

Assembly Language Crash Course

Introducción

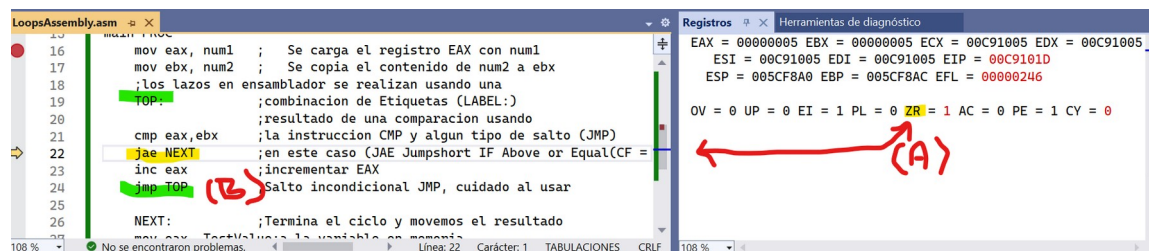
El lenguaje Ensamblador es simple, la dificultad esta en tener que poner atención a los detalles y realizar una escalabilidad para completar tareas complejas simplemente usando el conjunto sencillo de instrucciones que ofrecen los microprocesadores, las cuales pueden llegar a ser una gran cantidad en Microprocesadores de nueva generación. Este documento comenzara describiendo las operaciones mas comunes a cualquier lenguaje de programacion (Instrucciones de asignación, Evaluacion de expresiones, Sentencias Selectivas, Lazos y Llamadas a procedimientos). Consulte los anexos para obtener detalles sobre como configurar un ambiente de desarrollo en la herramienta Visual Studio.

Lazos

Transcriba el siguiente código de ejemplo en la ventana de edición de Visual Studio:

```
1 ;Lazos en lenguaje Ensamblador
2 ;programa para mostrar la implementacion de lazos
3 ;de los programas de alto nivel
4 ;implementacion de ejemplo:
5 ;while (num1 < num2)
6 ; num1 = num1 + 1;
7 ;.386
8 .model flat,stdcall
9 .stack 4096
10 .data
11 num1 DWORD 2
12 num2 DWORD 5
13 TestValue DWORD 8
14 .code
15 main PROC
16     mov eax, num1 ; Se carga el registro EAX con num1
17     mov ebx, num2 ; Se copia el contenido de num2 a ebx
18     ;los lazos en ensamblador se realizan usando una
19     TOP: ;combinacion de Etiquetas (LABEL:)
20         ;resultado de una comparacion usando
21         cmp eax,ebx ;la instruccion CMP y algun tipo de salto (JMP)
22         jae NEXT ;en este caso (JAE Jumpshort IF Above or Equal(CF = 0)
23         inc eax ;incrementar EAX
24         jmp TOP ;Salto incondicional JMP, cuidado al usar
25
26     NEXT: ;Termina el ciclo y movemos el resultado
27     mov eax, TestValue;a la variable en memoria
28 main ENDP
29 END main
```

Ejecute usando un Break Point (Punto de Marca) junto con las herramientas de depuracion y ejecucion Paso a Paso (F11):



Se puede apreciar que el programa itera entre las etiquetas (TOP:) y la instruccion `jmp TOP` (B), la ejecucion terminara cuando el resultado de la instruccion `cmp eax, ebx` cumpla con las condiciones que

espera la instrucción *jae* NEXT (A), las cuales se cumplen cuando el resultado de la comparación EAX es Mayor o Igual que EBX, esto activa en el registro de banderas EFL, el bit ZeroFlag ZR, el cual cambia a 1, lo cual es interpretado por la instrucción *JAE* como verdadero y entonces el programa "Salta" hacia la Etiqueta NEXT, continuando la ejecución y saliendo del Lazo. Para mayor información sobre el tipo de instrucciones JMP existentes consulte, la bibliografía, para localizar una tabla como la que se muestra:

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
77 cb	<i>JA rel8</i>	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 cb	<i>JAE rel8</i>	D	Valid	Valid	Jump short if above or equal (CF=0).
72 cb	<i>JB rel8</i>	D	Valid	Valid	Jump short if below (CF=1).
76 cb	<i>JBE rel8</i>	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 cb	<i>JC rel8</i>	D	Valid	Valid	Jump short if carry (CF=1).
E3 cb	<i>JCXZ rel8</i>	D	N.E.	Valid	Jump short if CX register is 0.
E3 cb	<i>JECXZ rel8</i>	D	Valid	Valid	Jump short if ECX register is 0.
E3 cb	<i>JRCXZ rel8</i>	D	Valid	N.E.	Jump short if RCX register is 0.
74 cb	<i>JE rel8</i>	D	Valid	Valid	Jump short if equal (ZF=1).
7F cb	<i>JG rel8</i>	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D cb	<i>JGE rel8</i>	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C cb	<i>JL rel8</i>	D	Valid	Valid	Jump short if less (SF≠OF).
7E cb	<i>JLE rel8</i>	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠OF).
76 cb	<i>JNA rel8</i>	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 cb	<i>JNAE rel8</i>	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 cb	<i>JNB rel8</i>	D	Valid	Valid	Jump short if not below (CF=0).
77 cb	<i>JNBE rel8</i>	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).

Evaluando Expresiones

Una operación muy común a la hora de programar en lenguaje Ensamblador es convertir una expresión de lenguaje de Alto nivel en una que entienda el Ensamblador, el siguiente ejemplo muestra como convetir la expresion:

$$\text{expresult} = (2+6)-(3+1)$$

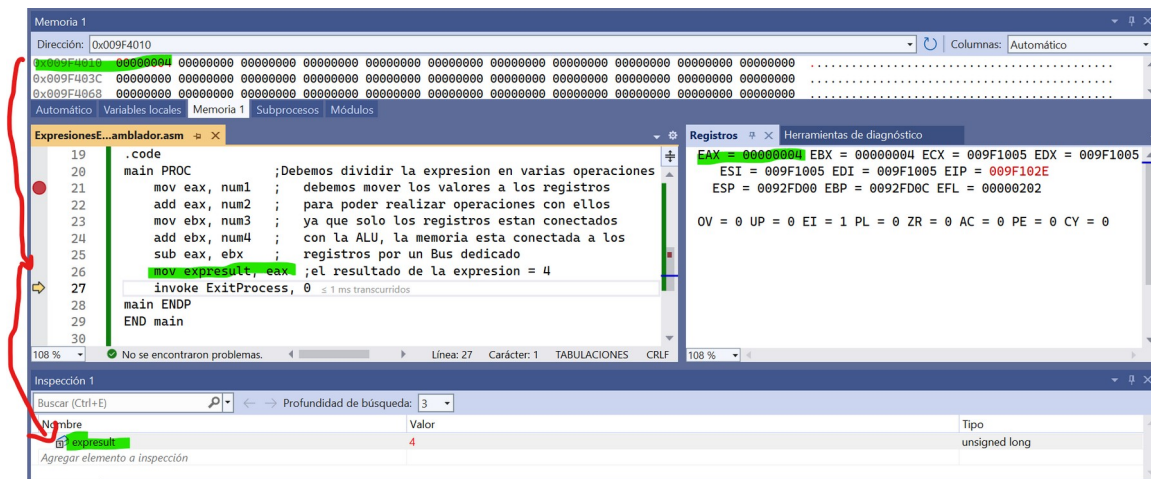
Para que el Microprocesador pueda ejecutar las operaciones y obtengamos el resultado.

```

1  ;Evaluando Expresiones en x86
2  ;Implementar la siguiente expresion de alto nivel en
3  ;Lenguaje Ensamblador x86
4  ;expresult=(2+6)-(1+3) = 4
5  ;Debemos dividir la expresion en varias mas sencillas
6  ;que pueda entender el Ensamblador y debemos de
7  ;valernos de los registros para completar las operaciones
8
9  .model flat,stdcall
10 .stack 4096
11 ExitProcess PROTO,dwExitCode:DWORD
12 .data
13 num1 DWORD 2
14 num2 DWORD 6
15 num3 DWORD 1
16 num4 DWORD 3
17 expresult DWORD ? ; El signo ? indica que no se asigna valor
18 .code
19 main PROC ;Debemos dividir la expresion en varias operaciones
20     mov eax, num1 ; debemos mover los valores a los registros
21     add eax, num2 ; para poder realizar operaciones con ellos
22     mov ebx, num3 ; ya que solo los registros estan conectados
23     add ebx, num4 ; con la ALU, la memoria esta conectada a los
24     sub eax, ebx ; registros por un Bus dedicado
25     mov expresult, eax ;el resultado de la expresion = 4
26     invoke ExitProcess, 0
27 main ENDP
28 END main

