8086 assembler tutorial for beginners (part 3)

Variables

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "**var1**" then at the address 5A73:235B, especially when you have 10 or more variables.

Our compiler supports two types of variables: **BYTE** and **WORD**.

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

DB - stays for Define Byte.
DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

As you probably know from *part 2* of this tutorial, **MOV** instruction is used to copy values from source to destination.
Let's see another example with **MOV** instruction:

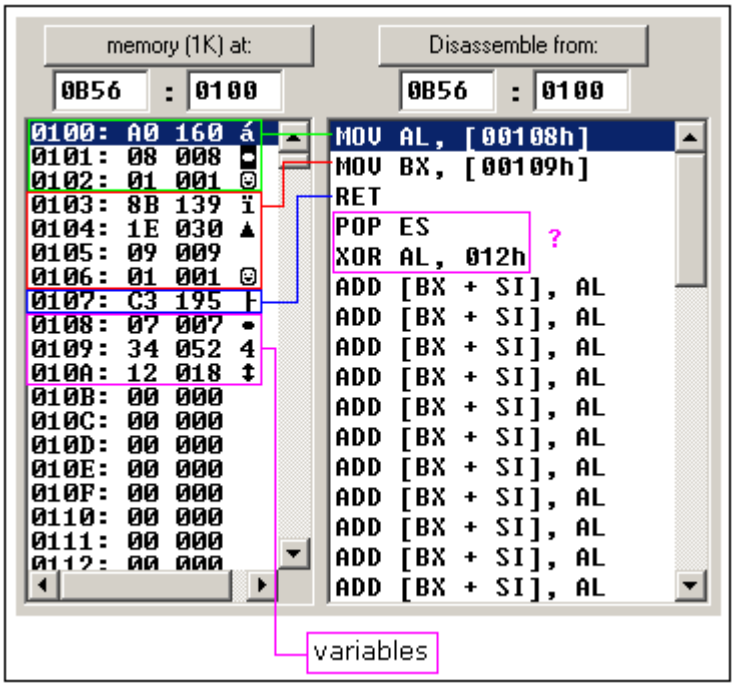
```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; stops the program.

VAR1 DB 7
var2 DW 1234h
```

Copy the above code to the source editor, and press **F5** key to compile it and load in the emulator. You should get something like:



As you see this looks a lot like our example, except that variables are replaced with actual memory locations. When compiler makes machine code, it automatically replaces all variable names with their **offsets**. By default segment is loaded in **DS** register (when **COM** files is loaded the value of **DS** register is set to the same value as **CS** register - code segment).

In memory list first row is an **offset**, second row is a **hexadecimal value**, third row is **decimal value**, and last row is an **ASCII** character value.

Compiler is not case sensitive, so "**VAR1**" and "**var1**" refer to the same variable.

The offset of **VAR1** is **0108h**, and full address is **0B56:0108**.

The offset of **var2** is **0109h**, and full address is **0B56:0109**, this variable is a **WORD** so it occupies **2 BYTES**. It is assumed that low byte is stored at lower address, so **34h** is located before **12h**.

You can see that there are some other instructions after the **RET** instruction, this happens because disassembler has no idea about where the data starts, it just processes the values in memory and it understands them as valid 8086 instructions (we will learn them later).
You can even write the same program using **DB** directive only:

```
ORG 100h

DB 0A0h
DB 08h
DB 01h

DB 8Bh
DB 1Eh
DB 09h
DB 01h
```

```
DB 0C3h

DB 7

DB 34h
DB 12h
```

Copy the above code to the source editor, and press **F5** key to compile and load it in the emulator. You should get the same disassembled code, and the same functionality!

As you may guess, the compiler just converts the program source to the set of bytes, this set is called **machine code**, processor understands the **machine code** and executes it.

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.
Why executable file is loaded at **offset** of **100h**? Operating system keeps some data about the program in the first 256 bytes of the **CS** (code segment), such as command line parameters and etc.
Though this is true for **COM** files only, **EXE** files are loaded at offset of **0000**, and generally use special segment for variables. Maybe we'll talk more about **EXE** files later.

Arrays

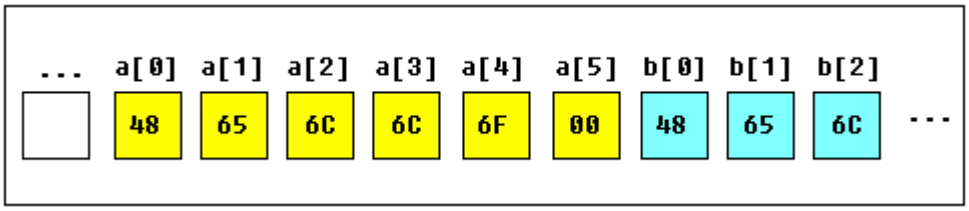
Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h

b DB 'Hello', 0
```

b is an exact copy of the *a* array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:



You can access the value of any element in array using square brackets, for example:

```
MOV AL, a[3]
```

You can also use any of the memory index registers **BX, SI, DI, BP**, for example:

