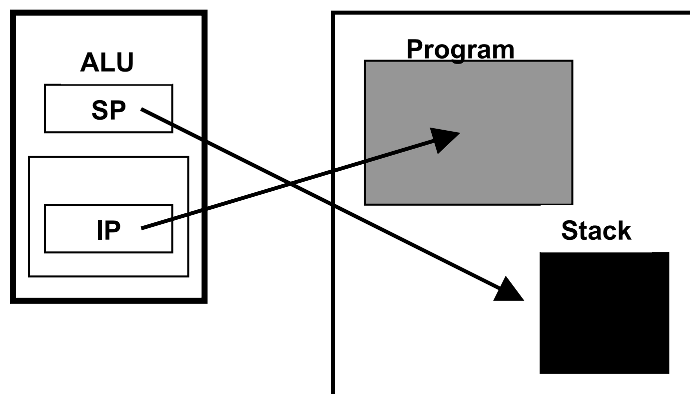
CPU

The **CPU** is usually organised in three parts: The **Arithmetic and Logic Unit** or **ALU** (which performs computations and comparisons) and the **Control Unit** which fetches information from and stores information into the **Memory Unit**.

Every **CPU** has at least up to a dozen **registers** which are special, small, very fast memory.
Registers that can be used for any type of arithmetic, as well as for addresses (pointers, subscripts, etc…) are called **general-purpose Registers** e.g. **AX, BX, CX** and **DX**.
Two registers of invaluable importance are the **Stack Pointer (SP)** in the **ALU** and the **Instruction Pointer (IP)** also referred as the **Program Counter (PC)** in the **Control Unit**.



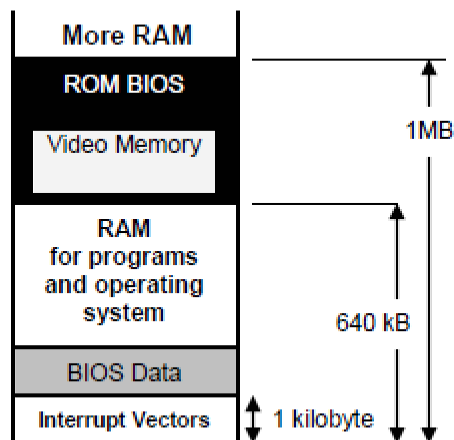
The **Instruction Pointer** contains the **address** in memory of the next instruction to be executed by the CPU. One of the main functions of the **Control Unit** is to perform an infinite loop of fetching the instruction at the **IP**, updating the **IP** to point to the next instruction and then executing the fetched instruction. This is called the **fetch-execute cycle**.

The **stack data structure** is a block of memory locations, and the **Stack Pointer** register in the **ALU** contains the address of one of them, called the **'top'** of the stack.

IBM PC and compatible Memory Structure

Any IBM PC makes special use of parts of its first **1 MB** address space.
Any Intel 80X86 chip uses the first **1024 bytes** (**1 kB**) for **interrupt vectors**.
The PC itself makes special use of memory in locations above the first **640 kB.**
All IBM PCs start up in **real** mode in which only the first **1 MB of RAM** is usable.

All modern operating systems immediately change to **protected mode** where **all the memory** becomes available.



Memory Addressing

Two types or memory addressing: **16-bit** and **32-bit**

- **16-bit addressing:**

The original IBM PC (80286) used this mode. However with 16-bit we can only reference 2^{16} i.e. 65,536 bytes, 64KB of memory.
So, how can we get full 20-bit address to be able to access the first 1MB of memory?

Use segment +some offset … *More later*

- **32-bit addressing**

Used by the 80386 and later. This allowed us a potential 2^{32} i.e 4 GB of memory
Modern Pentium processors can address even up to 64 GB of memory. Therefore, 32-bit addressing seems like bliss as it breaks the 1 MB barrier but …

Issues:
Backwards compatibility: 386 and later must still run programs requiring 16-bit addressing.
Memory protection: There must be a scheme to prevent programs from clobbering each other.

Number Systems

- **Denary (decimal) number system**
  This number system which is usually used by humans. Positional number system: the furthest right number has a weight of 10^0 e.g. decimal number 46,536 is
  $$4 \times 10^4 + 6 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 6 \times 10^0$$

- **Binary number system**
  This number system which is usually used by computers. Positional number system: the furthest right number has a weight of 2^0 e.g. binary number 1011010B is
  $$1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 64 + 16 + 8 + 2 = 90_{10}$$

Conversions:
Binary to denary: as above
Denary to binary: Reverse of Horner's method: repeated division by 2, sequence of remainders taken in reverse order.

