Lecture 1 - Introduction

Computer System Architecture

A computer system is basically a machine that simplifies complicated tasks.

It should maximize performance and reduce costs as well as power consumption.

The different components in the Computer System Architecture are:

- Input Unit
- Output Unit
- Storage Unit
- Arithmetic Logic Unit (ALU)
- Control Unit

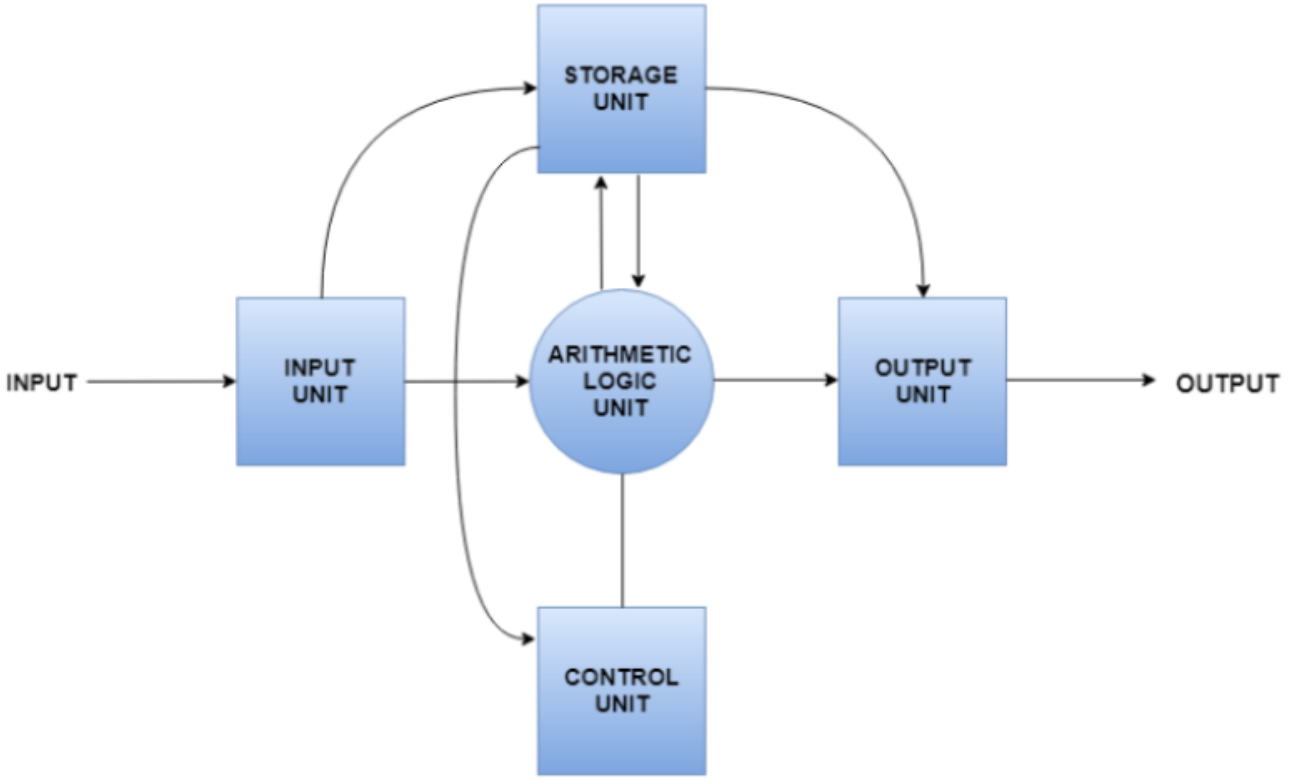
The process between components

The input data travels from input unit to ALU.

Similarly, the computed data travels from ALU to output unit.

The data constantly moves from storage unit to ALU and back again. This is because stored data is computed on before being stored again.

The control unit controls all the other units as well as their data.



Details about all the computer units (1)

- **Input Unit**

  The input unit provides data to the computer system from the outside. So, basically it links the external environment with the computer. It takes data from the input devices, converts it into machine language and then loads it into the computer system. Keyboard, mouse etc. are the most commonly used input devices.

- **Output Unit**

  The output unit provides the results of computer process to the users i.e it links the computer with the external environment. Most of the output data is the form of audio or video. The different output devices are monitors, printers, speakers, headphones etc.

- **Storage Unit**

  Storage unit contains many computer components that are used to store data. It is traditionally divided into primary storage and secondary storage.Primary storage is also known as the main memory and is the memory directly accessible by the CPU. Secondary or external storage is not directly accessible by the CPU. The data from secondary storage needs to be brought into the primary storage before the CPU can use it. Secondary storage contains a large amount of data permanently.

Details about all the computer units (2)

- **Arithmetic Logic Unit**
  - All the calculations related to the computer system are performed by the arithmetic logic unit.
  - It can perform operations like addition, subtraction, multiplication, division etc.
  - The control unit transfers data from storage unit to arithmetic logic unit when calculations need to be performed.
  - The arithmetic logic unit and the control unit together form the central processing unit.
- **Control Unit**
  - This unit controls all the other units of the computer system and so is known as its central nervous system.
  - It transfers data throughout the computer as required including from storage unit to central processing unit and vice versa.
  - The control unit also dictates how the memory, input output devices, arithmetic logic unit etc. should behave.

