Seminar 1

Introduction to the IA-32 assembly language (Intel Architecture)

Contents

- **The IA-32 is a 32-bit computing architecture (basically meaning that its main elements have 32 bits in size)** and it is based on the previous Intel 8086 computing architecture.
- it is an abstract model of a microprocessor specifying the microprocessor's elements, structure and instruction set.

1. Elements of the IA32 assembly language

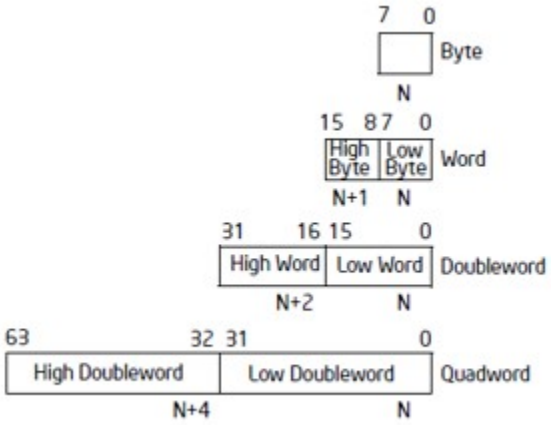
An **algorithm** is a sequence of steps/operations necessary to solve a specific problem.
For example, the algorithm for solving the 2^{nd} degree algebraic equation $a*x^2+b*x+c=0$ contains the steps:
1) compute the value of delta
2) if delta is greater or equal to zero, compute the solutions x1 and x2 using the well-known formulas.

An **algorithm** contains two things:
- a set of data/entities a, b, c, we need to compute x1 and x2
- and a sequence of operations/steps

When an algorithm is specified in a programming language, we refer to this algorithm as a **program**.

Throughout the semester we will study the IA-32 assembly language, starting with the **data part** of the assembly language and later the **operational part (instructions)**.

All data used in an IA-32 assembly program is **essentially numerical** (integer numbers) and can have different **types**:
- **Byte** – data is represented on 8 bits
  - **What is a bit?** The bit is **the smallest quantity of information** and it represents a **binary digit (0/1)**.
- **Word** – contains 2 bytes (data is represented on 16 bits)
- **Doubleword** – contains 4 bytes (data is represented on 32 bits)
- **Quadword** – contains 8 bytes (64 bits)



In the IA-32 assembly language we have:

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

1.1. Constants

We have 3 types of constants in the IA-32 assembly language:

- numbers (natural or integer):
  - written in base 2; ex.: 101b, 11100b
  - written in base 16; ex.: 34ABh, 0ABCDh
  - written in base 10; ex.: 20, -114
- character; ex.: 'a', 'B', 'c' .. – we use **apostrophe / single quote**
- string (sequence of characters); ex.: 'abcd', "test" …

Declaring a constant:

ten **EQU** 10

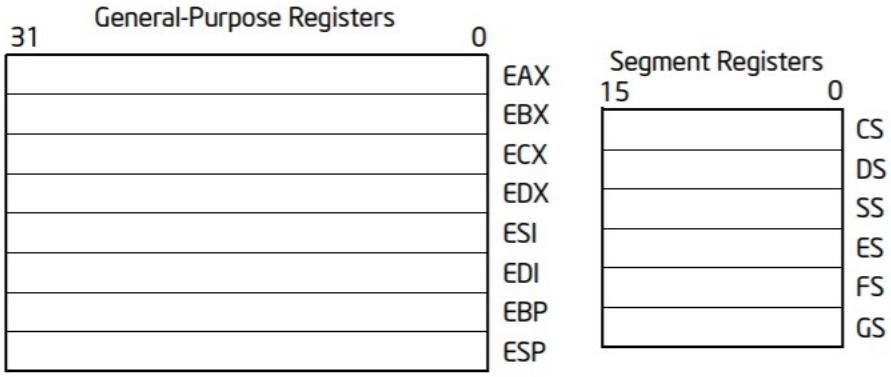
1.2. Variables

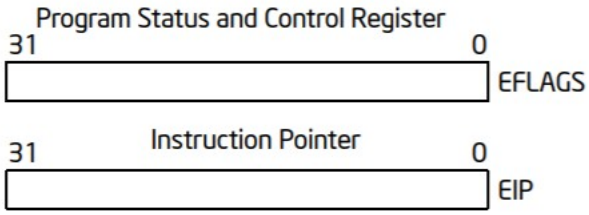
The IA-32 assembly language has 2 kinds of variables: pre-defined variables and user-defined variables. A variable has a **name**, a **data type** (*byte, word* or *doubleword*), a current **value** and a **memory location** (where the variable is stored).

Pre-defined variables (CPU registers):

The CPU registers are memory areas located on the CPU which are used for various computations. The IA-32 CPU registers are:

1) General registers (each register has 32 bits in size):





- **Obs**. **Every register may also be seen as the concatenation of two 16 bits subregisters. The upper register, which contains the most significant 16 bits of the 32 bits register, doesn't have a name and it isn't available separately**

- General-purpose registers are represented on a **doubleword,** but we can also use them as word or byte, for example:

EAX (32 de biti)			
Word high		Word low: **AX** (16 biti)	
Byte high din word high	Byte low din word high	Byte high din word low: **AH** (8 biti)	Byte low din word low: **AL** (8 biti)

! we cannot access the high word of EAX (similarly for EBX, ECX, etc.)