```

Nuevamente usando las herramientas de depuración, puentes de espera y ejecución paso a paso, podemos ver como se realiza el proceso.



En la imagen anterior podemos ver que las ubicaciones marcan el mismo resultado, de la evaluación de la expresión, este tipo de separación de una expresión es lo que automáticamente realizan los compiladores de lenguajes de Alto nivel.

Selección (IF Then ... Else ...)

Las sentencias de selección, confían en que los valores ya están cargados previamente en Registros, o las operaciones son sobre de al menos un registro a memoria, deben estar formadas por una combinación de la instrucción *CMP* seguida de un *salto condicional* de la tabla:

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 85 cw	JNE <i>rel16</i>	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
0F 85 cd	JNE <i>rel32</i>	D	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E cw	JNG <i>rel16</i>	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF≠OF). Not supported in 64-bit mode.
0F 8E cd	JNG <i>rel32</i>	D	Valid	Valid	Jump near if not greater (ZF=1 or SF≠OF).
0F 8C cw	JNGE <i>rel16</i>	D	N.S.	Valid	Jump near if not greater or equal (SF≠OF). Not supported in 64-bit mode.
0F 8C cd	JNGE <i>rel32</i>	D	Valid	Valid	Jump near if not greater or equal (SF≠OF).
0F 8D cw	JNL <i>rel16</i>	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
0F 8D cd	JNL <i>rel32</i>	D	Valid	Valid	Jump near if not less (SF=OF).
0F 8F cw	JNLE <i>rel16</i>	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.

Consulte bibliografía

El salto condicional apunta a una ETIQUETA (*LABEL:*), la cual marca la bifurcación de la operación de selección. Si la instrucción tiene un apartado ELSE, deberá contar con otra Etiqueta adicional, tomando en cuenta que el código se ejecuta secuencialmente, es posible evitar ejecutar código, y simular la sentencia de Selección, como muestra el siguiente ejemplo:

```

1 ;tambien es posible emular el comportamiento de las
2 ;sentencias de seleccion
3 ;implementar IF (num1 == num2) THEN
4 ;           x = 1
5 ;           ELSE
6 ;           x = 2
7
8 .386
9 .model flat,stdcall
10 .stack 4096
11 .data
12 num1 DWORD 3
13 num2 DWORD 3
14 x DWORD ?
15 .code
16 main PROC
17     mov eax, num1 ; Cargamos acumulador EAX con num1
18     cmp eax, num2 ; par de instrucciones CMP y JMP forman IF
19     jne L1 ; Comparamos el valor en EAX con num2 (en memoria)
20     ; Si la comparacion no es igual saltar a L1:
21     mov x, 1 ; Si la comparacion es igual continua la ejecucion
22     jmp L2 ; Por lo que tenemos que saltar la siguiente instruccion
23 L1: mov x, 2 ; Esta es el resultado equivalente a Else
24 L2: ; Esta etiqueta es parte del THEN
25
26     mov eax, 5 ; Algo redundante EAX = 5 para ver el cambio
27     mov num1, eax ; num1 = 5, de valor en la ventana de Inspeccion
28     mov eax, num1 ; Despues EAX = Num1
29     cmp eax, num2 ; par de instrucciones CMP y JMP forman IF
30     jne L3 ; Comparamos el valor en EAX con num2 (en memoria)
31     ; Si la comparacion no es igual saltar a L1:
32     mov x, 1 ; Si la comparacion es igual continua la ejecucion
33     jmp L4 ; Por lo que tenemos que saltar la siguiente instruccion
34 L3: mov x, 2 ; Esta es el resultado equivalente a Else
35 L4: ; Esta etiqueta es parte del THEN
36
37 main ENDP
38 END main

```

Caso en el que se cumple la condicion (num1 == num2):

Selección Ensamblador.asm

```

17     mov eax, num1 ; Cargamos acumulador EAX con num1
18     cmp eax, num2 ; par de instrucciones CMP y JMP forman IF
19     jne L1 ; Comparamos el valor en EAX con num2 (en memoria)
20     ; Si la comparacion no es igual saltar a L1:
21     mov x, 1 ; Si la comparacion es igual continua la ejecucion
22     jmp L2 ; Por lo que tenemos que saltar la siguiente instruccion
23 L1: mov x, 2 ; Esta es el resultado equivalente a Else
24 L2: ; Esta etiqueta es parte del THEN
25
26     mov eax, 5 ; Algo redundante EAX = 5 para ver el cambio
27     mov num1, eax ; num1 = 5, de valor en la ventana de Inspeccion
28     mov eax, num1 ; Despues EAX = Num1

```

Registros

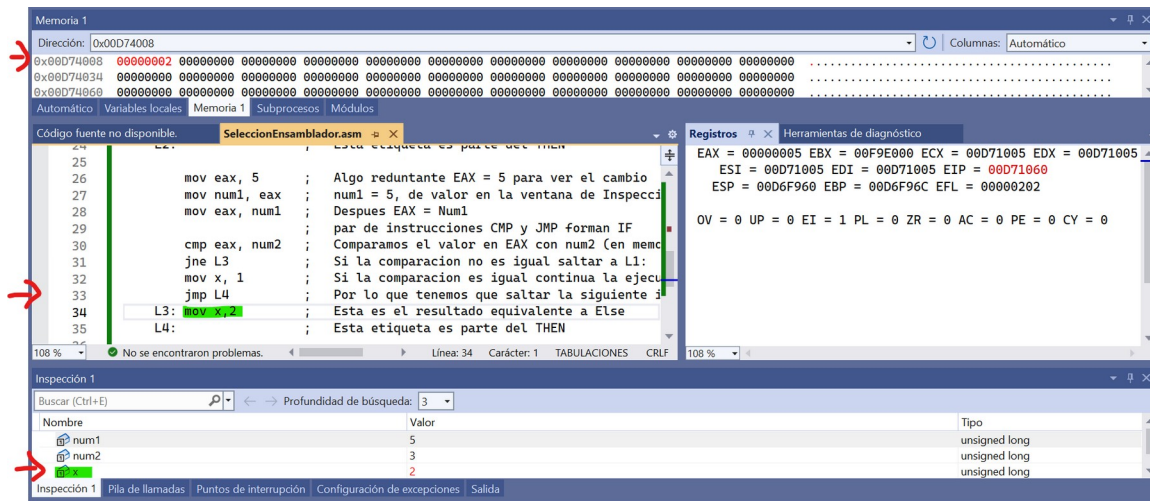
EAX = 00000003 EBX = 00F9E000 ECX = 00D71005 EDX = 00D71005
ESI = 00D71005 EDI = 00D71005 EIP = 00D71027
ESP = 00D6F960 EBP = 00D6F96C EFL = 00000246

