flow of the program and to reveal its processes. Debugging can be done in high-level programming languages such as C#, as well as in low-level languages such as Assembly.

Special programs are needed for debugging that are also known as "debuggers". For example, a debugger can be used in the "Visual Studio" software for debugging C# source codes. There are different debugger tools specific to each operating system in order to perform debugging at the Assembly language level. Below are some of the debuggers:

* OllyDbg (Windows) -

<https://www.ollydbg.de/>

* ImmunityDebugger (Windows) -

<https://www.immunityinc.com/products/debugger/>

* GDB (Linux) -

<https://www.sourceware.org/gdb/>

* IDA Pro (Windows & Linux) -

<https://hex-rays.com/ida-pro/>

* X64dbg (Windows) -

<https://x64dbg.com/>

* Windbg (Windows) -

<https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools>

### **Memory Layout**

## What is Memory?

Memory is one of the most basic hardware units used for running the programs and the operation of the computer. When the computer is started, the operating system which is actually a software, the system software is loaded into the memory. The operating system is much larger than other software. Memory is a storage unit that is temporarily used during the execution of programs. Today, programs use memory for data storage because they use too much program data in size to be kept in processor variables. Memory contains some data structures that have their own special functions.

**Note**: System level memory analysis is an advanced level subject that the SOC analyst should have knowledge of. The SOC analyst should also be able to perform memory analysis if needed. You can access the "**Memory Forensics**" training at the link below:

**Memory Forensics**:

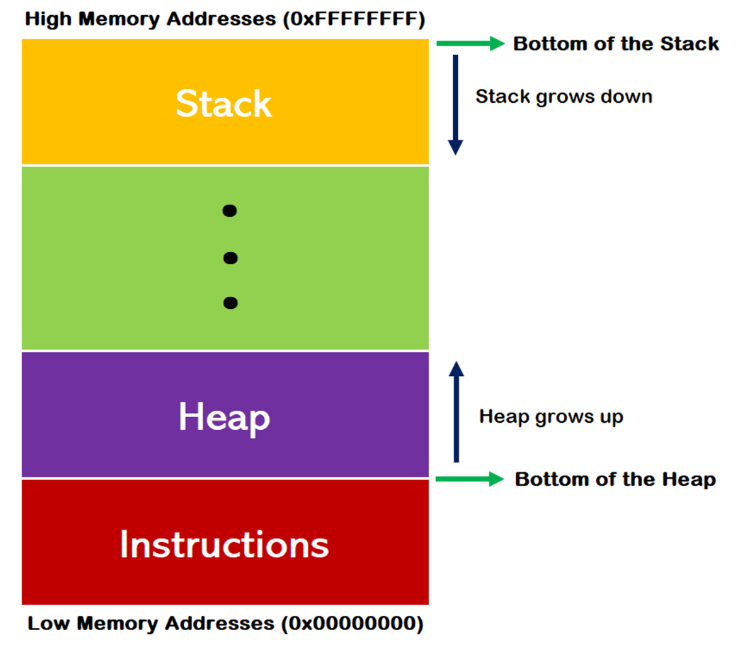
## What is Stack and Heap?

A “**Stack**” is a section of memory that is allocated when a program is run. Stack contains local variables and function arguments of the program.

“**Heap**” is a partition in memory that is reserved when a program is run. There are dynamic variables created for the program in the heap. There are no static variables in the heap as it is in the stack. Compared to the stack, heap is larger and has more freedom inside the operating system.

## How Stack and Heap Work?

Stack is an area that expands from high memory addresses to low memory addresses. The heap is the exact opposite and expands from low memory addresses to high memory addresses.



## Stack Operations

Stack applies a certain method when importing or extracting data from its data structure. This method is called “**Last In First Out (LIFO)**”. In this method, the data that enters the data structure last is removed from the stack in the first place. There are two basic functions in the Stack data structure. These are the "**Push**" and "**Pop**" functions.

**Push**

“**Push**” is the function that provides data intake to the stack. The below image shows how a few data are placed on the stack with the push function respectively, and the working principle of the stack:

**Pop**

“Pop” is a function that helps extract data from the stack. The below image shows the working principle of the stack as well as how some data that were previously placed in the stack in accordance with its working logic, are removed from the stack with the pop function

# **Tools notes**

## UPX

UPX: UPX (the Ultimate Packer for eXecutables) is a free and open-source executable packer supporting a number of file formats.

https://en.wikipedia.org/wiki/UPX

## IDA

IDA: The Interactive Disassembler (IDA) is a disassembler for computer software which generates assembly language source code from machine-executable code.

https://en.wikipedia.org/wiki/Interactive\_Disassembler

## Olevba tool

Olevba: While there isn't a dedicated Wikipedia page for Olevba, it is a script that can parse OLE compound files, such as Microsoft Office documents or Outlook messages, mainly for malware analysis and debugging.

https://www.decalage.info/python/oletools

## dnSpy tool

dnSpy: dnSpy is a debugger and .NET assembly editor. There is no specific Wikipedia page for it, so the link provided directs you to its GitHub repository.

https://github.com/dnSpy/dnSpy

## peepdf

peepdf: Peepdf is a Python tool to explore PDF files in order to find out if the file can be harmful or not. Again, no specific Wikipedia page is available, so the link provided is to its GitHub repository.

https://github.com/jesparza/peepdf

## Debugger

Debugger: A debugger or debugging tool is a computer program used to test and debug other programs.

https://en.wikipedia.org/wiki/Debugger

# **Free learning References**

## **Malware analysis**

### Malware Analysis workshops

<https://malwareunicorn.org/#/workshops>

### Malware Analysis online learning free courses

<https://exploitation.ashemery.com/>

## Malware Analysis In 5+ Hours - Full Course - Learn Practical Malware Analysis! by HuskyHacks <https://lnkd.in/eR3_ki-6> Malware Analysis – Mind Map by Thatintel <https://lnkd.in/evyAhNWt>

## Malware Analysis Tutorials: a Reverse Engineering Approach by Dr Xiang Fu <https://lnkd.in/eHZFTSqp> Malware Analysis and Reverse Engineering courses by [DFIR Diva](https://www.linkedin.com/company/dfirdiva/) <https://lnkd.in/eCxGV2iP>

## Reverse Egineering

### [Reverse Engineering for Beginners by](https://www.linkedin.com/feed/update/urn:li:activity:7080137856442085376/)[Ophir Harpaz](https://www.linkedin.com/in/ACoAABqXu_oBya-g0puXCr4Ghqe9XrdLbKFuiKk) <https://www.begin.re/> Reverse Engineering for Everyone by [Kevin Thomas](https://www.linkedin.com/in/ACoAAAzt8mABFPKjmD7gNK9R3kcvzuxIrd7h8wY) My Technotalent <https://lnkd.in/eUqUDdXS> Reverse Engineering for beginners by Dennis Yurichev (available in many languages) <https://lnkd.in/eHsdurZG> Reverse Engineering 101 by 0x00 (with exercises) <https://lnkd.in/ebtKjS-W>