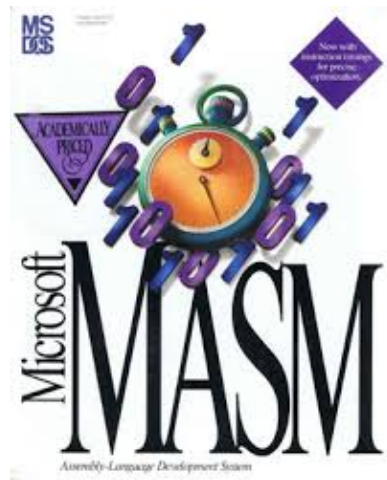


MASM Code Analysis: Multiples of 11

Marc Aliaga

GitHub: github.com/markh4ck

Email: marcaliaga@gmail.com



“Understanding assembly language means understanding the foundation of computation itself.”

Introduction

In this report, we will analyze a MASM assembly program that counts the multiples of 11 between 0 and 1000. The complete code is divided into segments, each explained in detail on separate pages, including visual illustrations and memory flow explanations.

Block 1 — Model definition, libs and .data segment

```
1 .386
2 .model flat, stdcall
3 .stack 4096
4
5 includelib kernel32.lib
6
7 ExitProcess PROTO :DWORD
8 GetStdHandle PROTO :DWORD
9 WriteConsoleA PROTO :DWORD, :DWORD, :DWORD, :DWORD, :DWORD
10
11 STD_OUTPUT_HANDLE EQU -11
12
13 .data
14
15 bytes\_written dd ?
16
17 mensaje db "Cantidad de multiplos de 11: ",0 ;db = define byte (cada
    char es un byte)
18
19 buffer db 16 dup(0) ; array de 16 bytes
20
21 contador dd 0 ;double word 4 bytes
22
23 handle dd ? ;guardamos el valor del handle
```

Listing 1: Configuración inicial y sección de datos

Explanation

In this section, we define both the structure of the functions that will be used from the Windows API and the variables that can be considered as constants within the program.

At the top of the listing, we specify the **instruction model** using the directive `.model flat, stdcall`, which defines the calling convention (how parameters are passed to functions) and how memory segments are organized. We also set the **stack size** to 4096 bytes through the directive `.stack 4096`. This stack serves as a temporary storage area for local variables, parameters, and return addresses during program execution.

The **flat memory model** means that all code, data, and stack share the same continuous address space in RAM, as opposed to older segmented memory models from 16-bit architectures. This simplifies addressing and is the standard model in modern 32-bit and 64-bit Windows programs.

Finally, the constant `STD_OUTPUT_HANDLE` represents the standard output device, usually the console window. It will be used to store the handle (a reference value provided by the operating system) that allows the program to write text directly to the console using the Windows API function `WriteConsoleA`.

