**Chapter 1: BASIC CONCEPTS**

**Topic – 1: Welcome to Assembly Language**

**Introduction**

* Recommended to run in latest version of **Microsoft Visual Studio**.
* Check installation & setup steps at [www.asmirvine.com](http://www.asmirvine.com).
* We are going to learn **MASM** (macro assembler).

**Other Assembly Languages**

* **TASM** (turbo assembler) [For Windows]
* **NASM** [For Linux] {syntax similar to **MASM**}
* **MASM32** (a variant) [For Windows]
* **GAS** (GNU assembler) [For Linux]

**Fact!**

**🡪 Donald Knuth is known as father of asymptotic-notations.**

**Topic – 2: Questions You Might Ask**

**Common Terms**

* **Assembler:** Converts assembly codes to machine code.
* **Linker:** Combines all related assembly programs into single **executable** program.
* **Debugger:** Allows programmer to examine **registers** & **memories**.

**What programs can MASM create?**

* **32-Bit Protected Mode:** For 32-bit Windows versions.
* **64-Bit Modes**
* **16-Bit Real Address Mode:** For 32-bit Windows versions & embedded systems.

**Language Relevancies**

* Assembly code to machine code – **One-to-one mnemonics**
* High-level code to assembly/machine code – **One-to-many relation**
* **One-to-many** as **one** instruction in high-level language will unfold into **multiple** assembly instructions.

**Topic – 3: C++ to Assembly Conversion**

**C++**

***int Y;***

***int X = (Y + 4) \* 3;***

**MASM**

***mov eax, Y ; move Y to the EAX register***

***add eax, 4 ; add 4 to the EAX register***

***mov ebx, 3 ; move 3 to the EBX register***

***imul ebx ; multiply EAX by EBX***

***mov X, eax ; move EAX to X***

**Topic – 4: Assembly Languages Rules**

**Unrestricted Memory Use**

* Java **restricts** programmers from accessing certain memory addresses.
* To address this problem, **JNI** is imported for writing **C** code in that memory space.
* **JNI:** Java native interface
* Assembly on the other hand has **no restriction** on accessing any memory address.

**Assembly Language Applications**

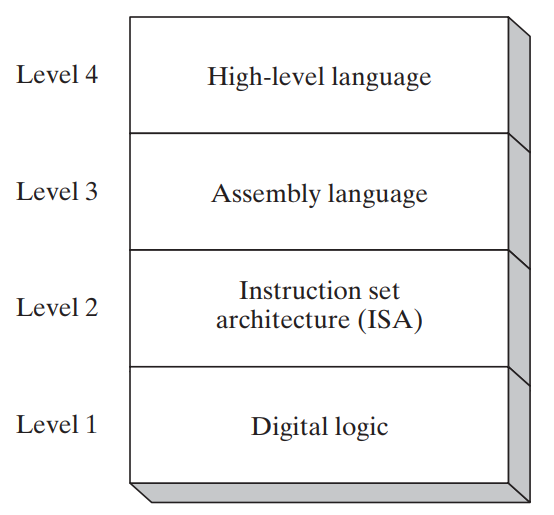
* Benefit of using **OOP languages** is that they can contain **millions** of lines of codes.

**Topic – 5: Virtual Machine Concept**

**Virtual Machines**

* Java uses **JRE** to convert its **Java** native code into **Java byte code**.
* And this **Java byte code** is at immediate lower level of abstraction to **Java** code.
* So, when executing, this **Java byte code** which works at lower level is executed by **JVM**.
* JVM is a virtual machine.
* Means user is seeing something, but there is something else happening in hardware.
* And this **JVM** implemented on various processor architecture is what makes Java **platform independent**.

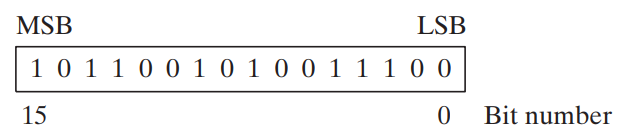
**Virtual Machine Levels**



* In **ISA (level-2)** the computer chip manufacturers embed **microprograms**.
* **Microprogram:** A program which executes **machine code**.
* **Microcode Interpreter:** Converts **machine code** into **digital circuit-level operations**.
* Assembly codes are easily translated to machine code.