- **Hexadecimal number system**
  Binary becomes too cumbersome for e.g. 1 million in denary is 20 binary digits
  Hence, hexadecimal (base 16) is used. Its digit starts from 0-9 and A-F.

Conversions:
Hex to Bin: each hex digit corresponds to exactly four binary digits.
Bin to Hex: Group of four bits taken from the right and padding with zero where needed.
Dec to Hex: divide repeatedly by 16, sequence of remainders taken in reverse order.
Hex to Dec: As denary, replacing base with 16.

Assembler Overview

In this section will show you how to **write, assemble, link and execute** a program that displays your name on the screen.

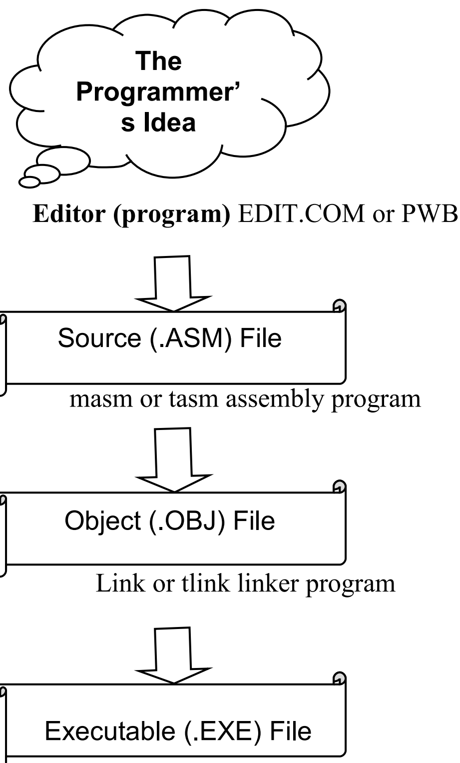
As we will be using a **Command Prompt** window and hence **using virtual 8086** mode for all our programs, the naming convention for filenames must be of the **8.3 format**.
Anyway, even if MS Windows allow us **256-character** long filenames, the trademark of a good programmer is still 8.3 format filenames.
No blank space allowed in the filename. Use underscores (another trademark of an IT literate person!!!)

The **3 types of files** we will be working with are indicated by the following extensions:

.ASM Assembly language source programs that you write.
.OBJ Object files; the output produced by the assembler from an input **.ASM** file.
.EXE Executable files produced by the linker utility combining one or more **.OBJ** files.

Registers Discussed

EAX	ah AX
EBX	bh BX
ECX	ch CX
EDX	dh AX
ESI	SI
EDI	DI
EBP	BP
	SP
	DS
	ES
	FS
	GS
	SS
	CS
	IP
	o d i t s z



;; FIRST.ASM – Our first Assembly Language Program. This program
;; displays the line " Hello, my name is nefertum" on the screen

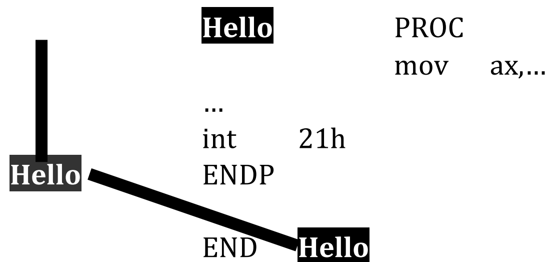
```
                .MODELSMALL
                .586            ; allows Pentium instructions. Must come after .MODEL

                .STACK          100h
                .DATA
Message                 DB              'Hello, my name is nefertum', 13, 10, '$'

                .CODE
Hello           PROC
                mov             ax, @data
                mov             ds, ax
                mov             dx, OFFSET Message
                mov             ah, 9h          ; Function code for Display String
                int             21h             ; The standard way to call the OS
                mov             al, 0           ; Return code for 0
                mov             ah, 4ch         ; Function code for Exit to OS
                int             21h
Hello           ENDP
                END             Hello           ; tells where to start execution
```

Synopsis

- Upper & lower case are not distinguished in Assembly Language (except within quotes).
- The name chosen for the **PROC**edure, **Hello** in *first.asm* is unimportant, but the three occurrence of **Hello** must match exactly.



- The **PROC** name **Hello** is independent from the source filename *first.asm*
- Anything following a semicolon on a line is a **comment** and is ignored by the assembler.

General structure of an IBM PC Assembly Language

Label	OPERATION	OPERANDS	;comments

LABEL

Labels are **optional** and are example of *symbolic constants*, which are strings of characters which the programmer invents. They can be up to 31 characters and can consists of alphanumeric and special characters like '@', '_', '$' and '?' but must not start by a number (to distinguish symbols from numbers).

