

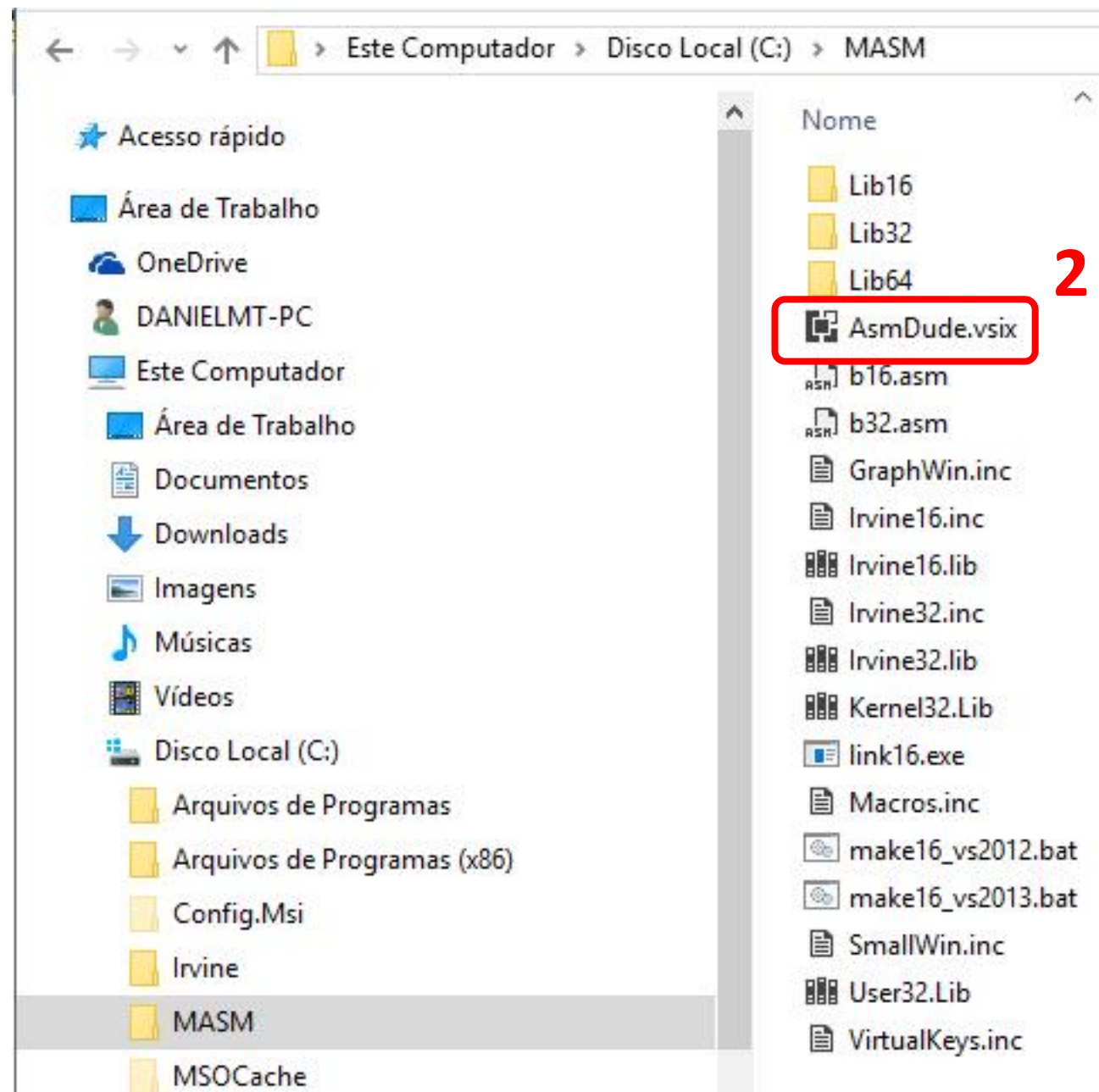
Arquitetura de Computadores

Prof. MSc. Daniel Menin Tortelli
danielmenintortelli@gmail.com

Criando um Projeto Microsoft Macro Assembly (MASM) no Microsoft Visual Studio

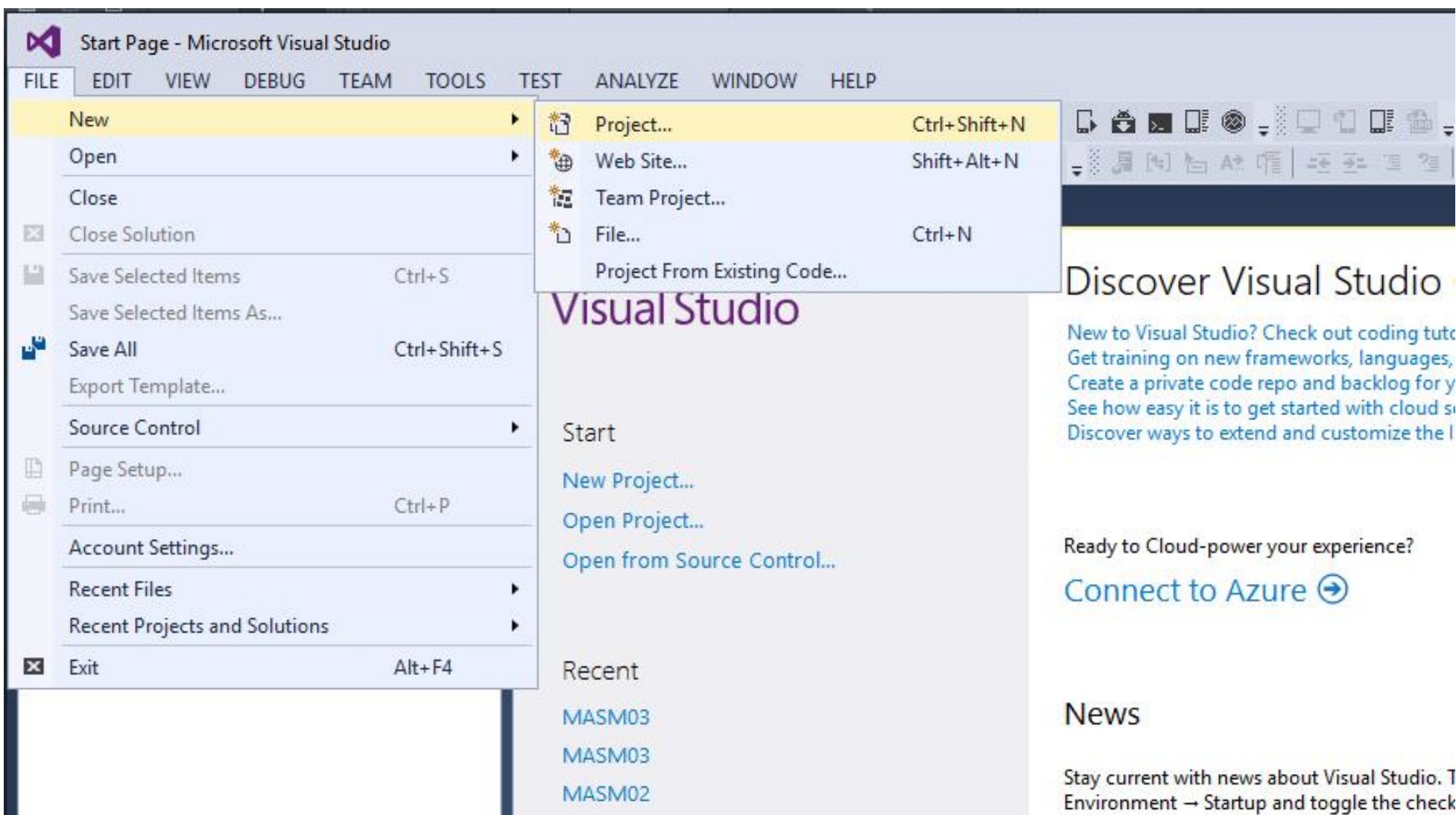
Preparativos iniciais (somente feito uma vez)

1. Descompacte o arquivo MASM.RAR na raiz do disco local (C:)
2. Dentro da pasta MASM, execute o arquivo **AsmDude.vsix**
 - Este arquivo configura o Visual Studio para reconhecer e destacar com cores a sintaxe das diversas instruções da linguagem Assembly.