Computer architecture definition

Computer architecture can be defined as a set of rules and methods that describe the functionality, management and implementation of computers.

Role of Computer Architecture

The main role of Computer Architecture is to balance the performance, efficiency, cost and reliability of a computer system.

For Example – Instruction set architecture acts as a bridge between computer's software and hardware. It works as a programmer's view of a machine.

Computers can only understand binary language (i.e., 0, 1) and users understand high level language (i.e., if else, while, conditions, etc).

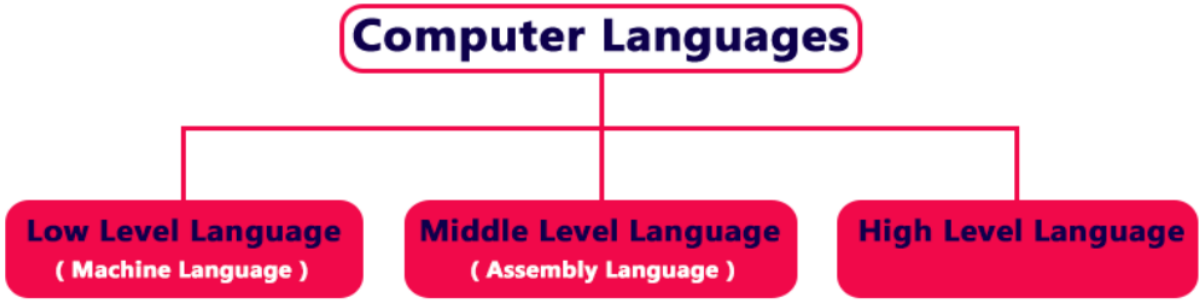
So to communicate between user and computer, Instruction set Architecture plays a major role here, translating high level language to binary language (machine code).

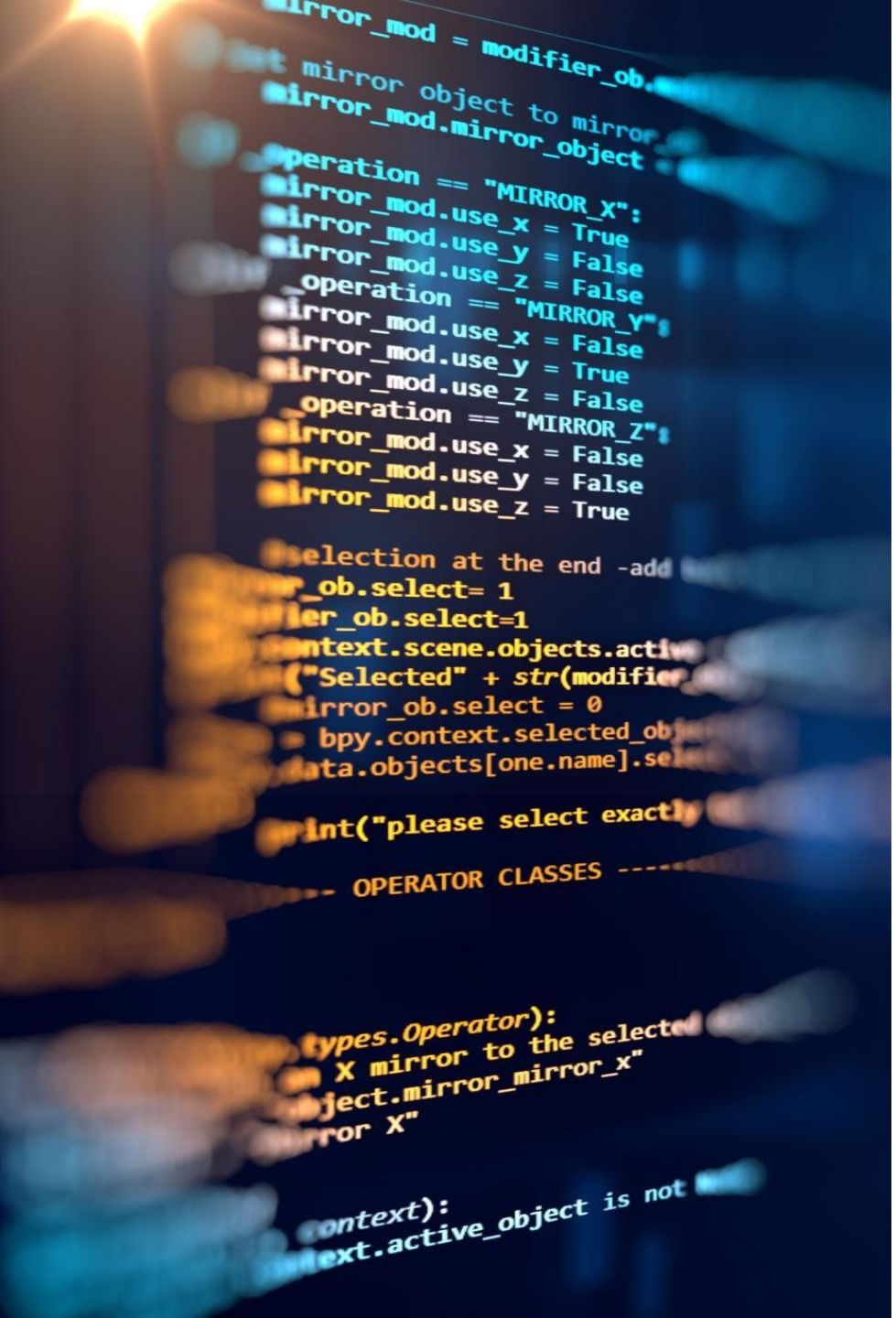
Machine Language (low level language)