**Topic – 6: Binary Integers**

**MSB & LSB**



* **Signed binary integer:** Positive or negative.
* **Unsigned binary integer:** Positive only (zero is also positive).
* Dots can be inserted between each **4-bits** or **8-bits** of a binary number, to make it more readable.

**Unsigned Binary Integers**

* **2n** = Possible number of values that can be formed of n-bits.
* Conversions involving binary values are actually **unsigned binary integers**.
* When converting decimal to binary, the **number of bits** after decimal point is **ceiling(log2x)**. Where **x** is given decimal value.

**Binary Addition**

**Suppose we add 1111 + 0001, now we have to consider storage area!**

**If storage can handle atleast 5 bits, then 1111 + 0001 = 1 0000**

**Otherwise, it is considered as 1111 + 0001 = 0000 [first 4-bits from right]**

**Topic – 7: Integer Storage Sizes**

**Size Measurement Units**

|  |  |
| --- | --- |
| **Unit** | **Size** |
| **Byte** | **1 byte** |
| **Word** | **2 bytes** |
| **Doubleword** | **4 bytes** |
| **Quadword** | **8 bytes** |
| **Double quadword** | **16 bytes** |

**Value range of any unit above = 2(Size in bits) – 1 [In decimal system]**

**Large Size Measurement Units**

|  |  |  |
| --- | --- | --- |
| **Unit** | **Comparison** | **In Bytes** |
| **Petabyte** | **1024 TB** | **250** |
| **Exabyte** | **1024 PB** | **260** |
| **Zettabyte** | **1024 EB** | **270** |
| **Yottabyte** | **1024 ZB** | **280** |

**Topic – 8: Hexadecimal Integers**

**Converting To Decimal**

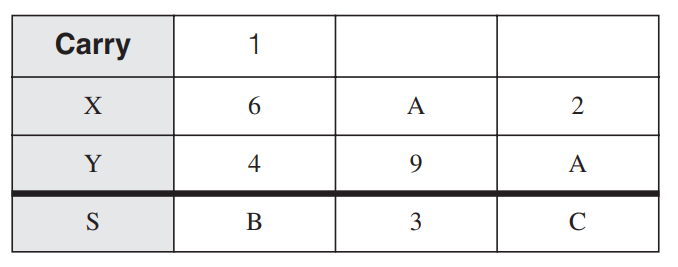
**Any system to decimal conversion:**

**(Dn-1 \* Bn-1) + (Dn-2 \* Bn-2) + … + (D1 \* B1) + (D2 \* B2)**

**Hexadecimal Addition**

* **Debuggers** usually display memory addresses in **hexadecimal** form.

**Hexadecimal Addition Example**



**When X’s A & Y’s 9 were added:**

**A + 9 = 19 [In decimal]**

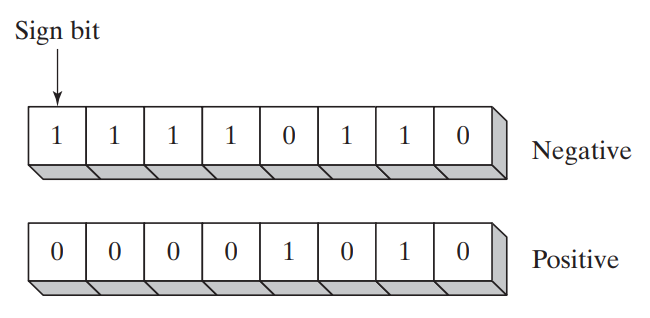
**So, 19 % B = 3 [B = base]**

**Hence, 3 is result & 1 is carried to next digit.**

**Topic – 9: Signed Binary Integers**

**Positive Binary v/s Negative Binary**

* In this, the **MSB** determines if value if positive or negative.



**2’s Complement**

* It is beneficial to processor designers as it can handle both addition & subtraction in the same digital circuit.
* **For example:** **A – B -> A + (-B)**
* **2’s complement is reversible** i.e., two numbers are each other’s 2’s complement.