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1 CY = 0

Inspección 1

Nombre	Valor	Tipo
num1	3	unsigned long
num2	3	unsigned long
x	1	unsigned long

Caso en el que no se cumple la condicion (num1 != num2):



Se requiere algo de practica acostumbrarse a explotar el hecho de que el código se ejecuta secuencialmente realizando saltos en ubicaciones pertinentes para "conducir" la ejecución de cierto grupo de instrucciones. Nuevamente los compiladores realizan esta adaptación de manera automatica con cada instrucción de Selección (incluyendo las anidadas).

Lazos for

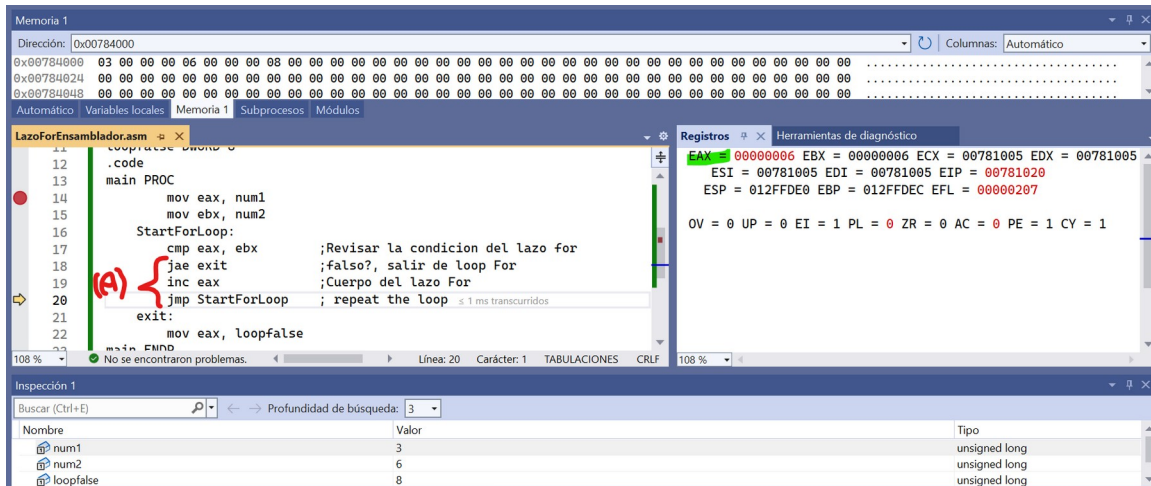
Como hasta ahora hemos hecho es posible implementar un ciclo For en ensamblador, observe el siguiente código:

```

1      ;Lazo For en lenguaje Ensamblador x86
2      ;Implementar el siguiente ciclo For en ensamblador
3      ;for (int i=0; num1 < num2; i++)
4      ;{
5      ; num1=num1+1; /*num1++*/
6      ;}
7      .386
8      .model flat, stdcall
9      .stack 4096
10     .data
11     num1 DWORD 3
12     num2 DWORD 6
13     loopfalse DWORD 8
14     .code
15     main PROC
16         mov eax, num1
17         mov ebx, num2
18     StartForLoop:
19         cmp eax, ebx      ;Revisar la condicion del lazo for
20         jae exit          ;falso?, salir de loop For
21         inc eax           ;Cuerpo del lazo For
22         jmp StartForLoop  ; repeat the loop
23     exit:
24         mov eax, loopfalse
25     main ENDP
26     END main

```

Igualmente usando las herramientas de diagnostico, recorreremos la ejecución del código paso a paso prestando atencion a los cambios del registro EAX.



Prestamos atención a la sección (A) formada por tres instrucciones, una es la condición de salto hacia fuera del lazo, la otra es el contador dentro del lazo, entre esta línea y la anterior se coloca el código que deseamos repetir en el ciclo For, la última línea es el salto al inicio del ciclo For, el uso de etiquetas e instrucciones de comparación emparejadas con saltos puede aumentar la complejidad de un programa, por esta razón existe una instrucción especial para la realización de lazos.

Instrucción loop.

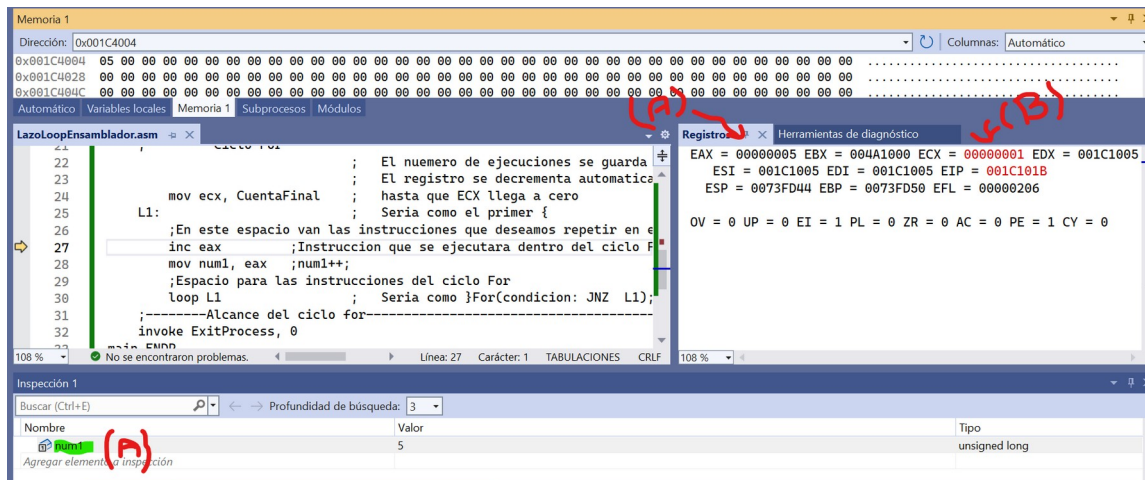
Observe con atención las diferencias entre esta implementación

```

1  ;Lazo For en lenguaje Ensamblador x86
2  ;Implementar el siguiente ciclo For en ensamblador
3  ;for (int i=6; i < 0; i--)
4  ;{
5  ; num1=num1+1; /*num1++*/
6  ;}
7  .386
8  .model flat, stdcall
9  .stack 4096
10 ExitProcess PROTO,dwExitCode:DWORD
11 .data
12 CuentaFinal DWORD 6 ;Este valor servira para iniciar el contador de ciclos
13 ;El cual es para la instruccion LOOP el registro ECX por
14 ;defecto, el cual se va decrementando hasta llegar a cero
15 num1 DWORD 0
16
17 .code
18 main PROC
19     mov eax, num1 ; es el valor de muestra que ira dentro del cuerpo
20 ; del ciclo For, el cual podria estar vacio
21 ;-----Ciclo For-----
22 ; El numero de ejecuciones se guarda en ECX
23 ; El registro se decrementa automaticamente
24     mov ecx, CuentaFinal ; hasta que ECX llega a cero
25 L1: ; Seria como el primer {
26 ;En este espacio van las instrucciones que deseamos repetir en el ciclo For
27     inc eax ;Instruccion que se ejecutara dentro del ciclo For
28     mov num1, eax ;num1++;
29 ;Espacio para las instrucciones del ciclo For
30     loop L1 ; Seria como }For(condicion: JNZ L1);
31 ;-----Alcance del ciclo for-----
32     invoke ExitProcess, 0
33 main ENDP
34 END main