- Machine code, also known as machine language, is the elemental language of computers. It is read by the computer's central processing unit (CPU), is composed of digital binary numbers and looks like a very long sequence of zeros and ones.
- Ultimately, the source code of every human-readable programming language must be translated to machine language by a compiler or an interpreter, because binary code is the only language that computer hardware can understand.
- Each CPU has its own specific machine language. The processor reads and handles instructions, which tell the CPU to perform a simple task.
- Depending upon the processor, a computer's instruction sets may all be the same length, or they may vary, depending upon the specific instruction.
- The architecture of the particular processor determines how instructions are patterned.
- The execution of instructions is controlled by firmware or the CPU's internal wiring.
- Human programmers rarely, if ever, deal directly with machine code anymore due to the complexity of the task.

Assembly Language (middle level language)

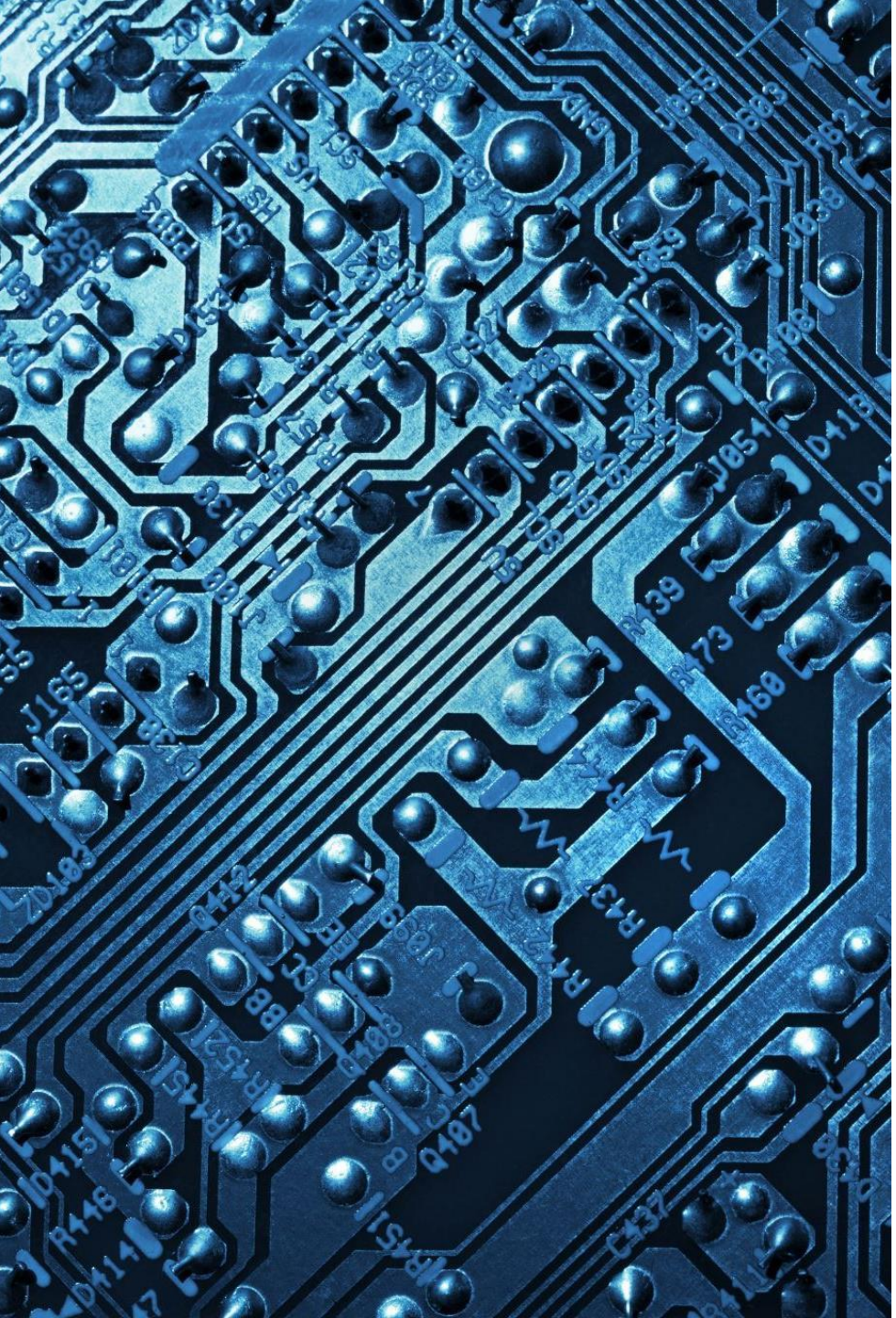
- **Middle-level** language is a computer language in which the instructions are created using symbols such as letters, digits and special characters.
- **Assembly language** is an example of middle-level language. In assembly language, we use predefined words called mnemonics. Binary code instructions in low-level language are replaced with mnemonics and operands in middle-level language. But the computer cannot understand mnemonics, so we use a translator called **Assembler** to translate mnemonics into machine language.
- Assembler is a translator which takes assembly code as input and produces machine code as output.
- That means, the computer cannot understand middle-level language, so it needs to be translated into a low-level language to make it understandable by the computer.
- Assembler is used to translate middle-level language into low-level language.