**Binary 2’s complement formula:**

**STEP 1: Reverse bits (in pack of 8-bits)**

**STEP 2: Add 1 to it.**

**Hexadecimal 2’s complement formula:**

**STEP 1: Subtract each digit’s value from 15 i.e., reversing.**

**STEP 2: Add 1 to it.**

**Signed binary to decimal conversion:**

**If MSB is 0, directly convert it to decimal.**

**If MSB is 1, then find its 2’s complement & then convert it to decimal.**

**Decimal to signed binary conversion:**

**For positive integer, simply convert it to binary.**

**For negative integer, convert it to binary & then find its 2’s complement.**

* Similar steps for decimal to hex and vice versa conversions.
* For hex integers, if MSD is **<= 7** then its **positive**. Else it is **negative**.

**Binary subtraction (alternative method):**

**STEP 1: Keep the first binary integer as it is.**

**STEP 2: Find 2’s complement of the other one.**

**STEP 3: Now add both.**

**Topic – 10: Character Storage**

**Introduction**

* Characters are represented using certain integers (**ASCII code**).
* **ASCII:** American standard code for information interchange
* Character variables earlier used to of **1 byte** (8-bits).
* **IBM computers** still uses ASCII codes.
* **8-bits** because the ASCII code is of **8 digited binary number**.
* In which, **7-bits identify character** whereas **extra 1-bit** **is for proprietary** over characters.
* **Proprietary:** Reference to ownership to something.
* On IBM compatible microcomputers, values **128 to 255** represent **graphic symbol** & **Greek characters**.

**ANSI Character Set**

* **ANSI** defines character set consisting **256 characters**.
* **First 128** characters are **letters & symbols** on standard U.S. keyboard.
* The **other 128** characters are **international characters, currencies & fractions** etc.
* Early versions of **Windows** used ANSI characters set.

**Unicode Standard**

* It universally defines **characters** & **symbols**.
* It defines **code points**.
* **Code points:** Numeric codes.
* These include characters of major languages, even right-to-left Arabic script.

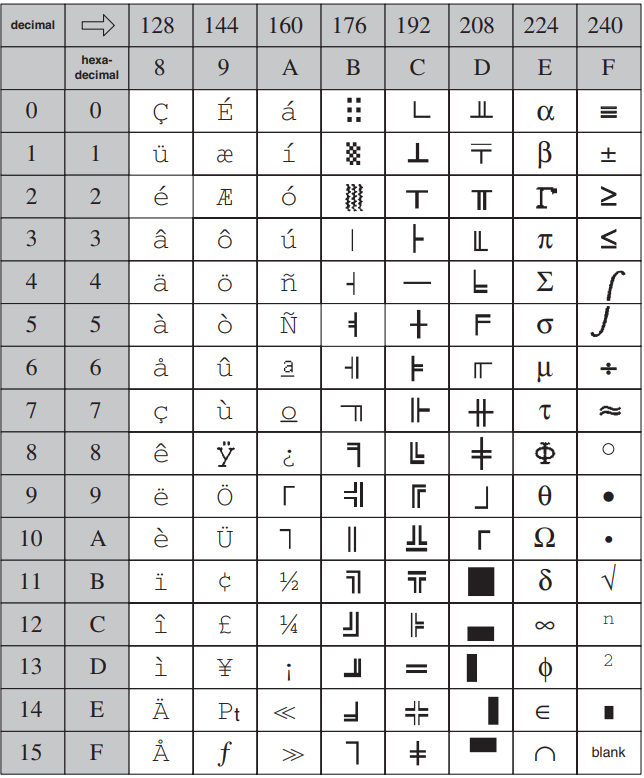
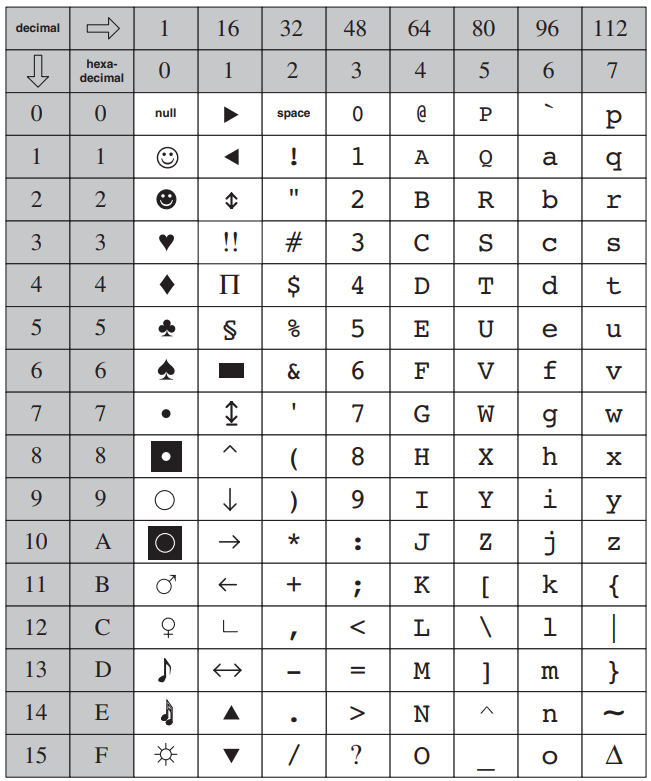
**Code To Graphic Representation Formats**