Criando o projeto

3. Abra o Visual Studio e crie um novo projeto. **File > New > Project...**



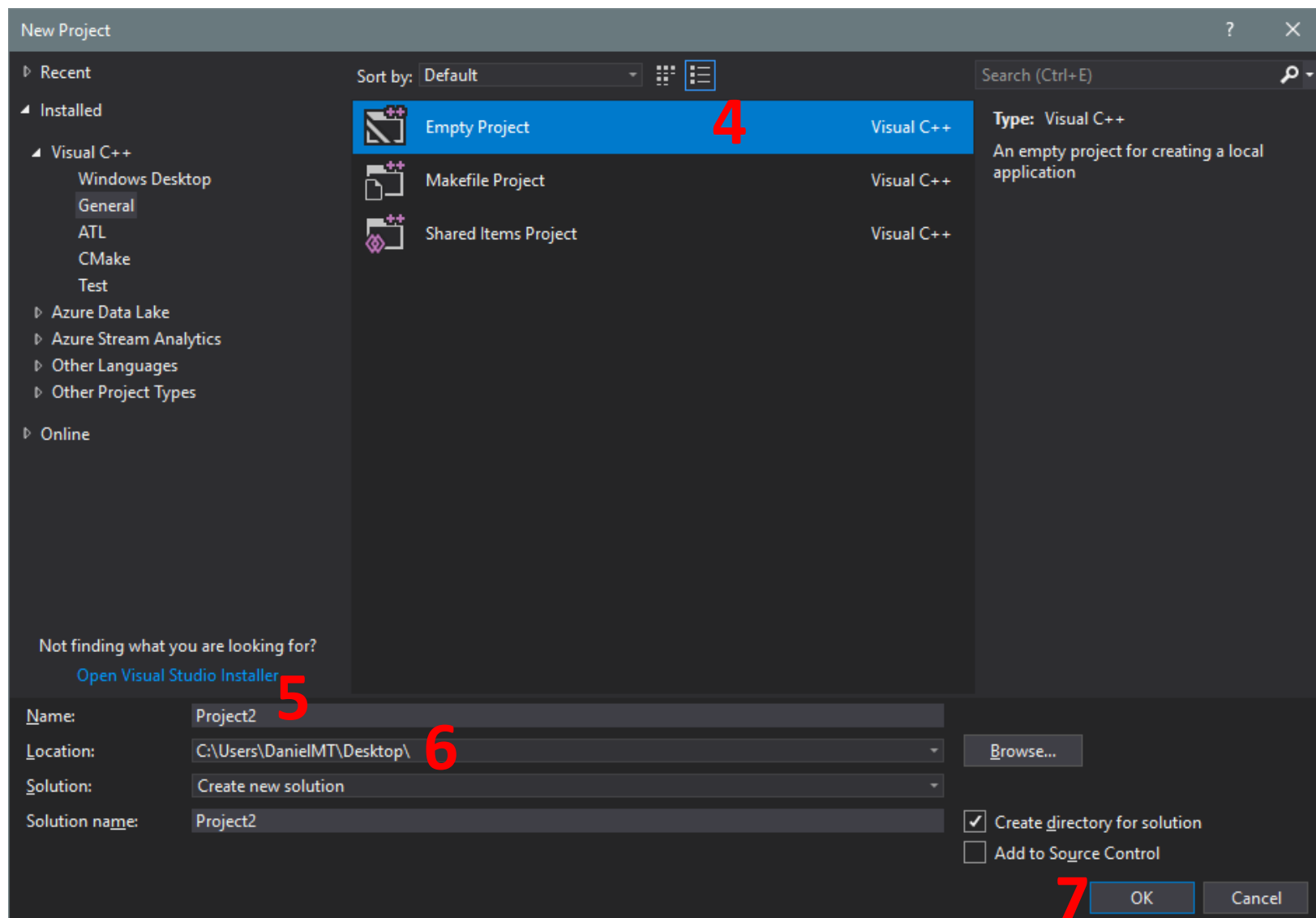
Criando o projeto

4. Crie um projeto **Visual C++ Empty Project**