High Level Language

- High-level language is a computer language which can be understood by the users.
- The high-level language is very similar to human languages and has a set of grammar rules that are used to make instructions more easily.
- Every high-level language has a set of predefined words known as Keywords and a set of rules known as Syntax to create instructions.
- The high-level language is easier to understand for the users but the computer can not understand it. High-level language needs to be converted into the low-level language to make it understandable by the computer.
- We use **Compiler** or **interpreter** to convert high-level language to low-level language.
- Languages like C++, JAVA, Python, etc., are examples of high-level languages.
- All these programming languages use human-understandable language like English to write program instructions.
- These instructions are converted to low-level language by the compiler or interperter so that it can be understood by the computer.



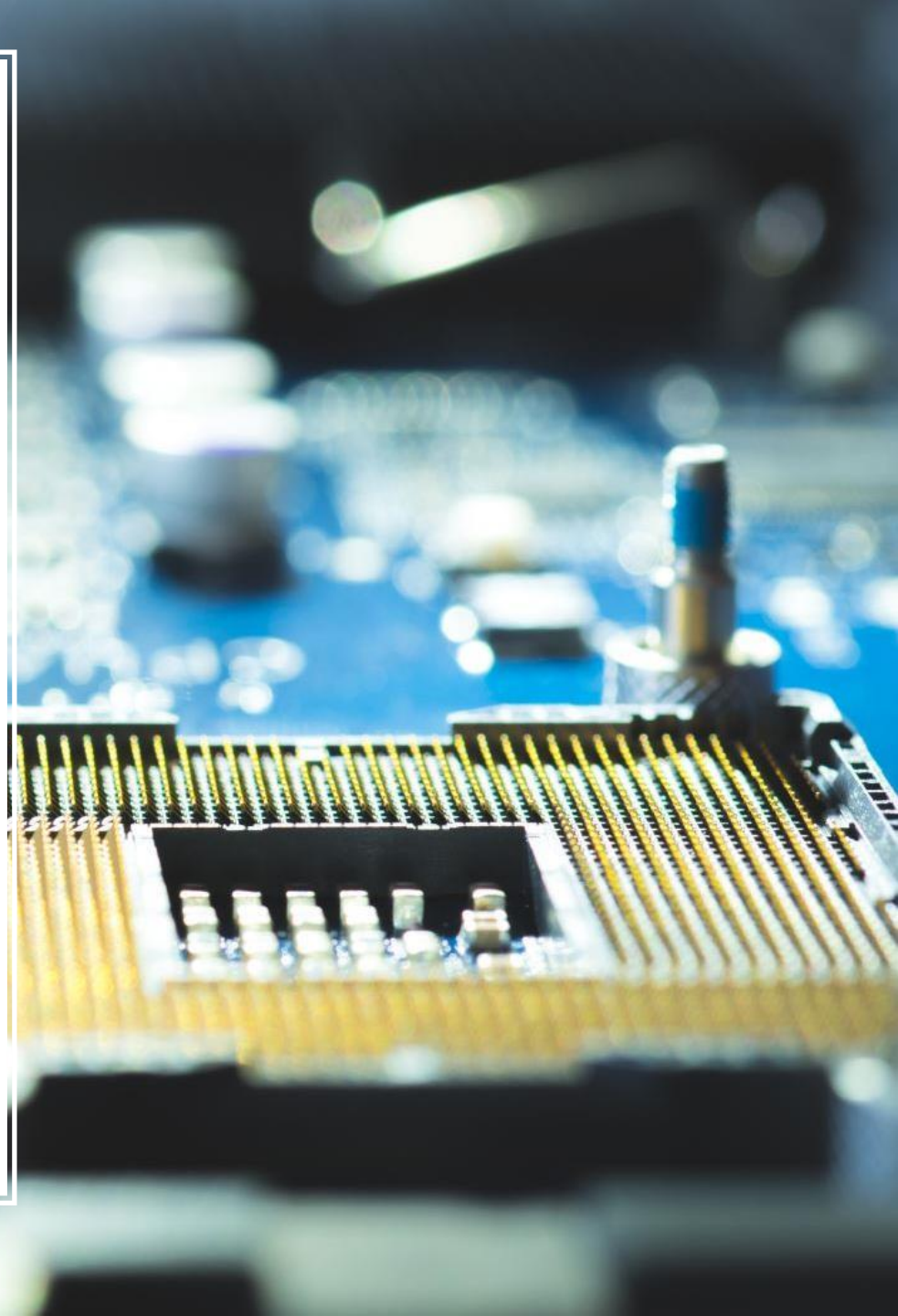
ASM Definition

Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems.

Assembly language is converted into executable machine code by a utility program referred to as an assembler like **NASM,** MASM, etc.

What is Assembly Language?

- Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.
- Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs.
- These set of instructions are called 'machine language instructions'.
- A processor understands only machine language instructions, which are strings of 1's and 0's.
- However, machine language is too complex for using in software development.
- So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.



Why ASM is necessary?