```

No pierda de vista los registros EAX y ECX, durante la ejecución de las instrucciones:



En la sección (A superior) podemos ver cómo el registro EAX va incrementando su valor a la vez que envía su contenido a `num1` en (A inferior), mientras que ECX va reduciendo su valor en (B), la ventaja de usar este esquema es que es más directo y requiere menos instrucciones.

Conclusiones:

El lenguaje ensamblador a pesar de contar con un conjunto reducido de instrucciones puede ejecutar las operaciones de alto nivel mediante la traducción que realizan de manera automática los compiladores, es posible también extender el conjunto de instrucciones para facilitar el trabajo de codificación al programador, para poder usar la mayoría de las instrucciones del lenguaje, se deberá realizar una configuración previa colocando ciertos valores en los registros, lo cual puede llevar a que se terminen la cantidad de registros de propósito general de los que dispone el CPU, para solventar esta tarea, se pueden guardar y recuperar los valores de los registros durante la ejecución de un programa, este tipo de técnicas así como instrucciones más avanzadas se deja para un curso más avanzado de programación el lenguaje ensamblador.





Anexo A

configurando un ambiente para trabajar en Visual Studio 2022

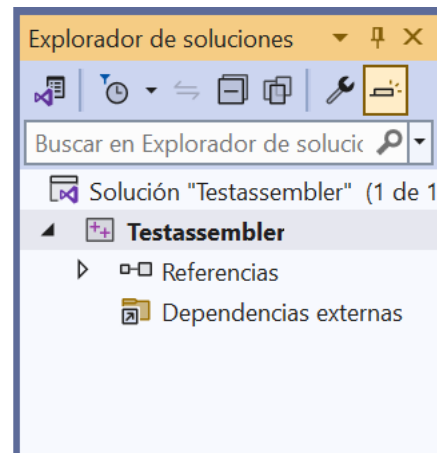
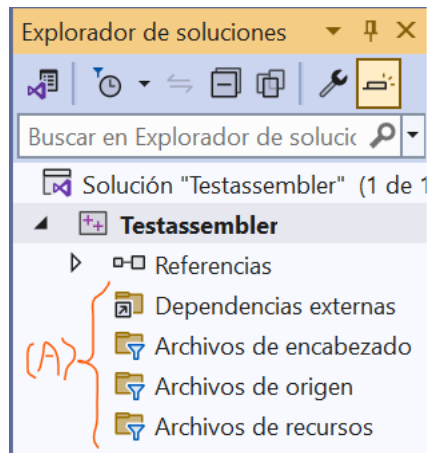
Comenzar a trabajar en un ambiente de diseño para lenguaje ENSAMBLADOR, en Visual Studio Requerimos "Desnudar" un ambiente de C++ un lenguaje de Alto Nivel. Comencemos por crear un proyecto vacío:

Crear un proyecto

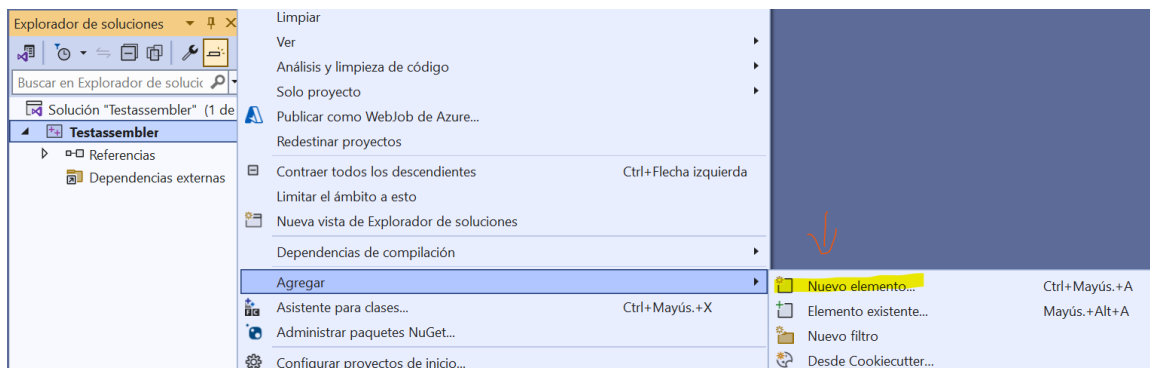
Plantillas de proyecto recientes

 Proyecto vacío	C++
 Aplicación de Python	Python
 Proyecto de base de datos de SQL Server	Lenguaje de consulta
 Aplicación de consola	C++

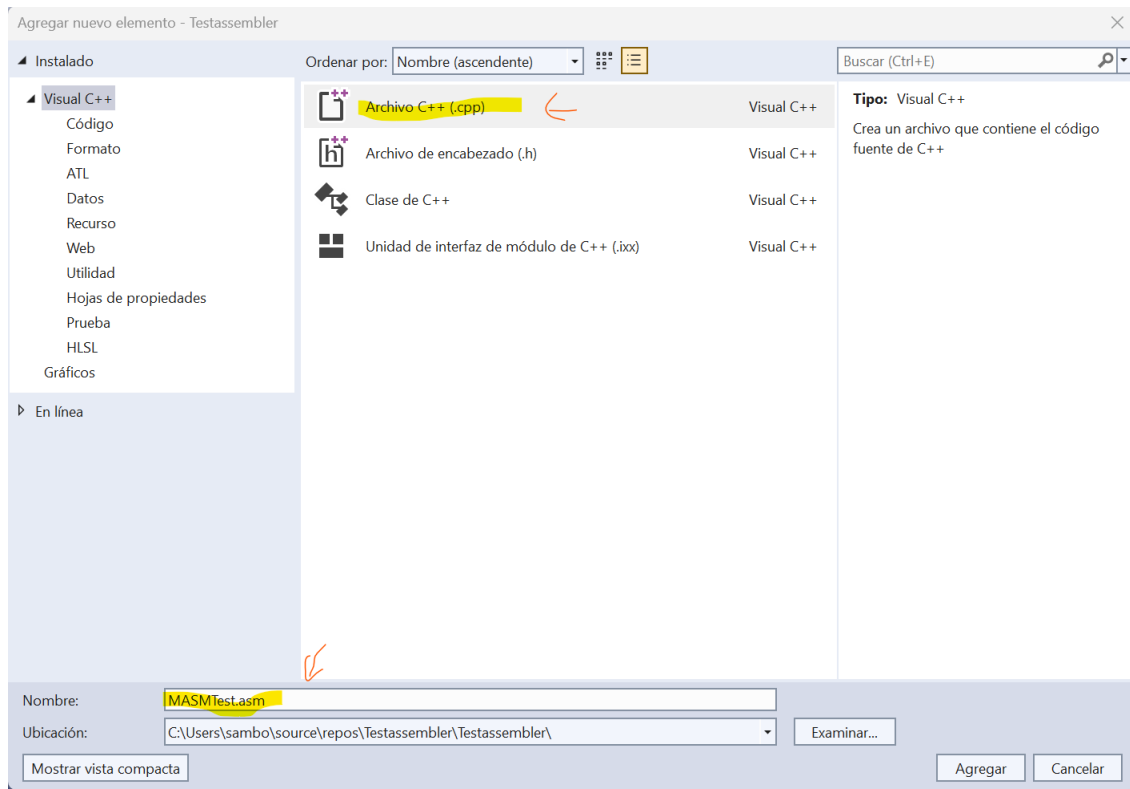
No son requeridas las carpetas (A), pueden ser removidas si se desea:



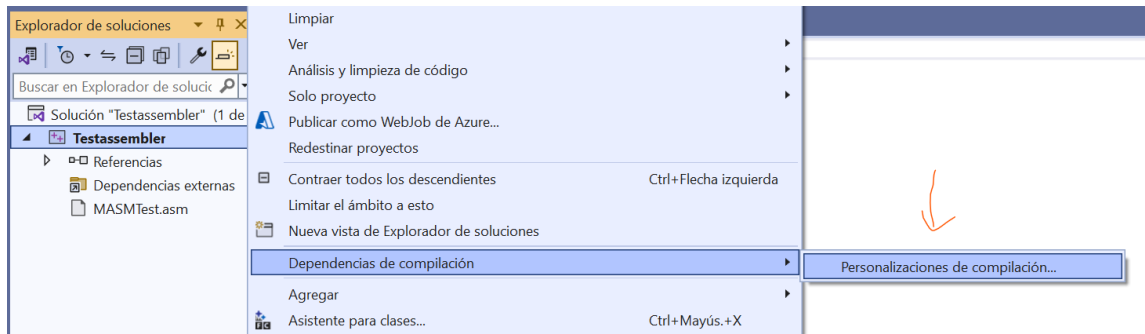
Se debe agregar un nuevo elemento:



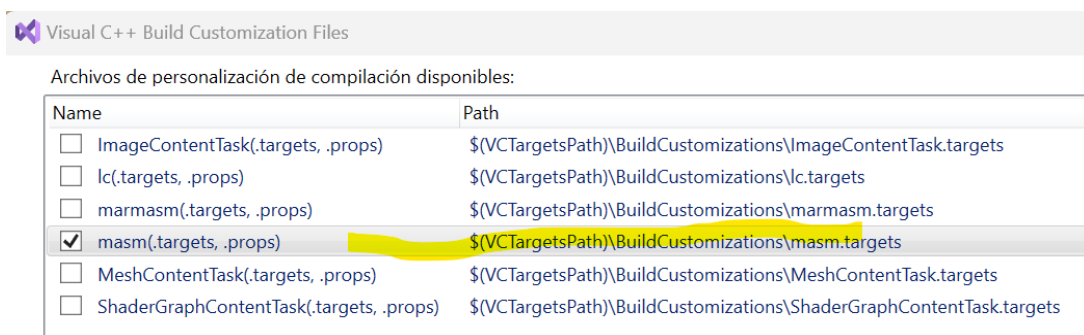
En la ventana de Nuevo elemento revise que la opción "Archivo C++(.cpp)" este activa y para el nombre del archivo agregue **.asm** como extension (si no ve la ventata, seleccione Mostrar Plantillas)



Nuevamente haga click con el botón derecho en el nombre del proyecto:

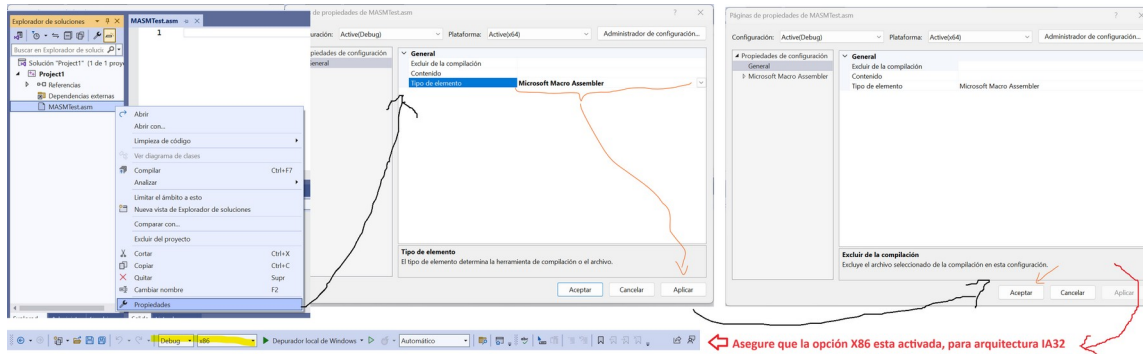


Para Personalizar las dependencias de compilación:



Seleccione **masm**, esto indicara que deseamos se use el "compilador" MASM.

Hacemos click con el boton derecho en el archivo que creamos anteriormente para agregarlo a la compilacion del proyecto:



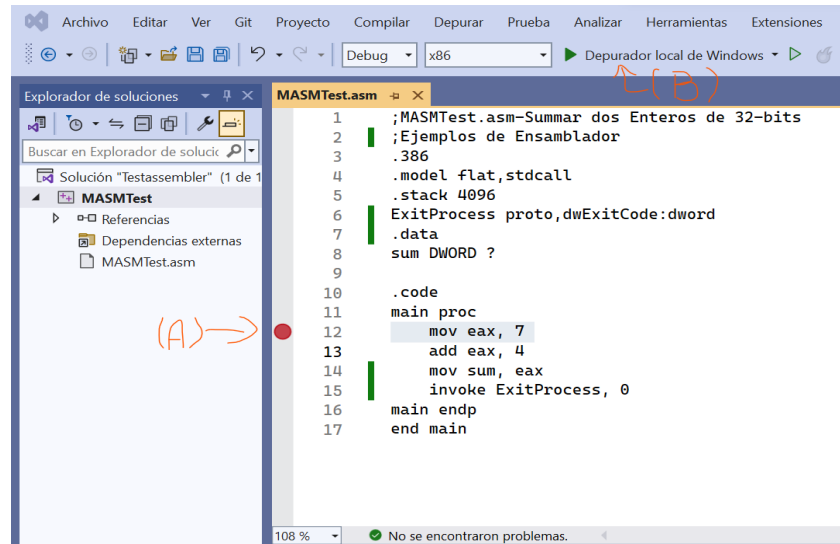
Anexo B

Manejando las herramientas de Depuracion

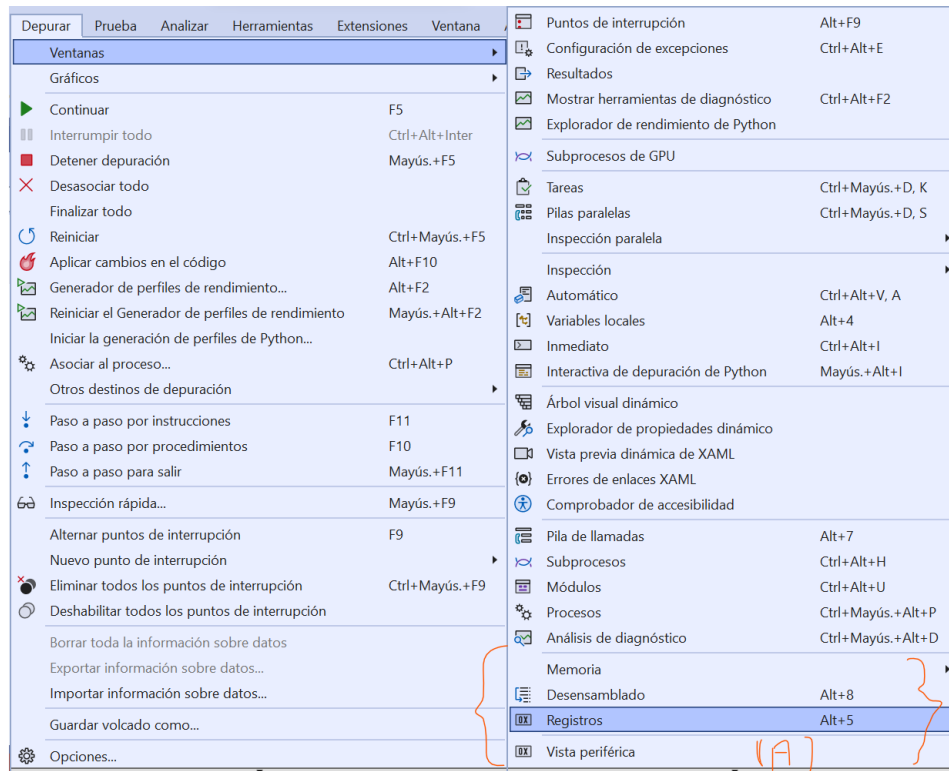
Utilizando el siguiente código de ejemplo:

```
1 ;MASMTest.asm-Summar dos Enteros de 32-bits
2 ;Ejemplos de Ensamblador
3 .386
4 .model flat,stdcall
5 .stack 4096
6 ExitProcess proto,dwExitCode:dword
7 .data
8 sum DWORD ?
9
10 .code
11 main proc
12     mov eax, 7
13     add eax, 4
14     mov sum, eax
15     invoke ExitProcess, 0
16 main endp
17 end main
```

El código anterior suma los números 7 y 4 almacenando el resultado en la variable `sum`, realizaremos una ejecución paso a paso de la ejecución del programa e iremos revisando el cambio en los registros de proposito general y la memoria de nuestro programa haciendo uso de las herramientas de depuración.

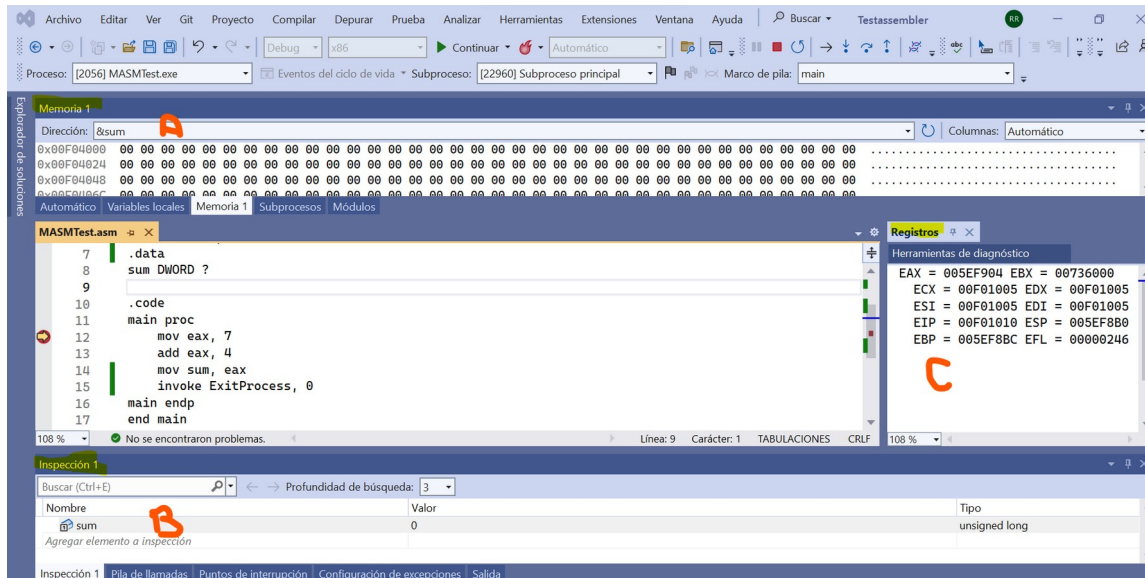


Comience por colocar un punto de depuración haciendo click en la parte derecha de la ventana de edición de código (A), posteriormente pulse la tecla PLAY para iniciar el depurador (B), el cual se detendrá justo antes de ejecutar la línea 12 marcada por el punto de depuración.

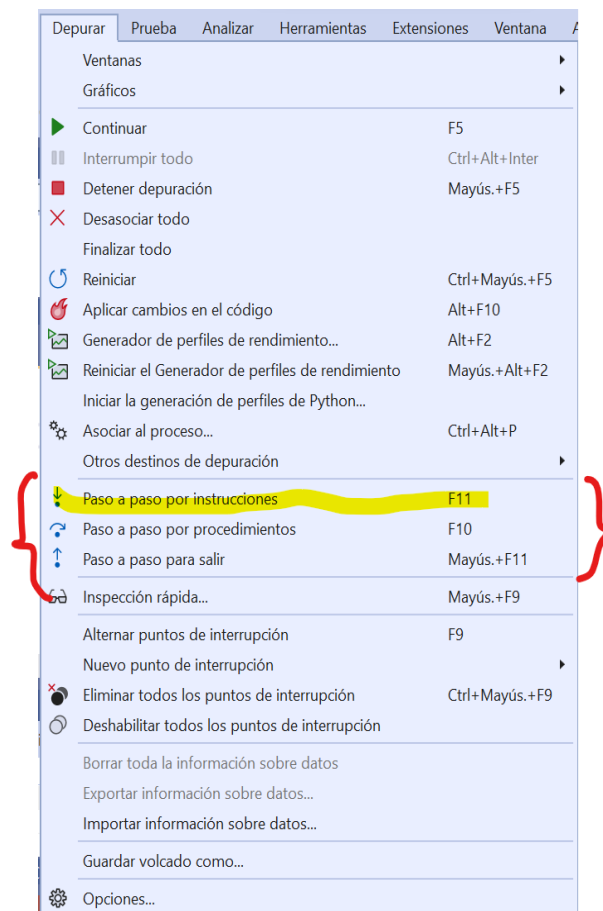


En la pestaña de Depuración (Debug), abra el menú de Ventanas (Windows) donde encontrará en (A) las ventanas que nos ayudarán a visualizar lo que ocurre con nuestro programa.

