**Chapter 3: ENVIRONMENT SETUP & BASIC SYNTAX**

**Topic – 1: Setting Up NASM In Linux**

**Check Processor Info**

***uname -a***

* The **processor** must be **x86-64**.

**Install NASM Assembler**

***sudo pacman -S nasm***

**Topic – 2: Assembling Program**

**Make Object File**

***nasm -f elf64 -o hello.o hello.asm***

**Make Executable File**

***ld hello.o -o hello***

**Run The Program**

***./hello***

**Topic – 2: Testing Basic Assembly Program**

**First NASM Program**

* Let’s create a program that **adds two numbers** & **stores** the sum in a variable.

***section .data***

***sum dd 0***

***section .text ; code section***

***global \_start ; must be used for GCC***

***\_start: ; main procedure starts***

***mov eax, 5***

***add eax, 6***

***mov [sum], eax ; [] is strictly needed***

***; Exiting the program***

***mov eax, 1 ; sys\_exit system call***

***xor ebx, ebx ; returns 0***

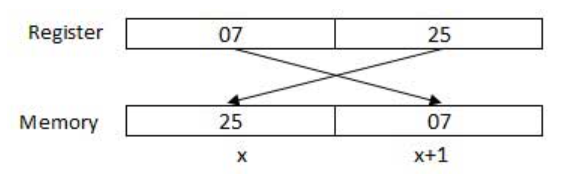
***int 0x80 ; interrupt to call sys\_exit***

* ***DWORD*** assigns **32-bits** of memory.
* Notice that we can put **any data type** variable with that assigned space.
* ***.data*** & ***.code*** are segments, we will see ***.stack*** segment later.
* **Literals:** Constants
* ***int*** above stands for **interrupt**.

**Topic – 3: Facts To Note**

**Addressing Data In Memory**

* Each **two digit** of a number is broken down into a **byte**.
* For example, **0725H** can be broken down into **07** (most significant byte) & **25** (least significant byte).
* But these bytes will be stored in **reverse order** at memory.



* When processor reads it from the memory, it is **reversed again** to be read in right order.
* **Absolute address:** Direct reference to a memory location.
* **Segment address/ offset:** Starting address of a memory address.

**Topic – 4: Basic Syntax**

**Sections**

* **Data section:** Used for declaring **global/static data** with section ***.data***.
* **Bss section:** Used for declaring **variables** with section ***.bss***.
* **Text section:** Used for **implementing code** with section ***.text***.

**Assembly Statements**

* Consists of three types – **Executable instructions**, **assembler directives** & **macros**.
* **Executable instruction:** Also simply known as **instruction**, tells processor what to do through **opcodes**. They generate machine codes.
* **Assembler directives:** Also known as ***pseudo-ops***, tells about some processor aspects & doesn’t produce machine codes.
* **Macros:** A text substitution mechanism.

**Assembly Syntax**

***[label] mnemonic [operands] [;comment]***

* Fields in square brackets are **optional**.
* **Operands** are also known as **parameters**.

**Assembly Instruction Example**

***inc count ; increments variable count***

**Topic – 5: Hello World Program**

***section .data***

***msg db "Hello, World!", 0xa ; 0xa is newline***

***len equ $ - msg ; length of constant above***

***section .text***

***global \_start***

***\_start:***

***mov edx, len***

***mov ecx, msg***

***mov ebx, 1 ; file descriptor (stdout)***

***mov eax, 4 ; calls sys\_write***

***int 0x80 ; calls kernel***

***mov eax, 1 ; calls sys\_exit***

***xor ebx, ebx ; Returns 0 (sets ebx to 0)***

***int 0x80 ; calls kernel***

* ***sys\_write*** tells the kernel that we are about to perform a **write operation**.
* **File descriptor** sends the **data** pointed by register ***edx*** of **length** in ***ecx*** to the **standard output** (**terminal** here).