- ASM:
  - allow us to comunicate directly with hardaware using human readable texts.
  - is used for direct hardware manipulation and it gives us permision to acces specialized processor instrcuctions to adreess critical performance issues.
  - is used in writing device drivers, in system operating design and embeded systems programs

- **Where is ASM used?**
  - In aviation industry
  - In mallware detection programs
  - In security
  - In firmware aplications.

Binary Number System and the **Bit** Notion

The binary number system is a system of notation that uses base 2, instead of the base 10 (ten) largely used.

So apart from 0 and 1 which are the same in both base-2 and base-10 number notation systems, the 2 in base-10 is written as 0010, the 3 is 0011, the 4 is 0100 and so forth.

Perhaps not very intuitive for us humans used to the base-10 but really convenient for computer technology, because it is much easier and more reliable from a physical point of view to handle elements associated with just 2 states ("0" and "1", "on" or "off", etc) than it would be handling 10 states.

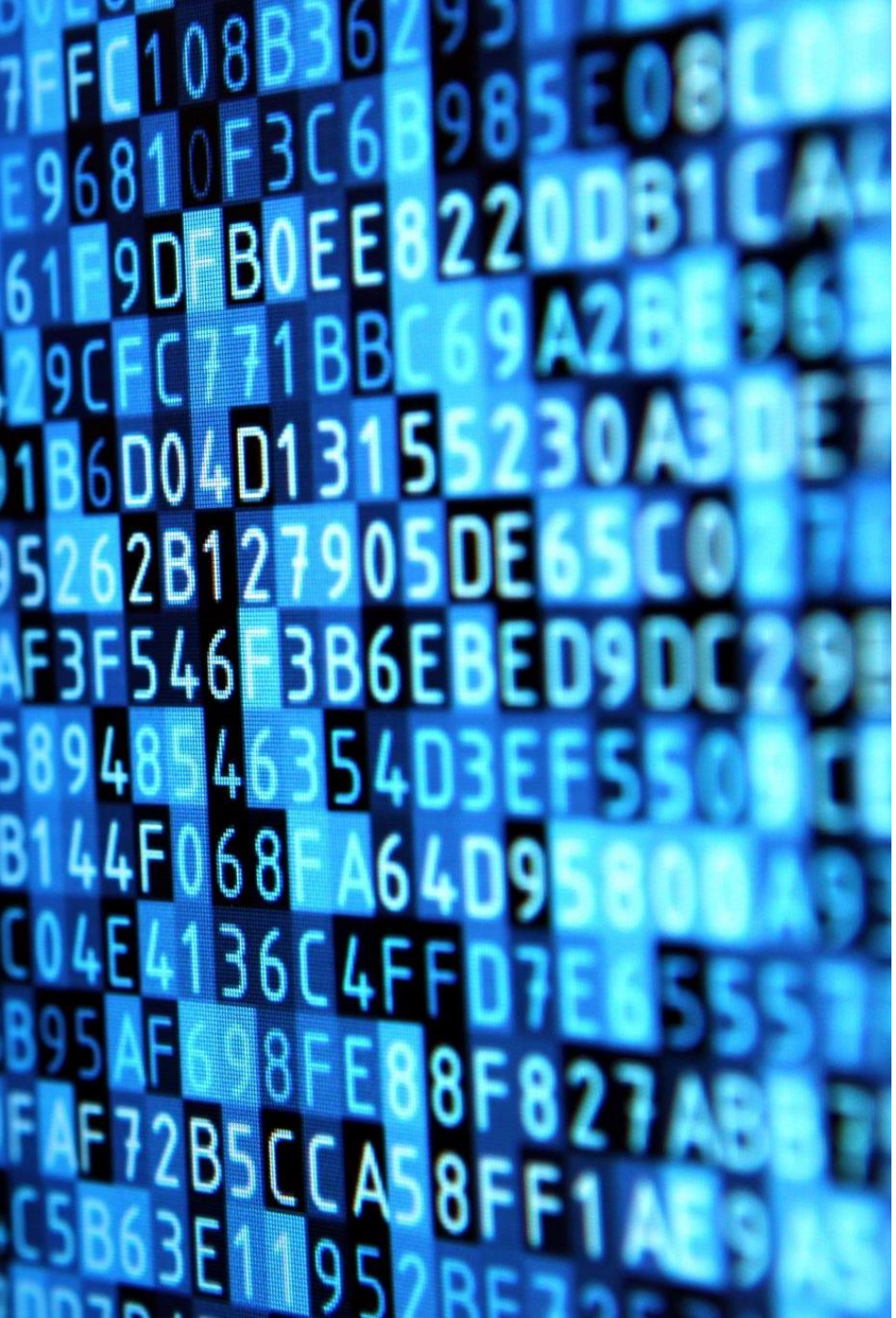
So computer hardware uses the "0" and "1" which is called a *bit* (from '**b**inary dig**it**') and bits are usually grouped in 8 bits, called a **byte**.

Base 10	Base 2
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Hexadecimal Number System

- Hexadecimal number system uses base 16.
- The digits in this system range from 0 to 15.
- By convention, the letters A through F is used to represent the hexadecimal digits corresponding to decimal values 10 through 15.
- Hexadecimal numbers in computing is used for abbreviating lengthy binary representations.
- Basically, hexadecimal number system represents a binary data by dividing each byte in half and expressing the value of each half-byte.
- The next table provides the decimal, binary, and hexadecimal equivalents.