- EAX - **accumulator**. **Used by most instructions as one of their operands**
  - the lower or least significant part of EAX can be referred by AX
  - AX is formed by two 8-bit subregisters, AL and AH
- EBX - **base register**
  - the lower or least significant part of EBX can be referred by BX
  - BX is formed by two 8-bit subregisters, BL and BH
- ECX - **counter register**
  - the lower or least significant part of ECX can be referred by CX
  - CX is formed by two 8-bit subregisters, CL and CH)
- EDX - **data register**
  - the lower or least significant part of EDX can be referred by DX
  - DX is formed by two 8-bit subregisters, DL and DH
- ESP - **stack register**
  - the lower or least significant part of ESP can be referred by SP
- EBP - **stack register**
  - the lower or least significant part of EBP can be referred by BP
- EDI - **index register** (**Destination Index**)

    - the lower or least significant part of EDI can be referred by DI
    - usually used for accessing elements from bytes and words strings
- ESI - **index register** (**Source Index**)
    - the lower or least significant part of ESP can be referred by SI
    - usually used for accessing elements from bytes and words strings

2) Segment registers (each register has 16 bits – represented as word):

Every program is composed by one or more segments of one or more types. At any given moment during run time there is only at most one active segment of any type

Registers **CS (Code Segment)**, **DS (Data Segment), SS (Stack Segment)** and **ES (Extra Segment)** from BIU contain the values of the selectors of the active segments, correspondingly to every type. So registers CS, DS, SS and ES determine the *starting addresses* and *the dimensions of the 4 active segments*: code, data, stack and extra segments.

Registers FS and GS can store selectors pointing to other auxiliary segments without having predetermined meaning.

CS, DS, SS, ES, FS, GS – are not used in a program

3) Other registers (32 bit registers): ***EIP and Flags***.
Register EIP (which offers also the possibility of accessing its less significant word by referring to the IP subregister) contains **the offset of the current instruction inside the current code segment**, this register being managed exclusively by BIU.

User-defined variables:
For these variables, the programmer has to define the **name**, data **type** and initial **value**.
Examples:

- *a* ***DB*** *23* : defines the variable with the name "a", data type byte (DB-Define Byte) and initial value 23
- *a1* ***DW*** *23ABh* : defines the variable with the name "a1", data type word (DW-Define Word) and initial value 23ABh
- *a12* ***DD*** *-101* : defines the variable with the name "a12", data type doubleword (DD-Define DoubleWord) and initial value -101.\
- b **DQ** 10

Declaring variables with no initial value:

- a **RESB** 1 ; reserve 1 byte
- a **RESB** 64 ; reserve 64 bytes
- a **RESW** 1 ; reserve 1 word
- **RESD, RESQ**

1.3. Instructions

- Usually instructions have at most two operands and can be of the form INSTRUCTION destination, source

MOV 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

Restrictions: both operands need to be of the **same size**

Example:

```
a   dd  0
    mov ax, 3 ; ax = 3
    mov bx, ax ; bx=ax
    mov [a], eax
```

Arithmetic expressions

1. ADD dest, source ;

Effect: dest = dest + source

- dest and source are either registers, variables or constants of type **byte, word or dword**;
- **dest cannot be a constant**
- **at least one of the operands should be a register or a constant value (at most one operand can be a memory location)**
- both operands should have the same type: byte, word or doubleword

Examples:

```
add bx, cx
add [a], 101b
```

2. SUB dest, source

Effect: dest = dest - source

- dest and source are either registers, variables or constants of type byte, word or dword;
- dest cannot be a constant
- at least one of the operands should be a register or a constant value
- both operands should have the same type: byte or word

Examples:

```
sub ax, 2
sub [a], eax
```

2. Structure of a program

```
; Comments are preceded by the ';' sign. This line is a comment (is ignored by the assembler)

bits 32 ; assembling for the 32 bits architecture

global start ; we ask the assembler to give global visibility to the symbol called start
             ;(the start label will be the entry point in the program)
extern exit ; we inform the assembler that the exit symbol is foreign
             ;it exists even if we won't be defining it
import exit msvcrt.dll  ;we specify the external library that defines the symbol
      ; msvcrt.dll contains exit, printf and all the other important C-runtime functions

; our variables are declared here (the segment is called data)
segment data use32 class=data
; ...

; the program code will be part of a segment called code
segment code use32 class=code
start:
; ...

   ; call exit(0) ), 0 represents status code: SUCCESS
   push dword 0 ; saves on stack the parameter of the function exit
   call [exit] ; function exit is called in order to end the execution of
the program
```

Exercises

Ex 1: 4+2

segment code:

```
Mov al,4        ; al=4
Add al,2        ; al=1+2=3
```

Ex 2: 5-1

segment code:

```
Mov al,1        ; al=1
Sub al,5        ; al=-4
```

Ex. 3: a+b, a,b – word

```
segment data
        a dw 5
        b dw 7
segment code
        Mov  AX, [a]; ax=5
        Add AX, [b]; ax=12
```