En la siguiente imagen podemos ver las tres ventanas que nos serán de utilidad *Memoria*, *Registros* e *Inspección*.

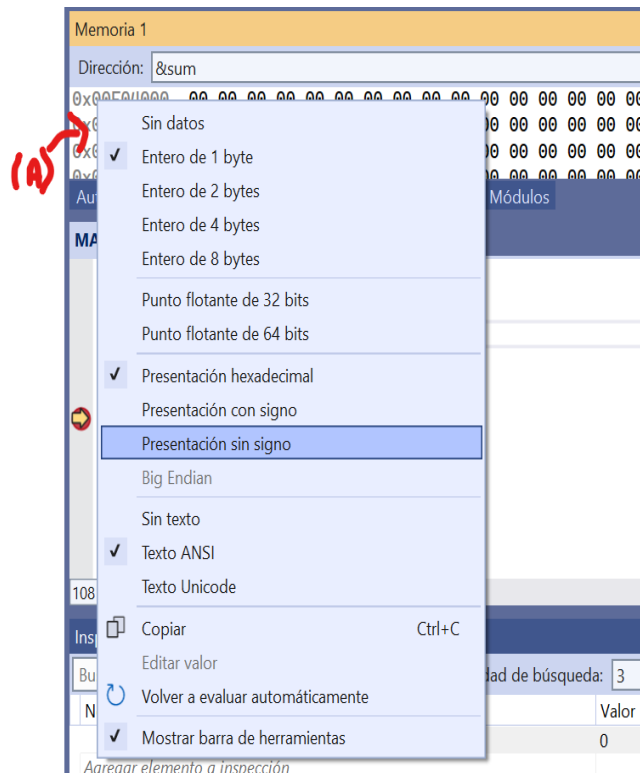


En la seccion (A) escribimos &sum para ir a la seccion de memoria que mantiene resguardado el valor de la variable sum (definida en la linea 8), en (B) escribimos el nombre de la variable que deseamos observar y en (C) podemos apreciar el contenido de los registro conforme nos movamos en la ejecución del programa (si desea ver un desgloce del registro de banderas EFL, click boton derecho y seleccione la opcion marcas).

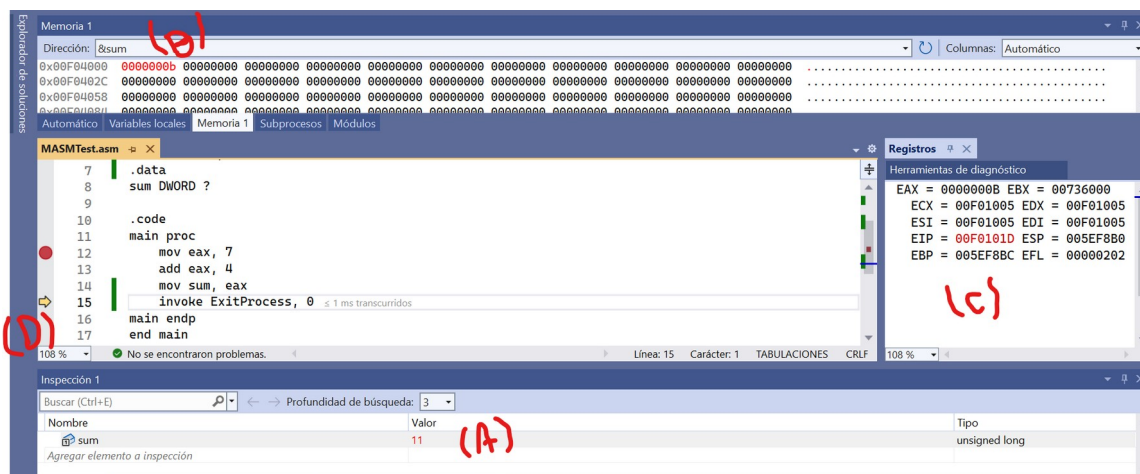


Utilizando la opción paso a paso (tecla de acceso rápido F11) puede ir ejecutando las instrucciones una a la vez mientras observa el cambio los cambios reflejados en las ventanas de las herramientas de depuración.

Es posible usar opciones de los menús que proporcionan las ventanas de depuración para cambiar la forma en la que se representan los datos, por ejemplo en la ventana de *Memoria*:



Si hacemos click derecho en el área del vacío de la memoria, podemos desplegar el menú que nos permite cambiar de la forma hexadecimal a la decimal y agrupar las regiones de memoria acorde a los tamaños que sean significativos durante la ejecución del programa, en este caso convendría usar la representación de 4 bytes debido a que la variable suma es DWORD (dos palabras dobles) y tiene exactamente el tamaño perfecto para representar el dato.



Tras la ejecución de algunas instrucciones de nuestro programa (D), podemos ver en (A) que la variable *sum* ya cuenta con el valor 11 (resultado de sumar 7 y 4). En (B) podemos apreciar el equivalente en Hexadecimal en la memoria, y como en (C) el registro EAX contiene el mismo valor Hexadecimal de nuestra suma .

Debido a que el Lenguaje Ensamblador es muy Basico y muy cercano a como trabaja el Hardware, no contamos con Salidas (ni entradas) tan avanzadas como las que nos presentan los lenguajes de alto nivel, para tener estas facilidades habra que programarlas e incluirlas en librerias para su posterior uso, consulte el Anexo C para más información.