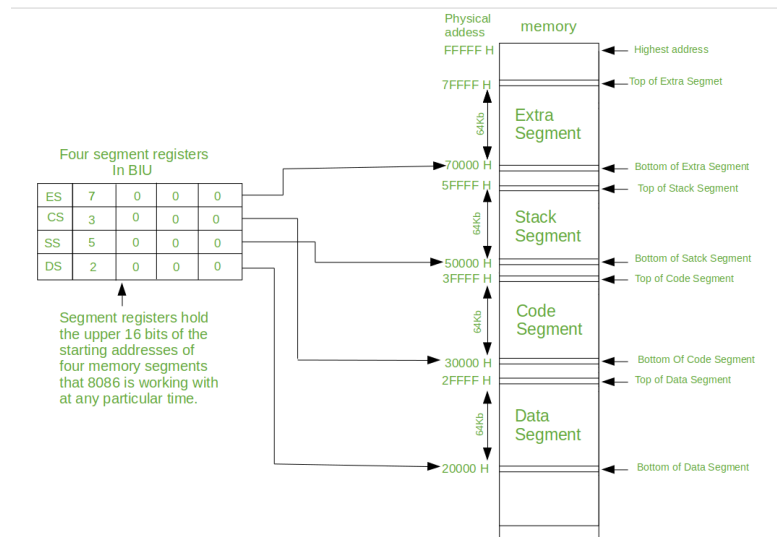


Figure 1: Memory segments

Block 2 — Main loop

```
1  .code
2
3  main PROC
4
5  push STD\ _OUTPUT\ _HANDLE
6
7  call GetStdHandle
8
9  mov [handle], eax
10
11 mov ecx, 1 ; hace como la variable n en C
12
13 mov ebx, 0 ; hace de contador para el "main" como en C
14
15 ;PARTE 1
16
17 bucle:
18
19 ;-----While(n<100)\{} -----
20
21 cmp ecx, 1000
22
23 jge fin\_bucle ; este salto, solo se ejecuta si ecx es mas grande o
    igual a 1000.
24
25 ;jump if is greater or equal
26
27 mov eax, ecx ; movemos el valor de ecx a eax para hacer la division.
28
29 cdq ;convertimos de 32 bits a 64 EDX:EAX
30
31 mov esi, 11 ;cargamos el divisor
32
33 div esi ; EDX:EAX / ESI
34
35 ;----- if (n \% 11 == 0) \{
36
37 ; contador++; \} -----
38
39 ;Una vez hecha la division, el modulo se guarda en EDX y el cociente en
    EAX
40
41 cmp edx, 0 ; si el modulo no es 0, entonces "n" no es un multiplo
42
43 jne no\_multiplo ; jump not equal
44
45 inc ebx; +ebx
46
47 no\_multiplo:
48
49 inc ecx; n++
```

```

50 |
51 | jmp bucle; volvamos hacer el bucle con el nuevo valor en n

```

Listing 2: Bucle principal que cuenta los múltiplos de 11

We start the code in the following way:

We compare if the 'ECX' register is already equal to '1000' (remember we want the multiples of '11' from '0' to '999' or from '1' to '1000').

If they are equal, we jump to the loop "fin.bucle".

But if 'n' is smaller than '1000', we proceed to check if 'n' ('ECX') is a multiple of '11' by obtaining its remainder.

Using the 'cdq' instruction, we extend the register so that one part ('EDX') stores the remainder and the other part ('EAX') stores the quotient.

With that done, we load the divisor into 'ESI' and perform the division:

$$\frac{EDX : EAX}{ESI}$$

Once the division is complete, we must check if the resulting modulo is:

$$remainder == 0$$

If the remainder is not '0', we repeat the loop again with a different value in 'n', and also increase the value of our counter 'EBX'.

multiple < 1000 de 11

$A = B \cdot 10$
 $A \div B = 10$

~~10~~ $\times 11$

max $(11 \div 11 = 1) \rightarrow 11, 22, 33, 44, \dots, 1$

max $(1000 \div 11 = 90.909 \times 11 = \underline{990})$

$$\sum_{k=1}^{90} 11 \cdot k = 90 \cdot 11 + 89 \cdot 11 + \dots$$

$$\sum_{k=1}^{90} 1 = 90$$

$121 = 1 - 2 + 1 = 0$; múltiplo de 11.

$n \bmod 11 = 11$ $\begin{array}{r} 11 \\ 10 \overline{) 9} \end{array}$

Expression	$n = 374$	Posicion
$n \div 100$	3	centenas
$(n \div 10) \bmod 10$	7	decenas
$n \bmod 10$	4	unidades

