# 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 like getting input from keyboard, displaying information on screen and performing various other jobs.
- These set of instructions are called 'machine language instructions'.
- Processor understands only machine language instructions, which are strings of 1's and 0's.
- However, machine language is too obscure and 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.

## Assembly Language

- The processor supports the following data sizes:
  - Word: a 2-byte data item
  - Doubleword: a 4-byte (32 bit) data item
  - Quadword: an 8-byte (64 bit) data item
  - Paragraph: a 16-byte (128 bit) area
  - Kilobyte: 1024 bytes
  - Megabyte: 1,048,576 bytes

# Assembly Language

- Binary, Octal, Decimal and Hexadecimal
- Conversion
- 1's Complement
- 2's Complement
- Binary operations

## Assembly Language

- Assemblers
  - NASM (Netwide Assembler)
  - MASM (Microsoft Assembler)
  - TASM (Borland Turbo Assembler)
  - GAS (GNU Assembler)
- On linux terminal type: *whereis nasm*
- For latest version refer: http://www.nasm.us/

## Assembly Language

- Sections: An assemble program can be divied into 3 sections:
  - The data section: .data
  - The bss section: .bss
  - The text section: .text

## Assembly Language

- **Data Section:** The data section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names or buffer size, etc., in this section.
  - Syntax: *section .data*
- **Bss section:** The bss section is used for declaring variables.
  - Syntax: *section .bss*

## Assembly Language

- **Text Section:** The text section is used for keeping the actual code. This section must begin with the declaration global _start, which tells the kernel where the program execution begins.
  - Syntax: *section .text*

    *global _start*

    *_start:*

## Assembly Language

- Comments: Assembly language comment begins with a semicolon (;). It may contain any printable character including blank.
- Assembly language programs consist of three types of statements:
  - Executable instructions or instructions
  - Assembler directives or pseudo-ops
  - Macros

## Assembly Language

- The **executable instructions** or simply instructions tell the processor what to do. Each instruction consists of an operation code (opcode). Each executable instruction generates one machine language instruction.
- The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.
- **Macros** are basically a text substitution mechanism

## Assembly Language

- Assembly language statements are entered one statement per line. Each statement follows the following format:

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

  The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.

## Assembly Language

- **Memory Segments:**
  - A segmented memory model divides the system memory into groups of independent segments referenced by pointers located in the segment registers.
  - Each segment is used to contain a specific type of data.
  - One segment is used to contain instruction codes, another segment stores the data elements, and a third segment keeps the program stack.

## Assembly Language

- **Data segment** - it is represented by .data section and the .bss. The .data section is used to declare the memory region, where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program. The .bss section is also a static memory section that contains buffers for data to be declared later in the program. This buffer memory is zero-filled.
- **Code segment -** it is represented by .text section. This defines an area in memory that stores the instruction codes. This is also a fixed area.
- **Stack** - this segment contains data values passed to functions and procedures within the program.
-

## Assembly Language

- To speed up the processor operations, the processor includes some internal memory storage locations, called **registers**.
- The registers store data elements for processing without having to access the memory. A limited number of registers are built into the processor chip.

# Assembly Language

- **Processor Registers:** There are ten 32-bit and six 16-bit processor registers in IA-32 (Intel Architecture, 32-bit) architecture. The registers are grouped into three categories:
  - General registers
  - Control registers
  - Segment registers
- The general registers are further divided into the following groups:
  - Data registers
  - Pointer registers
  - Index registers

# Assembly Language

- **Data Registers**
  - Four 32-bit data registers are used for arithmetic, logical and other operations. These 32-bit registers can be used in three ways:
  - As complete 32-bit data registers: EAX, EBX, ECX, EDX.
  - Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
  - Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.

# Assembly Language

- **AX** is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.
- **BX** is known as the base register as it could be used in indexed addressing.
- **CX** is known as the count register as the ECX, CX registers store the loop count in iterative operations.
- **DX** is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

# Assembly Language

- **Pointer Registers**
  - The pointer registers are 32-bit EIP, ESP and EBP registers and corresponding 16-bit right portions IP, SP and BP.
  - There are three categories of pointer registers:
    - Instruction Pointer (IP) - the 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
    - Stack Pointer (SP) - the 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
    - Base Pointer (BP) - the 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

## Assembly Language

- **Index Registers**
  - The 32-bit index registers ESI and EDI and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers:
    - Source Index (SI) - it is used as source index for string operations
    - Destination Index (DI) - it is used as destination index for string operations.

# Assembly Language

- **Control Registers**
  - The 32-bit instruction pointer register and 32-bit flags register combined are considered as the control registers.
  - Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

## Assembly Language

- Overflow Flag (OF): indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
- Direction Flag (DF): determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
- Interrupt Flag (IF): determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
- Trap Flag (TF): allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

# Assembly Language

- Sign Flag (SF): shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
- Zero Flag (ZF): indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
- Auxiliary Carry Flag (AF): contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
- Parity Flag (PF): indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
- Carry Flag (CF): contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

# Assembly Language

- **Segment Registers**
- Segments are specific areas defined in a program for containing data, code and stack. There are three main segments:
  - Code Segment: it contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
  - Data Segment: it contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
  - Stack Segment: it contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

# Assembly Language

```
section  .text
        global _start   ;must be declared for linker (ld)
_start:                 ;tells linker entry point
        mov  edx,len     ;message length
        mov  ecx,msg     ;message to write
        mov  ebx,1       ;file descriptor (stdout)
        mov  eax,4       ;system call number (sys_write)
        int    0x80      ;call kernel
        mov  eax,1       ;system call number (sys_exit)
        int    0x80      ;call kernel

section  .data
          msg db 'Hello, world!', 0xa  ;our dear string
          len equ $ - msg              ;length of our dear string
```

# Assembly Language

- Type the above code using a text editor and save it as hello.asm.
- Make sure that you are in the same directory as where you saved hello.asm.
- To assemble the program, type *nasm -f elf hello.asm*
- If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program named hello.o will be created.
- To link the object file and create an executable file named hello, type *ld -m elf_i386 -s -o hello hello.o*
- Execute the program by typing *./hello*
-

## References

- http://www.tutorialspoint.com/assembly_programming/