DAKOTA STATE UNIVERSITY

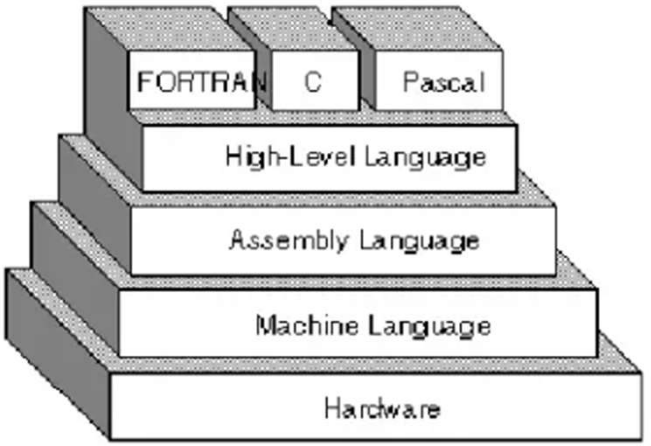
Assembly Language and Machine language (part 1)

Jihene Kaabi & Youssef Harrath

DAKOTA STATE UNIVERSITY

Programming Language

- The lowest-level language is called **Machine language**.
- A machine language is a coded set of instructions for a particular CPU, and it is also known **as machine code**.
- A machine language is designed to be used by a computer without the need of translation.



DAKOTA STATE UNIVERSITY

Machine Code

- Each type of CPU understands its own machine language.
- Instructions are numbers that are stored in bytes in memory.
- Each instruction has its unique numeric code, called the **opcode**.
- Instruction of x86 processors vary in size.
  - Some may be 1 byte, some may be 2 bytes, etc.
- Many instructions include operands as well.

- **E.g.:** On x86 there is an instruction to add the content of EAX to the content of EBX and to store the result back into EAX. This instruction is encoded (in hex) as: **03C3**
- Clearly, this is not easy to read/remember.

DAKOTA STATE UNIVERSITY

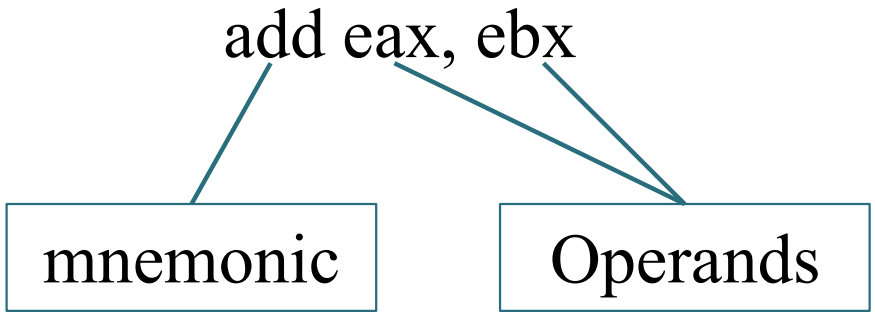
Assembly Language

- Assembly language is one level above the machine language. (Both Machine and Assembly language are considered as low-level language)
- It uses certain predefined symbolic codes instead of binary codes. These symbolic codes are called **mnemonics**.
- Assembly language programs are translated into machine language by a program called an **assembler**.

DAKOTA STATE UNIVERSITY

Assembly Code

- Each assembly instruction corresponds to exactly one machine instruction.
  - Not true of high-level programming languages
  - E.g.: a function call in C corresponds to many machine instructions
- The instruction on slide #3 (EAX = EAX + EBX) is written simply as:



DAKOTA STATE UNIVERSITY

Assembler

- An assembler translates assembly code into machine code.
- Assembly code is NOT portable across architectures.
- In this course we use the Netwide Assembler (NASM) assembler to write 32-bit Assembler.
- Note that different assemblers for the same processor may use slightly different syntaxes for the assembly code.

DAKOTA STATE UNIVERSITY

Comments

- Before we learn any assembly, it's important to know how to insert comments into a source file.
- Commenting an assembly code is necessary.
- With NASM, comments are added after a ';'
- Example:

```
ADD EAX, EBX    ; EAX = EAX + EBX
```

DAKOTA STATE UNIVERSITY

Assembly Language Syntax

- Assembly language is made up of two types of statements:
  - **Assembler Directives**: instructions that are directed to the assembler to do a specific thing.
  - **Executable Instruction**: one of the processor's valid instructions which can be translated into machine code form by the assembler.
- The syntax of an assembly language program statement obeys the following rules:
  - Only one statement is written per line.
  - Each statement is either an instruction or an assembler directive.
  - Each instruction has an opcode and possibly one, two or no operands at all.

DAKOTA STATE UNIVERSITY

Addressing mode

- Addressing mode refers to how the operands are specified.
- Operands can be in one of three places: in a register, in memory, or part of the instruction as a constant.
- Specifying a constant as an operand is called the ***immediate addressing mode***.
  - **e.g.**, ADD EAX , 4
- Similarly, specifying an operand that is in a register is called the ***register addressing mode***.
  - **e.g.**, ADD EAX, EBX
- All processors support these two addressing modes.

DAKOTA STATE UNIVERSITY

Register Addressing mode

- In this addressing mode, CPU registers contain the operands.
- For example, the instruction ***mov EAX, EBX*** requires two operands, and both are in the CPU registers.
  - The syntax of the move (mov) instruction is

    mov destination, source

- The mov instruction copies the contents of source to destination. The contents of source, however, are not destroyed.

DAKOTA STATE UNIVERSITY

Register Addressing mode (cont.)