```
MOV SI, 3

MOV AL, a[SI]
```

If you need to declare a large array you can use **DUP** operator.
The syntax for **DUP**:

number DUP (value(s))
number - number of duplicate to make (any constant value).
value - expression that DUP will duplicate.

for example:

```
c DB 5 DUP(9)
```

is an alternative way of declaring:

```
c DB 9, 9, 9, 9, 9
```

one more example:

```
d DB 5 DUP(1, 2)
```

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings.

Getting the Address of a Variable

There is **LEA** (Load Effective Address) instruction and alternative **OFFSET** operator. Both **OFFSET** and **LEA** can be used to get the offset address of the variable.
LEA is more powerful because it also allows you to get the address of an indexed variables. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

Reminder:
In order to tell the compiler about data type,
these prefixes should be used:

BYTE PTR - for byte.
WORD PTR - for word (two bytes).

For example:

```
BYTE PTR [BX]     ; byte access.
    or
WORD PTR [BX]     ; word access.
```

assembler supports shorter prefixes as well:

b. - for **BYTE PTR**
w. - for **WORD PTR**

in certain cases the assembler can calculate the data type automatically.

Here is first example:

```
ORG 100h

MOV    AL, VAR1              ; check value of VAR1 by moving it to AL.

LEA    BX, VAR1              ; get address of VAR1 in BX.

MOV    BYTE PTR [BX], 44h    ; modify the contents of VAR1.

MOV    AL, VAR1              ; check value of VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Here is another example, that uses **OFFSET** instead of **LEA**:

```
ORG 100h

MOV    AL, VAR1             ; check value of VAR1 by moving it to AL.

MOV    BX, OFFSET VAR1      ; get address of VAR1 in BX.

MOV    BYTE PTR [BX], 44h   ; modify the contents of VAR1.

MOV    AL, VAR1             ; check value of VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Both examples have the same functionality.

These lines:

```
LEA    BX, VAR1

MOV    BX, OFFSET VAR1
```

are even compiled into the same machine code: MOV BX, num
num is a 16 bit value of the variable offset.

Please note that only these registers can be used inside square brackets (as memory pointers): **BX, SI, DI, BP**!
(see previous part of the tutorial).

Constants

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants **EQU** directive is used:

```
name EQU < any expression >
```

For example:

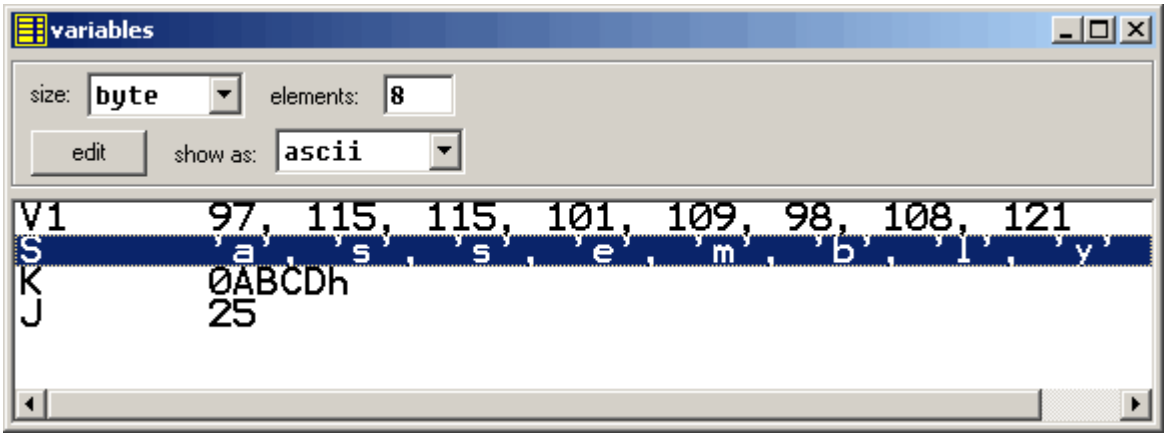
```
k EQU 5


MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

You can view variables while your program executes by selecting "**Variables**" from the "**View**" menu of emulator.



To view arrays you should click on a variable and set **Elements** property to array size. In assembly language there are not strict data types, so any variable can be presented as an array.

Variable can be viewed in any numbering system:

- **HEX** - hexadecimal (base 16).
- **BIN** - binary (base 2).
- **OCT** - octal (base 8).
- **SIGNED** - signed decimal (base 10).
- **UNSIGNED** - unsigned decimal (base 10).
- **CHAR** - ASCII char code (there are 256 symbols, some symbols are invisible).

You can edit a variable's value when your program is running, simply double click it, or select it and click **Edit** button.

It is possible to enter numbers in any system, hexadecimal numbers should have "**h**" suffix, binary "**b**" suffix, octal "**o**" suffix, decimal numbers require no suffix. String can be entered this way:
'hello world', 0
(this string is zero terminated).

Arrays may be entered this way:
1, 2, 3, 4, 5
(the array can be array of bytes or words, it depends whether **BYTE** or **WORD** is selected for edited variable).

Expressions are automatically converted, for example:
when this expression is entered:
5 + 2

it will be converted to **7** etc...

<<< previous part <<< **>>> Next Part >>>**