OPERATION

2 types: Actual (executable) single machine instructions, such as ***mov***
Non-executable instructions to the assembler, such as ***.DATA***, ***DB***

The second type of operation is called a *pseudo-operation.*
Both operations are specified using a symbolic code called the **OPERATION CODE** (OPCODE)

Pseudo-operations are usually shown in ***upper case*** to distinguish them from executable machine instructions.

OPERANDS

Each *op-code* can have zero or more operands. **Real** machine instructions usually have one or two.

Global Program Structure

```
;; FIRST.ASM – Our first Assembly Language Program. This program
;;                    displays the line " Hello, my name is nefertum" on the screen

             .MODEL      SMALL
             .586        ; allows Pentium instructions. Must come after .MODEL

                                                           Stack Segment
             .STACK      100h

             .DATA                                         Data Segment
Message      DB          'Hello, my name is nefertum', 13, 10, '$'

             .CODE
Hello        PROC                                          Code Segment
             mov         ax, @data
             mov         ds, ax
             mov         dx, OFFSET Message
             mov         ah, 9h        ; Function code for Display String
             int         21h           ; The standard way to call the OS
             mov         al, 0         ; Return code for 0
             mov         ah, 4ch       ; Function code for Exit to OS
             int         21h
Hello        ENDP

             END         Hello         ; tells where to start execution
```

The **Stack Segment** is used to reserve space for the **stack**. All programs must use a **stack**.
The **Data Segment** is used to declare variables and data storage, with/without initialization.
The **Code Segment** is used for executable program code.

Synopsis

.MODEL
This defines which **memory model** to use. Will be discussed later. Assume **SMALL** for now.

.586
This allows **Pentium instructions**; none are really used in this program. Will be discussed later.

.STACK Segment

The .STACK segment is declared with the statement

.STACK n

where **n (which must be even)** is the number of bytes reserved for the stack. We will use n = 100h i.e. 256_{10} which should be enough for our purposes.

.DATA Segment

The .DATA segment is used to define variables, with or without initial values. In our first, program we have used:

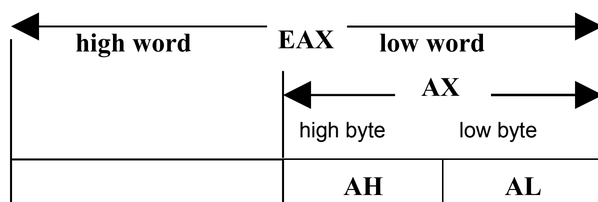
Message DB 'Hello, my name is nefertum', 13, 10, '$'

uses **DB** (*define byte*) pseudo-operation to define a string of byte, which is the message to be displayed on the screen. Three commonly used data types are ***byte, word*** and ***double word***. These are declared respectively by **DB**, **DW** and **DD**. These pseudo-operations also allow us to set initial values for the variables. E.g.

```
Unlucky     DB    13
Minus       DW    -299
Million     DD    1000000
```

defines *Unlucky* to be a byte (8-bit) variable with an initial value of 13, *Minus* to be a word (16-bit) variable with an initial value of -299 and *Million* to be a double word (32-bit) variable with an initial value of 1000000. Trying to initialize a value outside these ranges will cause an assembler error.

The **80X86 ALU** (Arithmetic and Logic Unit) has several registers, among which are four important registers used for computations. In the **286**, they were limited to **16 bits** and called **AX**, **BX**, **CX** and **DX** registers. From the **386** and later , there were extended to double words of **32 bits**, **EAX**, **EBX**, **ECX** and **EDX** respectively. The format of the **EAX** register is as follows:



The other registers are similar. "low" and "high" refer to the value they contain- the bits in the high byte represent larger value than those in the low byte (**Big-Endian**). There is no name for the high (left) word and there is no way to access it directly.

The **Stack Pointer SP** and the **Instruction Pointer IP** are **16 bit registers** that are **addresses in memory** and also have their extended (32-bit mode) counterparts' *viz.* **ESP** and **EIP**

The **executable** part (**.CODE** segment) of the **Hello** program does **three** things:

```
mov     ax, @data        Standard code
mov     ds, ax
```

```
mov     dx, OFFSET Message
mov     ah, 9h                 DOS Display String Call
int     21h
```

```
mov     al, 0
mov     ah, 4ch                DOS Return to DOS Call
int     21h
```

Two instructions are used: the ***mov*** instruction which moves data from place to place and the ***int 21h*** instruction which causes the operating system to perform various functions. Most 80X86 **opcodes** are written in three letters.

❖ The ***mov*** instruction
Syntax:

mov *destination, source* ;destination = source

mov is really a copy instruction since the previous content of the source is unchanged. Thus if **ax** = 1234 and **bx** = 9876 and the instruction **mov ax, bx** is executed, we will have both **ax** and bx equal to 9876.