- ***mov EAX, EBX*** copies the contents of the ebx register into the eax register.
- In this example, mov is operating on 32-bit data.
- However, we can also use the mov instruction to copy 16- and 8-bit data, as shown in the following examples:

```
mov   BX, CX
mov   AL, CL
```

DAKOTA STATE UNIVERSITY

Register Addressing mode (cont..)

- The register addressing mode is the most efficient way of specifying operands for two reasons:
  1. The operands are in the registers and no memory access is required.
  2. Instructions using the register mode tend to be shorter, as only 3 bits are needed to identify a register.
     - In contrast, we need at least 16 bits to identify a memory location.

DAKOTA STATE UNIVERSITY

Immediate Addressing mode

- Data are specified as part of the instruction.
  - As a result, even though the data are in memory, they are in the code segment, not in the data segment.
- This addressing mode is typically used to specify a constant, either directly or via the EQU directive.
- In the example

mov AL, 75

the source operand 75 is specified in the immediate addressing mode and the destination operand is specified in the register addressing mode. Such instructions are said to use mixed mode addressing.

DAKOTA STATE UNIVERSITY

Immediate Addressing mode (cont.)

- Immediate addressing mode is also faster because the operand is fetched into the instruction queue along with the instruction during the instruction fetch cycle.
- This prefetch, therefore, reduces the time required to get the operand from memory.

DAKOTA STATE UNIVERSITY

Direct Addressing mode

- Operands specified in a memory addressing mode require access to the main memory, usually to the data segment.
  - As a result, they tend to be slower than either of the last two addressing modes.
- Recall that to locate a data item in a data segment, we need to specify two components: the data segment base address and an offset value within the segment.
  - The offset is sometimes referred to as the *effective address*.
- The start address of the segment is typically found in the DS register.

DAKOTA STATE UNIVERSITY

Direct Addressing mode (cont.)

- The direct addressing can be used to access simple variables.
- **e.g.:** mov ax, [1000h]: loads a 2-byte object from the byte at address 4096 (0x1000 in hexadecimal) into ax.
- The main drawback of this addressing mode is that it is not useful for accessing complex data structures such as arrays and records that are used in high-level languages.

DAKOTA STATE UNIVERSITY

NASM Program Structure

```
; include directives

segment .data

    ; DX directives

segment .bss

    ; RESX directives

segment .text

    ; instructions
```

Include necessary files

Define initialized variables

Define uninitialized variables

DAKOTA STATE UNIVERSITY

Text Segment

- The text segment defines the asm_main symbol:

```
global  asm_main      ; makes the symbol visible

asm_main:             ; marks the beginning of
                        asm_main

    ; all instructions go here
```

DAKOTA STATE UNIVERSITY

NASM Program Skeleton

```
        ; include directives
segment .data
        ; DX directives
segment .bss
        ; RESX directives
segment .text
 global asm_main
 asm_main:
        ; instructions
```

DAKOTA STATE UNIVERSITY

More on the Text Segment

- Before and after running the instructions of your program, there is a need for some "setup" and "cleanup."
- We'll understand this later, but for now, let's just accept the fact that your text segment will always looks like this:

```
push ebp
mov ebp, esp
;
;Your Program here
;
mov eax, 0
mov esp, ebp
```

DAKOTA STATE UNIVERSITY

NASM Program Skeleton (cont.)

```
        ; include directives
segment .data
        ; DX directives
segment .bss
                ; RESX directives
segment .text
        global asm_main
        asm_main:
        push ebp
        mov ebp, esp
;Your Program here
        mov eax, 0
        mov esp, ebp
```

DAKOTA STATE UNIVERSITY

Our First Program

- Let's just write a program that adds two 4-byte integers and writes the result to memory.

- The two integers are initially in the .data segment, and the result will be written in the .bss segment.

DAKOTA STATE UNIVERSITY

Our First Program (cont.)

```
segment .data
   X   dd   15      ; first int
   Y   dd   6       ; second int
segment .bss
   result      resd  1       ; result
segment .text
   global asm_main
asm_main:
    push ebp
    mov ebp, esp
    mov eax, [X]          ; eax = int1
    add eax, [Y]          ; eax = int1 + int2
    mov [result], eax     ; result = int1 + int2
    mov eax, 0
    mov esp, ebp
```

DAKOTA STATE UNIVERSITY

Input/Output?

- The author of the course textbook provides an I/O package which comes as two files:
  - **asm_io.asm** (assembly code)
  - **asm_io.inc** (macro code)
- Simple to use:
  - Assemble asm_io.asm into asm_io.o.
  - Put "%include asm_io.inc" at the top of your assembly code.

DAKOTA STATE UNIVERSITY

Our First Program (cont.)

- If we want to print the result integer in addition to having it stored in memory.
  - We can use the print_int "macro" provided in asm_io.inc/asm.
  - This macro prints the content of the eax register, interpreted as an integer.
- We invoke print_int as: call print_int

DAKOTA STATE UNIVERSITY

Our First Program (cont..)

```
%include "/usr/local/share/csc314/asm_io.inc"
segment .data
     X   dd   15      ; first int
     Y   dd   6       ; second int
segment .bss
     result     resd  1       ; result
segment .text
      global asm_main
asm_main
   push ebp
   mov ebp, esp
   mov eax, [X]          ; eax = int1
   add eax, [Y]          ; eax = int1 + int2
   mov [result], eax     ; result = int1 + int2
   call print_int        ; print result
   mov eax, 0
   mov esp, ebp
```