Base 10	Base 16	Base 2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111



B2 -> B16 and B16 -> B2

To convert a binary number to its hexadecimal equivalent, break it into groups of 4 consecutive groups each, starting from the right, and write those groups over the corresponding digits of the hexadecimal number.

Example − Binary number 1000 1100 1101 0001 is equivalent to hexadecimal - 8CD1

To convert a hexadecimal number to binary, just write each hexadecimal digit into its 4-digit binary equivalent.

Example − Hexadecimal number FAD8 is equivalent to binary - 1111 1010 1101 1000

Nasm Assembler

Assembly language is dependent upon the instruction set and the architecture of the processor.

- We will use the NASM assembler, as it is:
  - Free. You can download it from various web sources.
  - Well documented and you will get lots of information.
  - Could be used on both Linux and Windows.
- The **Netwide Assembler** (**NASM**) is an assembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. It is considered one of the most popular assemblers

- ASM Elements

- Data types in ASM
  - user-defined variables
  - predefined variables

- Data representation in ASM (Little-Endian)
- Instructions
  - Instructions for unsigned numbers
    - mul - multiplication
    - div - division

ASM Elements

- Reserved words
- Identifiers
- Directives
- Instructions
- Operands

Reserved words

- Instruction names
- Register names
- Directives, which tell assembler how to assemble programs
- Attributes, which provide size and usage information for variables and operands
- Operators, used in constant expressions

The identifier

- A programmer-chosen name.
- It might identify a variable, a constant, a procedure, or a code label.
- The identifiers may contain between 1 and 247 characters.
- The first character must be a letter (A..Z, a..z)
- An identifier cannot be the same as an assembler reserved word.

The Directives

A directive is a command embedded in the source code that is recognized and acted upon by the assembler.

Directives can define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler.

Examples:

- Directives to define variables: DB, DW, DD, DQ
- Directive to define constants: EQU
- Directive to define variables without an initialization (only with a space allocation)

Examples:

zece EQU 10

The instruction

Is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime.

An instruction contains four basic parts:

1. Label (optional)
2. Instruction name (required)
3. Operand(s) (usually required)
4. Comment (optional)

This is the basic syntax:

[*label*:] *name* [*operands*] [;*comment*]

An 80x86 assembly language instruction general format	*INSTRUCTION_NAME destination, source* *destination – first operand and source – second operand* *Other forms:* *INSTRUCTION_NAME operand1, operand2* *INSTRUCTION_NAME operand* *INSTRUCTION_NAME*

The Operands

Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, constant expression,

Example	Operand Type
96	Constant (*immediate value*)
2 + 4	Constant expression
eax	Register
count	Memory

Comments

Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code.

The following information is typically included at the top of a program listing:

- Description of the program's purpose
- Names of persons who created and/or revised the program
- Program creation and revision dates
- Technical notes about the program's implementation

List of Commonly Used Abbreviations

reg	An 8, 16, or 32 bits general register from the following list: AH, AL, BH, BL, CH, CL, DH, DL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP
reg8, reg16, reg32	A register, identified by its number of bits
mem	A memory operand (a variable)
mem8, mem16, mem32	A memory operand, identified by its number of bits.
imm	An immediate operand (a constant)
imm8, imm16, imm32	An immediate operand (a constant), identified by its number of bits

Data types in ASM

Data types in ASM

All data used in an IA-32 assembly program is essentially numerical (integer numbers) and can have 4 basic types:

- **byte –** that data is represented on 8 bits
- **word** – that data is represented on 16 bits
- **doubleword** – that data is represented on 32 bits
- **quadword** – that data is represented on 64 bits

- In the assembly language we have data that does not changes its value throughout the execution of the program (i.e. **constant data** or constants) and data that does change its value throughout the execution of the program (i.e. **variable data** or variables).

Variables

- The IA-32 assembly language has 2 kinds of variables:
  - user-defined variables
  - pre-defined variables (registers)

- A variable has a name, a data type, a current value and a memory location (where the variable is stored).

  The value from a variable can be acces using [].