There are various possibilities for the source and destination operands:

mov *memory or register, memory or register or constant*

which is abbreviated as:

mov *mem / reg , mem / reg /* constant

Note: ***mov*** copies FROM right TO left-hand operand.

GOLDEN RULES

1. You cannot move MEMORY to MEMORY directly
2. You cannot move TO a CONSTANT
3. You cannot move operands of DISSIMILAR SIZES

E.g.

```
mov   ax, A          ;ok
mov   A, ax          ;ok
mov   ax, bx         ;ok
mov   A, 345         ;ok
mov   ax, 345        ;ok
mov   A, B           ;illegal – memory to memory
mov   345, A         ;illegal – can't move TO constant
mov   ax, 24         ;ok
mov   al, 24         ;ok
mov   eax, 24        ;ok
mov   al, 345        ;illegal – word to byte move
mov   ax, 100000     ;illegal – double to word move
mov   ax, al         ;illegal – byte to word move
```

- The **int 21h** instruction

Operating system functions on the IBM PC are called DOS calls, after the original operating system MS-DOS. Using DOS is much like using standard procedures like puts(), printf(), cout in C/C++.

DOS Call Syntax:

```
; Load parameters to DOS Call into registers
mov     ah, function code
int     21h      ;call DOS and  returned values (if any) are in registers
```

Each DOS call has its own **function code numbe**r (in Hex) and pattern of registers used for sending and receiving values.

- **The DOS Display String Call**

The DOS call with function code 9h is used to display a string of characters on the screen. The 9h function displays a string of ASCII characters whose address is in the **dx** register, *up to but not including the first '$' sign encountered.* The form of the call is

```
mov     dx, OFFSET Message
mov     ah, 9h
int     21h
```

The first **mov** instruction is used to load the address of the ASCII string into **dx**. The line in the data region:

```
Message     DB     "Hello, my name is nefertum", 13, 10, "$"
```

is used to define the string to be displayed. The numbers **13** and **10** are the ASCII values for **Carriage Return** and **Line Feed**, respectively.

- **The Exit to DOS Call**

An explicit return to the OS is needed because the computer will continue trying to execute whatever garbage lies in memory **beyond the end of your program**. To end execution of your program, you must execute the return to DOS call, which is:

```
mov     al, 0h          ; return code
mov     ah, 4ch         ; Exit to DOS function code
int     21h
```

The two **mov** instructions could also have been written as a single instruction:

```
mov     ax, 4c00h
```

DOS call **4ch** has one parameter, the **return code** in the **ah** register. The return code is traditionally zero for a normal return, nonzero for an error return. This call is analogous to the C/C++ **exit()** function and the return statement in the **main()** function that returns a value.

Down Memory Lane

A **byte** can contain any unsigned number from **0** to **255** or any signed number from **-128** to **127** (signed numbers **-128** to **-1** corresponds to the unsigned numbers **128** to **255** in a byte)

You can specify a sequence of values as such:

```
Abyte   DB   12, 99, 20
```

defines 3 consecutive variables with initial values 12, 99 and 20 respectively. *Abyte* is the name of the *first variable* whose value is 12. The byte containing 99 is an 'anonymous' variable which can be referred as *[Abyte+1]* and the variable containing value 20 as *[Abyte+2]*

These 2 code fragments are equivalent:

```
Abyte   DB   12
        DB   99
        DB   20
```

Or

```
Abyte   DB   12, 99
        DB   20
```

Uninitialized Variables

```
ByteVar     DB   ?   ;ByteVar is a byte with NO initial value
WordVar     DW   ?   ;WordVar is a word with NO initial value
```

ASCII Variables

Bytes containing the ASCII representation of characters can be defined by enclosing the characters in paired single (') or double (") quotes:

```
Hi   DB   'H', 'e', 'l', 'l', 'o'
```

is equal to

```
Hi   DB   'Hello'
```

or

```
Hi   DB   "Hello"
```

Non-printable Characters

For non-printable characters, such as Carriage Return or Line Feed, (whose ASCII values are 13 and 10 respectively) the only way to get them with a DB instruction is to include them numerically:

```
Message         DB   'Hello, my name is nefertum', 13, 10, '$'
```

Creating Arrays

Data definitions can be repeated by using the DUP operation. E.g.

```
Ones        DB      100 DUP (1)
```

defines 100 consecutive bytes containing the number 1 (the first is called '**Ones**')

```
            DB      10 DUP (2, 3, ?)
```

defines 30 bytes which are consecutively 2, 3, ?, 2, 3, ?, …

DW

DW operates similarly except that it defines words of storage.

```
Aword       DW      1234h
```

We can represent the result of as occupying two consecutive bytes of memory as follows:

e.g.

12	34