5. Digite um nome para o projeto (use apenas letras e números. *Nada de espaços em branco*)

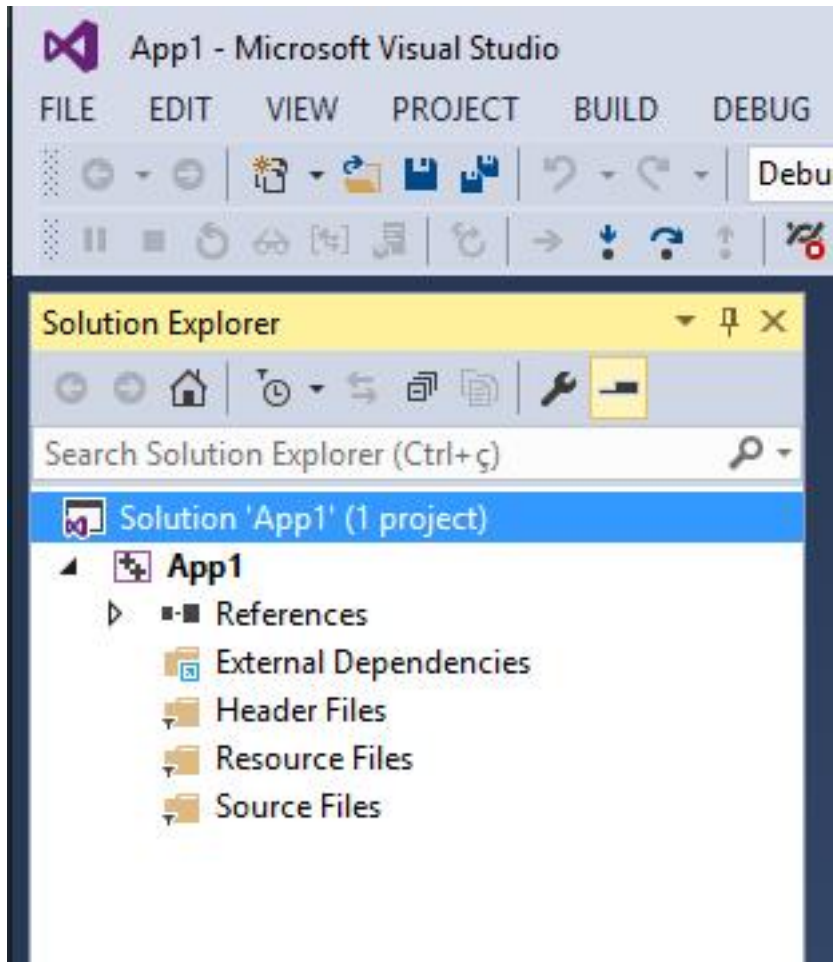
6. Escolha o local aonde o projeto será salvo.