Examples: Initial values for a particular data type variables

Byte: a db 10

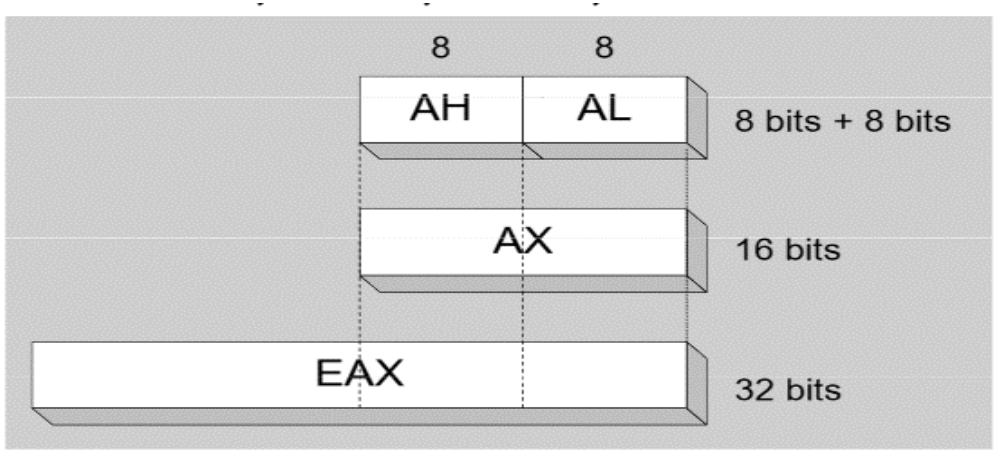
Word: b dw 12

Doubleword: c dd 23

Quadword: d dq 1002

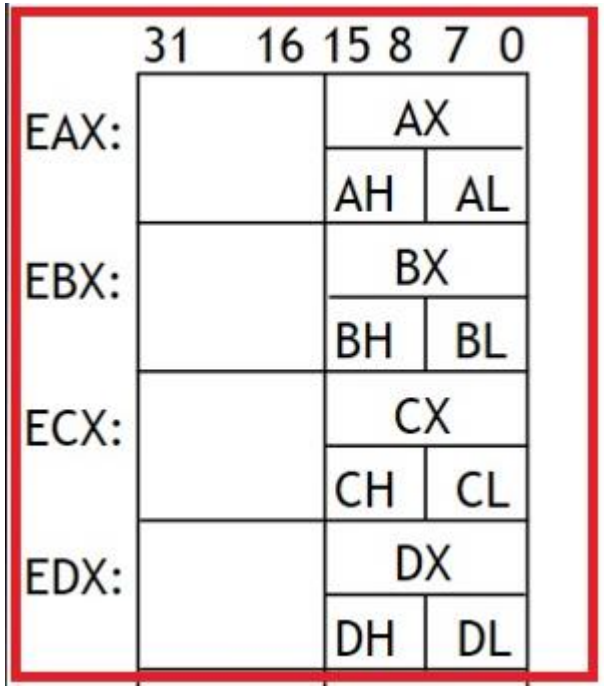
Registers = Predefined variable

Registers (Processor Registers) are small storage units (informally think of registers as variables that are built in the processor).
Using registers instead of memory to store values, makes the process faster and also more efficient.



32 Bits	16 Bits	8 Bits (HIGH)	8 Bits (LOW)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

General-purpose registers



- Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

Operations

- Arithmetic
- Data movement
- Logical

Doubleword registers (32 bits)

- EAX-Extended accumulator register
- EBX–Extended base register
- ECX-Extended counter register
- EDX–Extended data register

Data representation

Data Representation

The **smallest unit** of data in a computer is called **Bit** (Binary Digit) so the computer stores instructions and data in memory as collections of 0 and 1.

In Debugger (Olly Dbg) – only the **hexadecimal** values of the data is visible.

A collection of 8 bits form a **byte** (and is formed from 2 hexadecimal digits)

2 bytes create a **word (16 bits)** => 4 hexadecimal digits

4 bytes create a **doubleword (32 bits)** => 8 hexadecimal digits

8 bytes create a **quadword (64 bits)** => 16 hexadecimal digits

Data representation in memory: little-endian

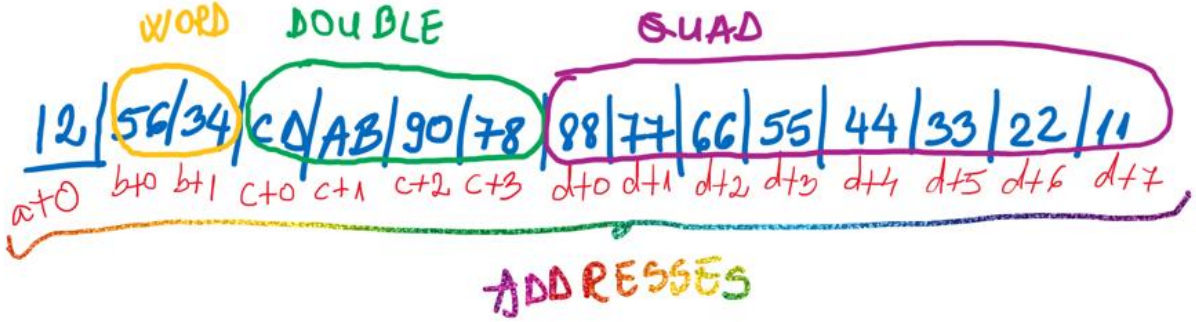
- Each data has an address
  - An *address* refers to a single location in memory, and x86 processors permit each byte location to have a separate address