* **UTF-8:** Used in **HTML** where byte codes are same as **ASCII**.
* **UTF-16:** Makes **economical** use of storage, and each character is of **2-bytes**. Used by recent versions of **Windows**.
* **UTF-32:** Used where storage is **not** a concern & **width** **of characters are fixed**. Each character is of **4-bytes**.

**ASCII Strings**

* Numeric codes of chars are stored **contagiously** in memory.
* **Null terminated string:** String ending with a null character.
* **Null character:** 0 (1-bit)

**ASCII Table**



* In the tables above, **2nd row is the MSB** & **2nd column is LSB** of hex code.

**ASCII Control Characters**

|  |  |  |
| --- | --- | --- |
| **ASCII Code (Decimal)** | **Name** | **Description** |
| **8** | **Backspace** | **Moves one column to the left.** |
| **9** | **Horizontal tab** | **Skips forward n columns.** |
| **10** | **Line feed** | **Moves to next output line.** |
| **12** | **Form feed** | **Moves to next printer page.** |
| **13** | **Carriage return** | **Moves to leftmost output column.** |
| **27** | **Escape character** | **-** |

**Terminology For Numeric Data Representation**

* Remember that numbers & characters are stored in memory & displayed on screen in **different** ways.

**For example:**

**Decimal 65 is stored as 0100 0001 in memory.**

**A debugging program may think it is 41h.**

**And if it is copied to video memory then A (ASCII code 0100 0001) will be displayed on screen.**

* Binary integers are stored in memory in its **raw form** & in **multiple of 8-bits**.
* **Digit strings:** Strings which appear to be numbers, like **“185”** or **“97”**.

**Types Of Digit Strings**

|  |  |
| --- | --- |
| **Format** | **Value** |
| **Binary digit string** | **“01000001”** |
| **Decimal digit string** | **“65”** |
| **Hexadecimal digit string** | **“41”** |
| **Octal digit string** | **“101”** |

**Topic – 11: Boolean Expression**

**Introduction**

* Can be defined as operation on values **true** & **false**.
* Invented by **George Boole**.
* Remember all possible symbols for Boolean operators.

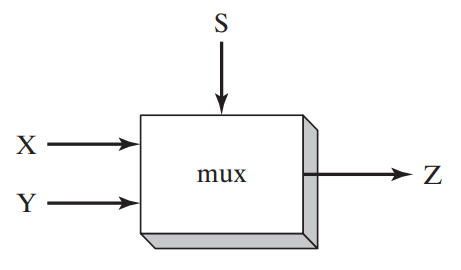
**AND**

* AND operation is carried out at **bit-level** (**bitwise**) in assembly.

**Operator Precedence**

* As per the brackets.
* If there are no brackets, move from **left to right**.

**Topic – 12: Truth Table for Boolean Functions**



* If S is **false**, then output Z will be **X**.
* Else if S is **true**, output Z will be **Y**.

**Number of S:**

**x = 2/log(n) [n is number of inputs]**

**Or simply, 2x = n [x is number of S]**