Asc 6

**CHAPTER 3: ASSEMBLY LANGUAGE BASICS**

We always are able to find a solution!!!

Machine language

* The native language which the arcitecture understands

Assembly language

* The english version of the machine language
* It is a symbolic language (you as a programmer are expressing yourself by labelling and mnemonics = names for instructions and giving names to variables)
* The processor translates everything in numbers and at assembly time evaluate everything that can be evaluated
* Any offset that is issued or accessed as a direct adressing mode (everytime you put name of variable) the processor will direct to the constant part „your name” is pointing to
* It is focused on the assembler part and it’s tasks (NASM)
* The big diff is in assembler (a translation program) has 2 tasks
  + Check the validity of the code
  + Generates bytes according to the instructions from the source code
* During this process has the task to evaluate expressions which must be computable at assembly time
* These rules are applied for any assembler!!

The basic elements with which an assembler works are:

* Labels
* Instructions
* Directives
* Location counter
  + Any memory segment from your program has it’s own location counter

AN ADRESS IS ALWAYS AN INTEGER NUMBER => „Sa ma ierte domnu” pt sqrt(pointer)

You will not obtain a valid adress by multiplying pointers

Let’s take the last 2 pointers, by adding or substracting you will obtain something outside of memory

Pointer arithmetic (operations with pointers) (NOT 8, ONLY 3)

We work with POINTERS AS DATA TYPES

* (M-a nenorocit)
* Subtraction
  + By doing this, it will show me how many bytes of memory are from the beginning of the memory area to the end of the memory area
  + Example: By performing a p – q -> you obtain the bytes between p and q
  + $ - a -> it is the diff between the beginning of the memory area to the place a points to
  + P – Q scalar value
* Addition with a constant
  + A[7] = \*(A + 7) = \*(7+A) = 7[A] – the contents from a+7
  + The constant acts as an index in trying to pass a logical area or a certain area inside an array
  + P + 7 - a pointer data type
* Subtracting a constant
  + Example: P – 7
  + It allows you to go inside the memory without going of it
  + WHAT IF YOU GO OUT OF IT?
    - (If you take the last possible memory adress which is represented on a double word you will go outside of a doubleword)
    - Usually it will tell you ‚memory access violation”
    - In some cases it accepts and can make it circulate back to the beginning
    - Nevertheless it is an awkward idea
  + P – 7 - a pointer data type

In NASM **p + q** is supported. This is because a **p + q** doesn’t mean addition of 2 pointers. This is an expression with pointers but not pointer arithmetic

**Do we use arithmetic in formula from 2am?**

[base] + [index \* scale] + [const]

It is not pointer arithmetic because you don’t have pointers there, but you are required to use addition signs and not substractions

**EXAMPLES**

V db 17

A dw 232, -17 , ‚xy’

B dd 12345678h

Add edx, [ebx + ecx \* 2 + v - 7]

* CORRECT!
* V – 7 -> constant
* Ebx -> base
* Ecx \* 2 -> index \* scale

Mov ebx, [ebx + ecx \* 2 - v – 7]

* SYNTAX ERROR! – Impossible segment base modifier (-v is not accptible)

Mov [ebx + ecx \* 2 + a + b – 7]

* SYNTAX ERROR!
* You don’t have pointer addition (a + b)

Sub [ebx + ecx \* 2 + a – b – 7], eax

* CORRECT! (but depends)
* From a syntactical point of view it is correct
* It will calculcate everything correctly, but it will decide after it calculates wether it is a memory violation, depending on the value of a-b-7

While a is a word and b is a dword, we do not substract their values(continut), but the adresses which are represented on 32 bits!!

ANY PROGRAM THAT WAS WRITTEN IN HISTORY will always end with a very primitive thing: it is all based on an assignment segment. It is nothing else than a poor sequence of assignment segments. They are sad attemps of changing the values of memory cells =(((

i := i + 1

L R

LHS RHS

* L – value
* R - value
* LHS – ADRESS
* RHS – CONTENTS (it has to be dereferenced)
* If we are talking about the left and right value of i, is there a diff between these 2 or do they express the same thing?
  + When the processor sees this command, it goes to the memory adress of i and then STOPS (there is no diferencering)

In clasa a saptea vancea nu intelegea ca i e i+1

C++

+ - OBJECT ORIENTED PROGRAMMING

+ - MANY OTHER FEATURES THAT WEREN’T PRESENT IN C

Example:

* you can only pass by refference in c++, not in c
* You cannot define a constant in c
* C++ has c++ refference variables
  + ussage: define alliases
    - Ampersant is c++ refference variable
    - Int &j = i // you define alliases
  + Ussage: pass by refference some parameters
  + Ussage: return left values
    - Examples: int &F(x,y){...return v[i]}
    - F(a,7) = 49 -> it will return in the left area, meaning you return the left value of v[i]
    - V[7] = 49

NOT THE GENERAL FORM!!!!

Symbol := expression

Identifier := expression

TRUE GENERAL FORM

**OFFSET SPECIFICATION VALUE!!!!**

Segment data

A db 17, -2, 0ffh, ‚xyz’

B db 232, -112

C db -3, 80h, 7, -2, 432

Lga dw $-a

Lga db $ - $$ (usually it’s not the same, but in our case yes, because a is the first defined element in the data segment)

Lga EQU $-a; it’s ok from the point of view of $-a. EQU is a dirrective, so it’s a command adressed to the assembler by which we define our constant

Mov [lga], ax; you don’t have a memory area associated to lga

Lga dw lga – a; the moment we write lga dw, the variable is already defined so we can use it and it is exactly where $ is!!

$ - the location counter (offset, pointer)

$$ - the starting adress of your segment (...)

What is the starting offset which you see in your code segment and data segment?

* The adresses to which ollydebugger is loaded at
* Olly is always loaded at the same adress
* The offset of the end of a minus the offset of the beginning of a