Ex. 4: (a+b) - c; a,b,c – doublewords

```
segment data
        A  dd 2
        B dd 5
        C dd 12
segment code
        Mov eax, [a]; eax=2
        Add eax, [b]; eax=7;
        Sub eax, [c]; eax=-5
```

3. Representation of integer numbers in the computer's memory

Consider the following instruction:

mov ax, 7

ax=0007h = 00000000 00000111b (you will talk about converting numbers between bases at your first lab)

The above instruction instructs the CPU (i.e. microprocessor) to set the value of the ax register (which is a memory zone on the CPU) to 7. A natural question that arises is: **how does the CPU *represent integer numbers in the memory (and also on the CPU registers)* ?**

The CPU represents an integer number on 1, 2, 4 or 8 bytes on the IA-32 architecture (1 byte = 8 consecutive bits). In fact, there are **two kinds of representation** of integer numbers in the computer's memory: **signed representation and unsigned representation**. *The CPU **chooses** one of these two representations **depending on the specific instruction** it executes*.

Unsigned representation of numbers

- in unsigned representation we can **only represent positive** natural numbers
- the unsigned representation of a positive number is equal to the representation of that number in base 2
- ex.1: the unsigned representation of 17 on 8 bits is : 0001 0001
- ex.2: the unsigned representation of 5 on 8 bits is : 0000 0101

Signed representation of numbers

- in the signed representation we can represent positive and negative integer numbers
- the **signed representation of a positive number is equal to the unsigned representation of that number** (i.e. it is equal to the representation of that number in base 2)
- the **signed representation** of a negative number is equal to the representation **in *2's complementary code*** (Romanian: codul complementar fata de 2) of that number; in order to obtain the *2's complementary code* of a negative number, **we subtract the absolute value of the number (Romanian: modulul numarului) from 1 followed by as many zeroes as needed in order to represent the absolute value of the number**.
- in the signed representation, the most significant bit (i.e. binary digit) of the representation is the sign bit(1=negative number; 0=positive number).
- ex.1: the signed representation of **17** on 8 bits is : 0001 0001 (the most significant bit is 0, so the number is positive)
- ex.2: the signed representation of **-17** on 8 bits is : 1110 1111 (note that the sign bit is 1 in this case, so the number is negative)

  1 0000 0000 –
  1 0001
  1110 1111
- ex.3: the signed representation of -39 on 16 bits is : 1111 1111 1101 1001 (note that the sign bit is 1 in this case, so the number is negative)

  1 0000 0000 0000 0000–
  10 0111
  1111 1111 1101 1001

Now, we can consider the reverse problem of "representation", that is "interpretation". Let's assume that the binary content of the AL register is 1110 1111 and the next instruction to be executed is:

When the CPU executes an instruction (for example a multiplication) it needs to ask itself the question (the human programmer also asks himself the same question): ***what integer number does the sequence of bits from AL (i.e. 1110 1111) represents in our conventional numbering system*** *(i.e. base 10)*? The CPU must ***interpret*** the sequence of bits from AL into a number in order to perform the mathematical operation (multiplication).

Just like we have two **types of "representations**", we also have two corresponding **types of "interpretations**": **signed interpretation and unsigned interpretation.**

In our example where we have in the AL register the sequence of 8 bits: 1110 1111, this value can be interpreted:

- unsigned: in this case, we now that in the unsigned representation, only positive numbers are represented, so our sequence of bits represents a positive number and it is the representation in base 2 of that number; so the number in base 10 is:

  1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1* 2^2 + 1*2^1 + 1*2^0=

  128 + 64 + 32 + 0 + 8 + 4 + 2 + 1 = 239
- signed: in this case, we know that in the signed representation the most significant bit of the representation is the sign bit; our sign bit is 1 which means that this is the signed representation of a negative number; in other words this is the complementary code representation of the negative number; in order to obtain the *direct code* (the representation in base 2 of the absolute value of the number) we use the following rule: *take all the bits of the complementary code representation, from right to left, keep all the bits until the first 1, including this one, and reverse the remaining bits (1 becomes 0, 0 becomes 1)*. So, for our example of 1110 1111, the direct code is: 0001 0001 = 17. So the sequence of 8 bits 1110 1111 from the memory is interpreted signed into the number -17.

4. Signed and unsigned instructions

On the IA-32 architecture, related to the unsigned and signed representation of numbers, there are 3 classes of instructions:

- instructions which do not care about signed or unsigned representation of numbers: **mov, add, sub**
- instructions which interpret the operands as unsigned numbers: **div, mul**
- instructions which interpret the operands as signed numbers: **idiv, imul, cbw, cwd, cwde**

It is important to be consistent when developing a IA-32 assembly program: either consider all numerical values in a program to be unsigned (in which case you should use only instructions from class 1 and 2) or consider all numerical values in a program to be signed (in which case you should use only instructions from class 1 and 3).