$\rightarrow 374 \div 10 \Rightarrow 37 \times 10$

Figure 2: Example of number 123 getting the multiple of 11

Block 3 — BUFFER

```
1 fin_bucle:
2
3 mov [contador], ebx
4
5 ; aqui lo que hacemos es pasar el valor del registro ebx que actuaba
   como
6
7 ;contador, a la direccion de memoria reservada para ese valor .
8
9 ; pasamos de CPU a RAM, ebx ---> [EBP-x] donde x es un numero R
10
11 mov eax, [contador]
12
13 ; esta parte esta ilustrada en la imagen 3.
14
15 ;pero lo quu ehacemos aqui es ir rellenando el buffer empezando desde
   la ultima posicion.
16
17 lea edi, buffer + 15 ;obtenemos la direccion base "EBP" del buffer y le
   sumamos el offset +15 para ir al final
18
19 mov byte ptr [edi], 0
20
21 mov ebx, edi ; Usamos EBX para guardar la direcci n de inicio del
   n mero. Lo inicializamos al final.
22
23 cmp eax, 0
24
25 je es_cero ; Salta si el contador es 0 (no se ejecutar a el bucle)
26
27 jmp inicio_conv_bucle
28
29 es_cero:
30
31 dec edi ; Retrocede a buffer + 14
32
33 mov byte ptr [edi], '0' ; Escribe el '0'
34
35 mov ebx, edi ; EBX ahora apunta al primer d gito ('0')
36
37 jmp fin_conv_bucle
```

Listing 3: Conversi3n de n3mero a cadena ASCII

Assuming now that ‘n = 1000’, what we want to do next is move the counter of numbers that are multiples of 11, which resided in ‘EBX’, into the variable ‘[CONTADOR]’ so we can proceed to print that number to the standard output.

For that, we need to convert the number to ASCII. To achieve this, we need to separate the number into individual digits and find their ASCII equivalents to add them to the array of numbers in the ‘BUFFER’, which as we defined earlier, is an array with a maximum of 15 operational bytes, since the byte at the final position is the null-terminator “0”.

Additionally, the number 90 (the amount of numbers that are multiples of 11 from 'n' to 1000) must be filled into the buffer from right to left, because if we start from 9, we'd get "09", and what we want is "9" + "0".

For that, we use instructions like 'lea' (to obtain the memory address of the buffer) plus the 15 offset.

Let's remember that we have 'buffer' at a memory address like: '0x100', and the last position would be at 'buffer + 15', which is the same as '0x100F'.

It should be noted that: 'lea edi, buffer + 15' equals 'mov edi, OFFSET 0x100 + 15' (or 'buffer + 15' in case we don't know the base address of 'buffer'). This resides in the '.data' segment.

In case 'buffer' is on the stack, we could do:

sub esp, 16- mov edi, esp- add edi, 15

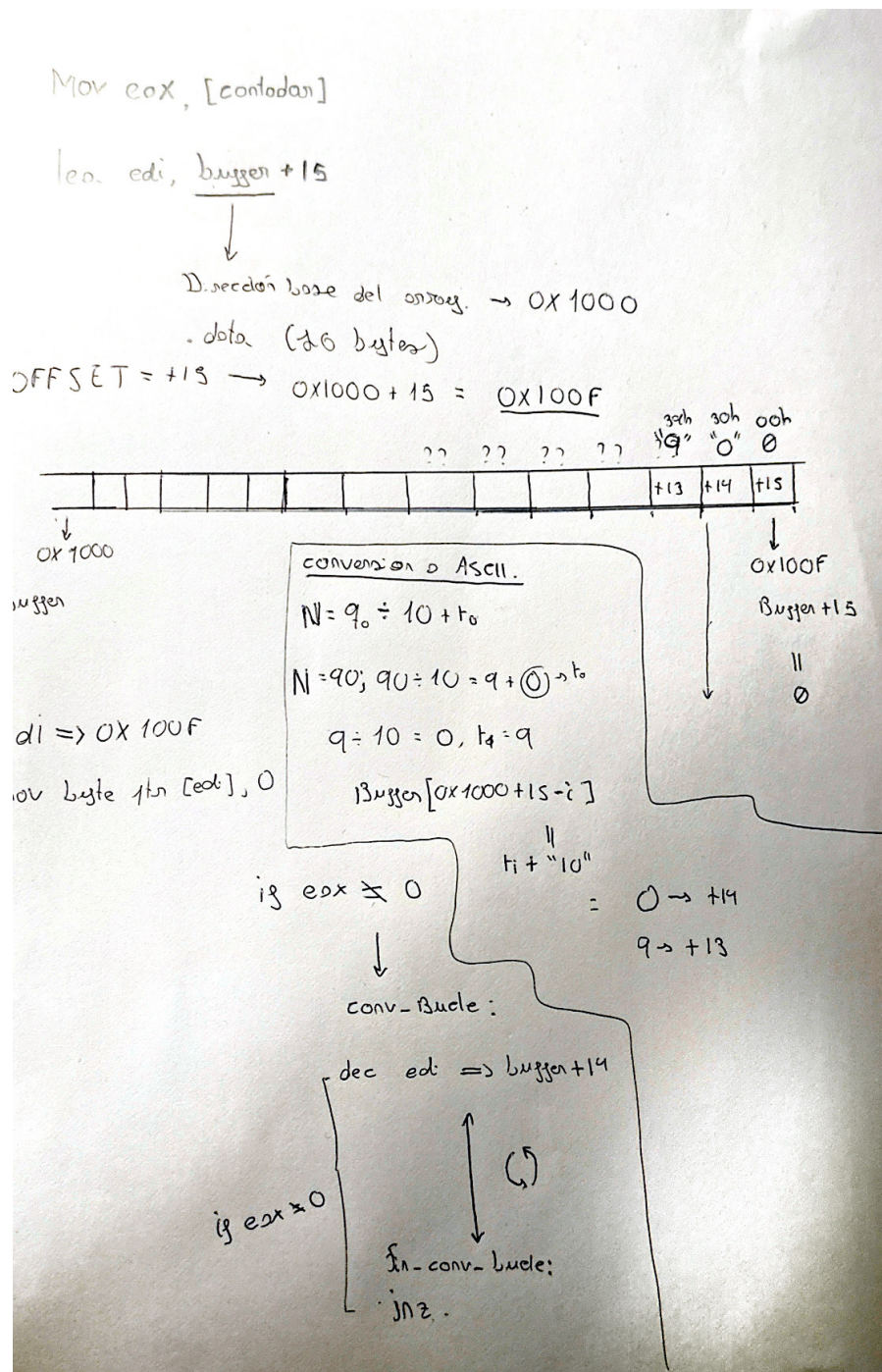


Figure 3: Logic to do the conversion to ASCII

Segmento 4 — OUTPUT and WRITECONSOLEA call

```
1 fin_conv_bucle:
2     push 0
3     push offset bytes_written
4     push 29
5     lea eax, [mensaje]
6     push eax
7     push dword ptr [handle]
8     call WriteConsoleA
9
10    lea eax, buffer+15
11    sub eax, ebx
12    push 0
13    push offset bytes_written
14    push eax
15    push ebx
16    push dword ptr [handle]
17    call WriteConsoleA
18
19    push 0
20    call ExitProcess
21
22 main ENDP
23 END main
```

Listing 4: Impresión del mensaje y del número en consola

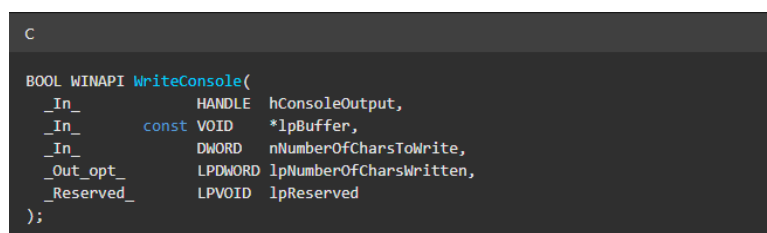
In this part, we simply follow the structure of the Windows API function.

`bytes_written` is a variable that stores the number of bytes corresponding to the length of the message.

In the first text output from `[mensaje]`, since it has a fixed size, we specify it manually using `push 29`.

However, in the next output, we calculate the length by subtracting the value of `eax - ebx`, that is, the base address (`0x1000 - 0x100D`). In this way, we obtain the exact number of characters to be printed.

The other parameters are used for the correct operation of the function, such as `[handle]`, which is used to obtain the standard output resource.



```
C
BOOL WINAPI WriteConsole(
    _In_ HANDLE hConsoleOutput,
    _In_ const VOID *lpBuffer,
    _In_ DWORD nNumberOfCharsToWrite,
    _Out_opt_ LPDWORD lpNumberOfCharsWritten,
    _Reserved_ LPVOID lpReserved
);
```

Figure 4: WritecosnoleA structure Windows API