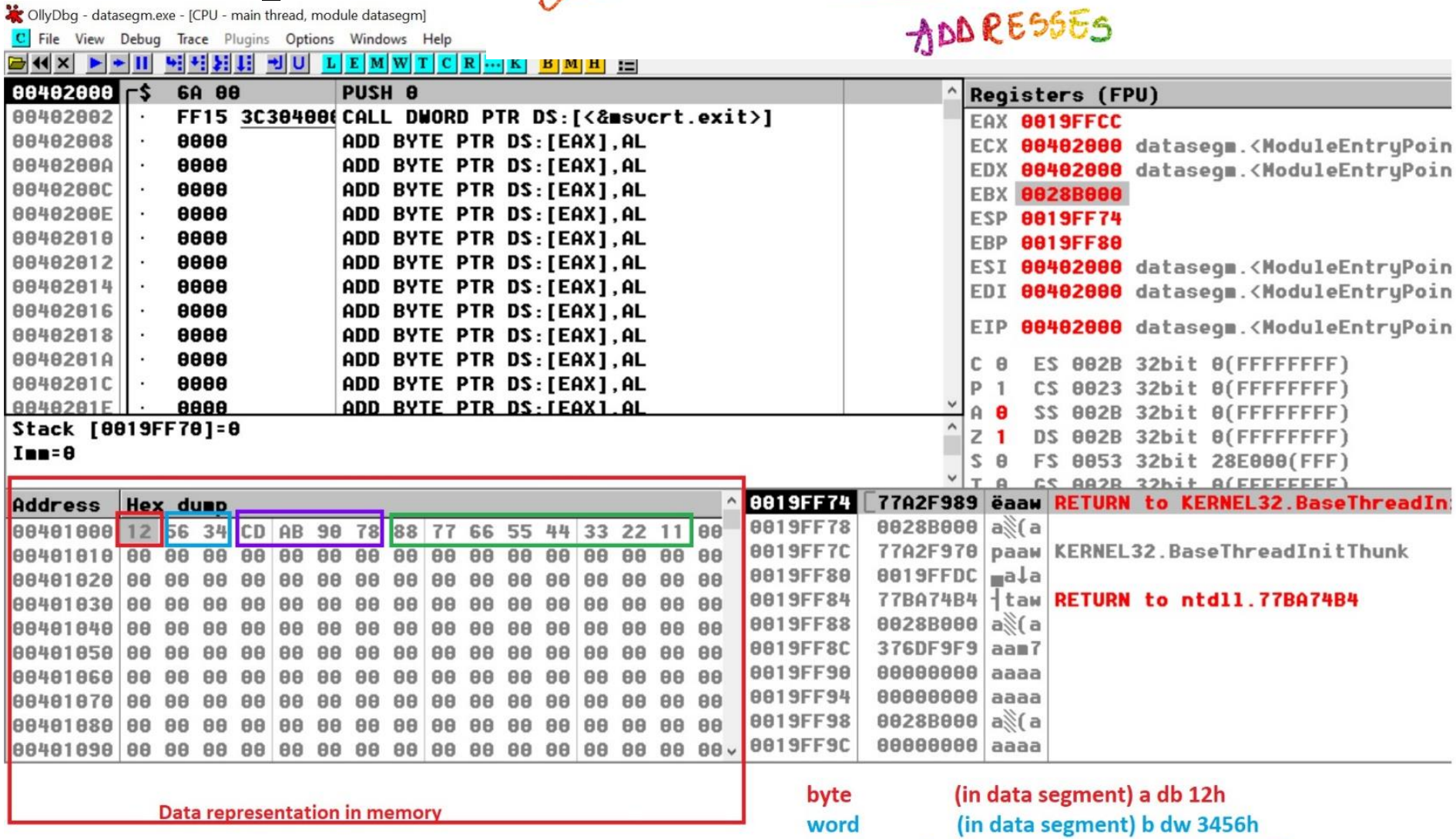
- Each data has a value:
  - x86 processors store and retrieve data values from memory using ***little endian*** **order (low to high byte from the intial configuration)**:
    - The least significant byte is stored at the first memory address allocated for the data.
    - The remaining bytes are stored in the next consecutive memory positions.

Little-endian example

```
;We have 4 variables
defined in data segment:

a db 12h
b dw 3456h
c dd 7890abcdh
d dq 1122334455667788h
```





byte (in data segment) a db 12h
word (in data segment) b dw 3456h
doubleword (in data segment) c dd 7890ABCDh
quadword (in data segment) d dq 1122334455667788h

INSTRUCTIONS

In an assembly program, the instructions are executed sequentially (one by one, in the order listed in the source code)

MOV – assignment instruction

Syntax: mov dest, source

(where dest and source are either registers, variables or constants of type byte, word or dword; dest can not be a constant)

Effect: dest := source

- The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.
- *Syntax*
  mov <reg>,<reg>
  mov <reg>,<mem>
  mov <mem>,<reg>
  mov <reg>,<const>
  mov <mem>,<const>
- *Examples*
  mov eax, ebx — copy the value in ebx into eax
- mov ax, 2 – copy the constant 2 in reg ax
- mov word[a], bx – copy the bx value into variable a value

Instructions: ADD – addition instruction

Syntax: add dest, source

Effect: dest := dest + source

Constraints:

- dest and source – same dimensions
- dest and source are either registers, variables or constants of type byte, word or dword;
- dest can not be a constant
- at least one operand should be a register or a constant

Examples:

add bx, cx

add [a], 101b

Add [a], al

add cl, [a]

add [a], [b] – is not correct – because at least one operand should be a register or a constant

Instructions: SUB – substraction instruction

Syntax: **sub dest, source**

Effect: dest := dest – source

Constraints:

- dest and source – same dimensions
- dest and source are either registers, variables or constants of type byte, word or dword;
- dest can not be a constant
- at least one operand should be a register or a constant

Examples:

sub bx, cx

sub [a], 101b

sub cl, [a]

sub [a], [b] – is not correct – because at least one operand should be a register or a constant

!!!

Dimensions	8 bits	8 bits	16 bits	32 bits	64 bits
Data Type	byte	byte	word	doubleword	quadword
Number of Hexadecimal digits	2	2	4	8	16
Registers	AH	AL	AX	EAX	EDX:EAX
	BH	BL	BX	EBX	
	CH	CL	CX	ECX	ECX:EBX
	DH	DL	DX	EDX	

Base 10	Base 16	Base 2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111