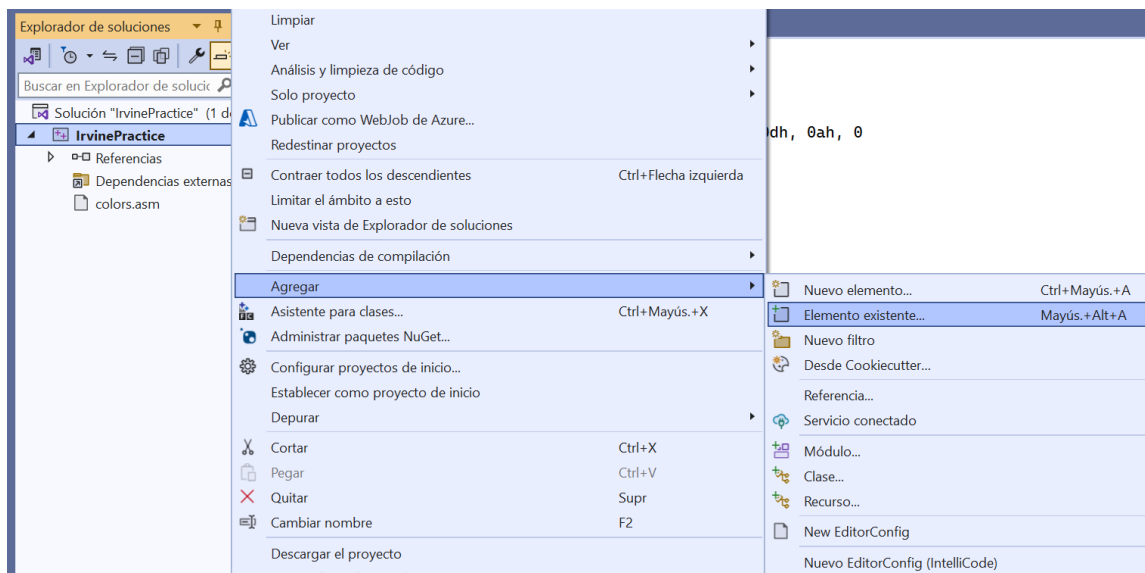
Anexo C

Anexando la libreria Irvine32 de apoyo

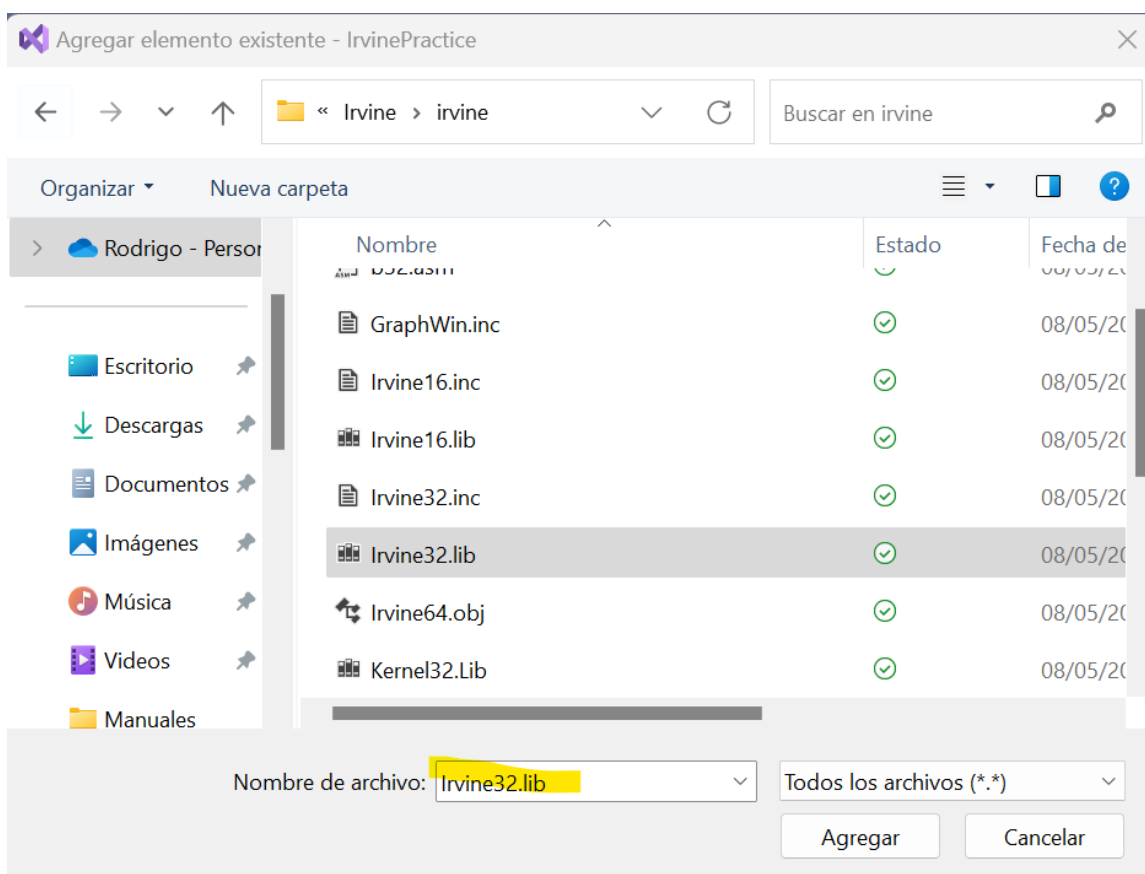
Usando el siguiente código de ejemplo que se sirve de las librerias de Irvine32:

```
TITLE add and Subtract (AddSubAlt.asm)
;This programa adds and subtracts 32-bit integers.
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO
WriteString PROTO
.data
str1 BYTE "Cadena de Muestra", 0dh,0ah, 0
.code
main PROC
    mov edx,OFFSET str1
    call WriteString
    call DumpRegs
    mov eax,10000h ; EAX = 10000h
    add eax,40000h ; EAX = 50000h
    sub eax,20000h ; EAX = 30000h
    call DumpRegs
    INVOKE ExitProcess, 0
main ENDP
END main
```

Hacemos click con el boton derecho en el nombre del proyecto:



Buscamos por el archivo con extension .lib, el de 32 bits:



Ahora si podemos ejecutar el archivo:

```
1  TITLE add and Substract (AddSubAlt.asm)
2  ;This programa adds and substracts 32-bit integers.
3  .386
4  .model flat,stdcall
5  .stack 4096
6  ExitProcess PROTO, dwExitCode:DWORD
7  DumpRegs PROTO
8  WriteString PROTO
9  .data
10 str1 BYTE "Cadena de Muestra", 0dh,0ah, 0
11 .code
12 main PROC
13 mov edx,OFFSET str1
14 call WriteString
15 call DumpRegs
16 mov eax,10000h ; EAX = 10000h
17 add eax,40000h ; EAX = 50000h
18 sub eax,20000h ; EAX = 30000h
19 call DumpRegs
20 INVOKE ExitProcess, 0
21 main ENDP
22 END main
```

Y obtenemos la salida:

```
Consola de depuración de Microsoft Visual Studio

Cadena de Muestra

EAX=0073F9A0  EBX=0054C000  ECX=00A810AA  EDX=00A86000
ESI=00A810AA  EDI=00A810AA  EBP=0073F958  ESP=0073F94C
EIP=00A8366B  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0

EAX=00030000  EBX=0054C000  ECX=00A810AA  EDX=00A86000
ESI=00A810AA  EDI=00A810AA  EBP=0073F958  ESP=0073F94C
EIP=00A8367F  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1

C:\Users\sambo\source\repos\EnsambladorPracticas\Debug\EnsambladorPracticas.exe (proceso 2160) se cerró con el código 0.
Para cerrar automáticamente la consola cuando se detiene la depuración, habilite Herramientas -> Opciones -> Depuración -> Cerrar la consola automáticamente al detenerse la depuración.
Presione cualquier tecla para cerrar esta ventana. . .
```

nota: puede que necesite desactivar temporalmente su antivirus para poder incluir la libreria Irvine32.lib

BIBLIOGRAFIA

libro:

Programacion lenguaje ensamblador para computadoras basadas en Intel

autor Kip R. Irvine, sitio web del libro para configurar visualstudio

<https://github.com/surferkip/asmbook>

Sitio Web del libro para obtener la libreria

<http://asmirvine.com/gettingStartedVS2019/index.htm>

Manual de referencia del conjunto de instrucciones de la arquitectura x86:

64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383

<https://cdrdv2-public.intel.com/671110/325383-sdm-vol-2abcd.pdf>