**Basics of 8086 Emulation**

Table of Contents

[Software Installation 2](#_Toc83228405)

[Assembly Language Code 3](#_Toc83228406)

[Emulation 4](#_Toc83228407)

[Data Transfer Instructions 6](#_Toc83228408)

[Arithmetic Operations 7](#_Toc83228409)

[Other Features 8](#_Toc83228410)

## Software Installation

We will be using emu8086 as an emulator. It can be downloaded from [here](https://archive.org/details/tucows_325007_Emu8086_-_Microprocessor_Emulator). The installation process is straightforward.

## Assembly Language Code

Open up the emulator and click on the New button and then Empty Workspace. This will bring up a blank editor screen. Paste in the sample code:

ORG 0100H *; Program start memory location*.DATA *; Data Segment Starts*A DB 11 *; BYTE value 11*B DW 500 *; WORD value 500*SUM DW ? *; WORD variable without any value*DIFFERENCE DB ? *; BYTE variable without any value*.CODE *; Code Segment Starts*MAIN PROC *; Main procedure starts* MOV AL, 30 *; Move decimal 30 to AL register* ADD AL, 15 *; Add decimal 15 to the content of AL; store the result in AL*MAIN ENDP *; End Procedure*END MAIN *; End MAIN*RET *; Return to DOS*

ASSEMBLY

Most of the code should be understandable from the comments. The **segments** defined will be discussed in detail later, but for now, they can be thought of as **sections**.

In the data segment, the variables declared are of two types, either DB or DW. DB variables occupy 1 byte of memory while DW variables occupy two bytes. The variable B required two bytes since it has a value of 500, while 1 byte can store a maximum value of 255.

In the code segment, the MAIN procedure is started. Again, this is something we will discuss in depth later, but it can be compared to the main function from C++ for now. Here, we write our actual source code. Once this is done, we end the MAIN procedure, end the program and then finally return to the terminal.

## Emulation

If you click on the emulate button at the top, two new windows will pop up. One will have a copy of the source code we wrote, while the other will show all the registers in the system.

There are several registers visible in the emulator. AX, BX, CX and DX are **general purpose** registers, while the rest have special uses. The general-purpose registers are all **16-bit registers**, divided into two parts, **high** and **low**. Thus, each part is called AH, AL, BH, BL and so on. The reason for this division is to conserve memory. Intel 8086 allows both 16-bit and 8-bit operations, so operations that do not require the full 16-bits of the register will be allocated one half of it. Note that both of these halves are registers on their own.

There should be a yellow highlight at the beginning of the data segment. This is where the program will begin. If you check the **IP** register, the value should be 0100. This is the hex value we set on the very first line, the memory location where our program begins.

Click on the single step button to execute one more instruction. This should bring the yellow highlighting to the line where we move 30 to AL.This line has not yet been executed. The **IP** register should now have a value of 0108. The four variables we declared have a total memory of 6 bytes, and the 2 extra bytes are for a special purpose that shall be discussed later.

Click on single step once more. This will move 30 to AL. The value of the AL register should be 1E, the hex form of . The reason we moved 30 to AL specifically is because 30 does not need 16 bytes of memory, so we should not use it.

Click single step again to add 15 to AL. The value of the AL register should be 2D, the hex format of .

Notice that for each step, the value of the **IP** register changed accordingly.

## Data Transfer Instructions

We can move contents between registers, but we need to make sure the registers have the **same size**.

MOV AL, BL  
MOV AX, BX

ASSEMBLY

We can move a value directly to a register. Decimal values can be moved directly, but binary and hex values need to be indicated with a B and an H respectively.

MOV CL, 240  
MOV CL, 11110000B  
MOV CL, 00F0H  
MOV CX, 256  
MOV CX, 0000000100000000B  
MOV CX, 0100H

ASSEMBLY

We can also move variables into registers, but we need to make sure the register we are moving it into is the appropriate size. DB variables need to go to AL, BL, CL or DL registers while DW variables need to go to AX, BX, CX or DX registers.

MOV DL, COUNT *; here COUNT is a 8-bit variable*MOV DX, COUNT *; here COUNT is a 16-bit variable*

ASSEMBLY

## Arithmetic Operations

*; Addition operations. Results stored in AX.*ADD AX, 4  
ADD AX, BX  
  
*; Subtraction operations. Results sored in DL/DX.*SUB DL, 4  
SUB DX, CX  
  
MUL BL *; Multiply AX by BL. Results stored in AX.*DIV CL *; Divide AX by CL. Results stored in AX.*INC BX *; Increment BX*DEC BX *; Decrement BX*XOR AX, AX *; Clear AX*NEG AL*; 2’s Complement*NOT AL*; 1’s Complement*

ASSEMBLY

For the division operation, if the **quotient** is **8-bits** or less, it is stored in the AL register and the **remainder** is stored in the AH register. If the **quotient** is more than **8-bits**, it is stored in the AX register and the **remainder** is stored in the DX register.

For both the multiplication and division operations, if the **existing value** of the AX register is more than **8-bits**, then we need to use the BX register to store the value being operated with. Essentially, the value needs to be stored as a 16-bit value, even if it does not need it. Otherwise, we will get incorrect results.

Regarding the XOR operation, this is actually a part of a set of **logical operations**. We can also use the AND, OR and other logical operations. In this particular case, since we are XORing the value of a register with itself, the result is , thus ‘clearing’ the register.

## Other Features

There are a few more features that are important in emu8086.

Towards the top of the **editor**, there is a **calculator** and a **convertor**. The calculator is a little weird and may get a while to get used to, but the convertor is extremely useful for converting between numbers in different bases to do things like check register values.

In the **emulator** screen, towards the bottom, there are several buttons. One of these is flags. Clicking this reveals a small window that shows all the different **flag values**. We need to check here in case of things like an **overflow**. Another button there is vars, which brings up a window that shows the values of the different variables we declared in our code.