7. Clique OK para finalizar.

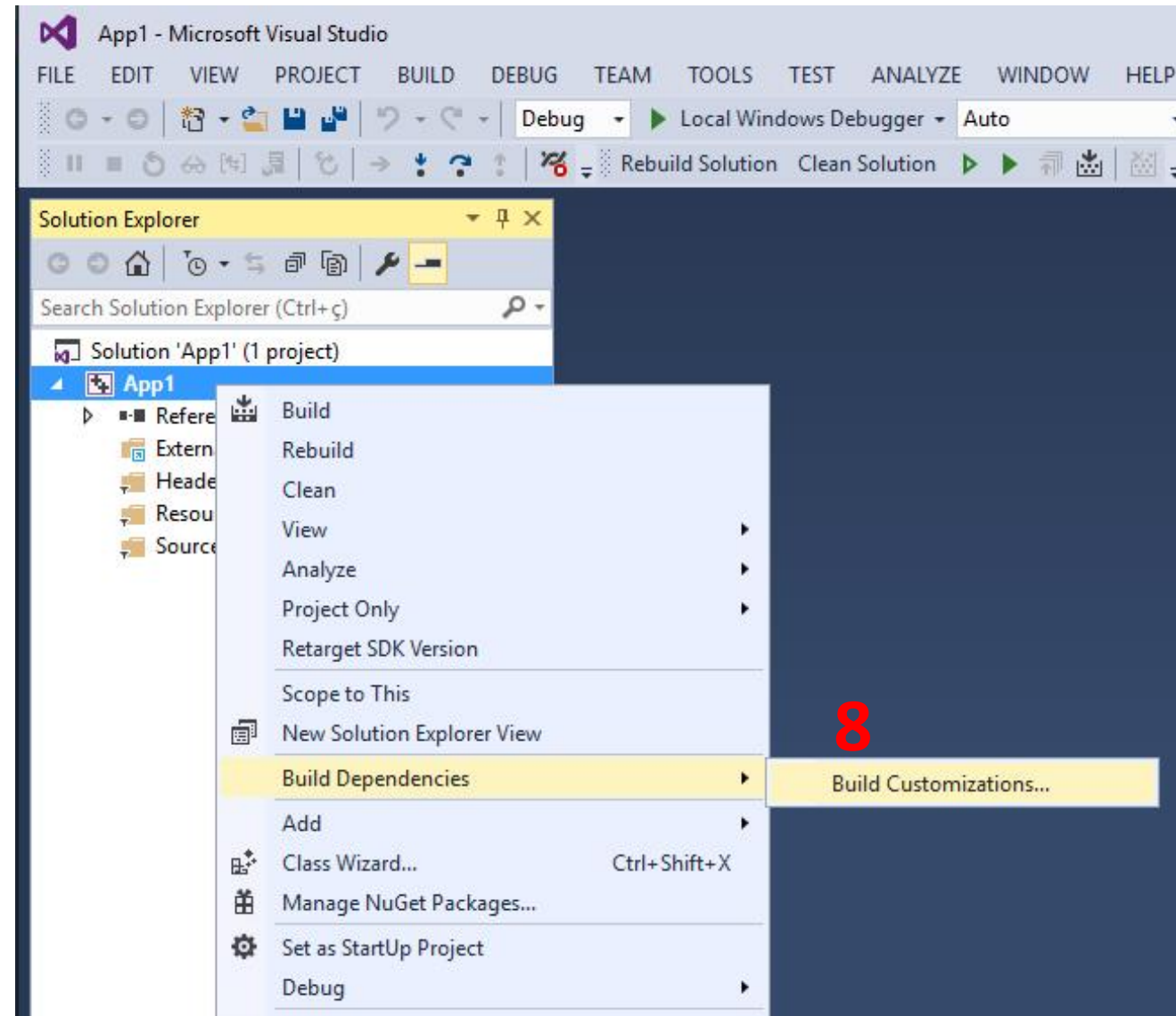


Adicionando MASM

Se o projeto foi criado com sucesso, a seguinte estrutura pe mostrada:



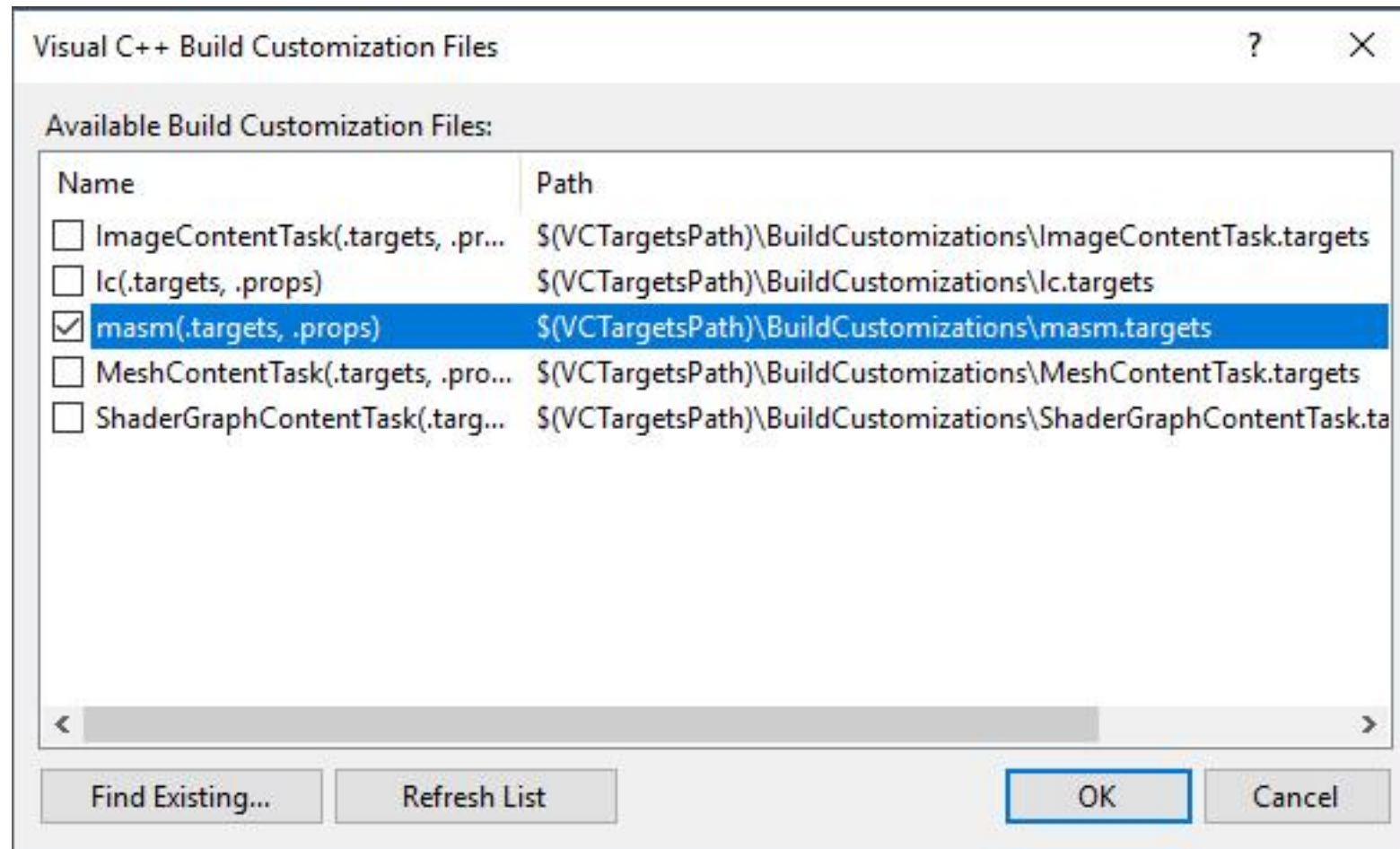
8. Clique com o botão direito do mouse sobre o nome do projeto. No menu, escolha **Build Dependencies > Build Customizations...**



Adicionando MASM

9. Adicione o **masm(.targets, .props)** como uma dependência para a compilação. Ao inserir essa dependência, o Visual Studio usará o montador MASM para compilar o código-fonte em Assembly.

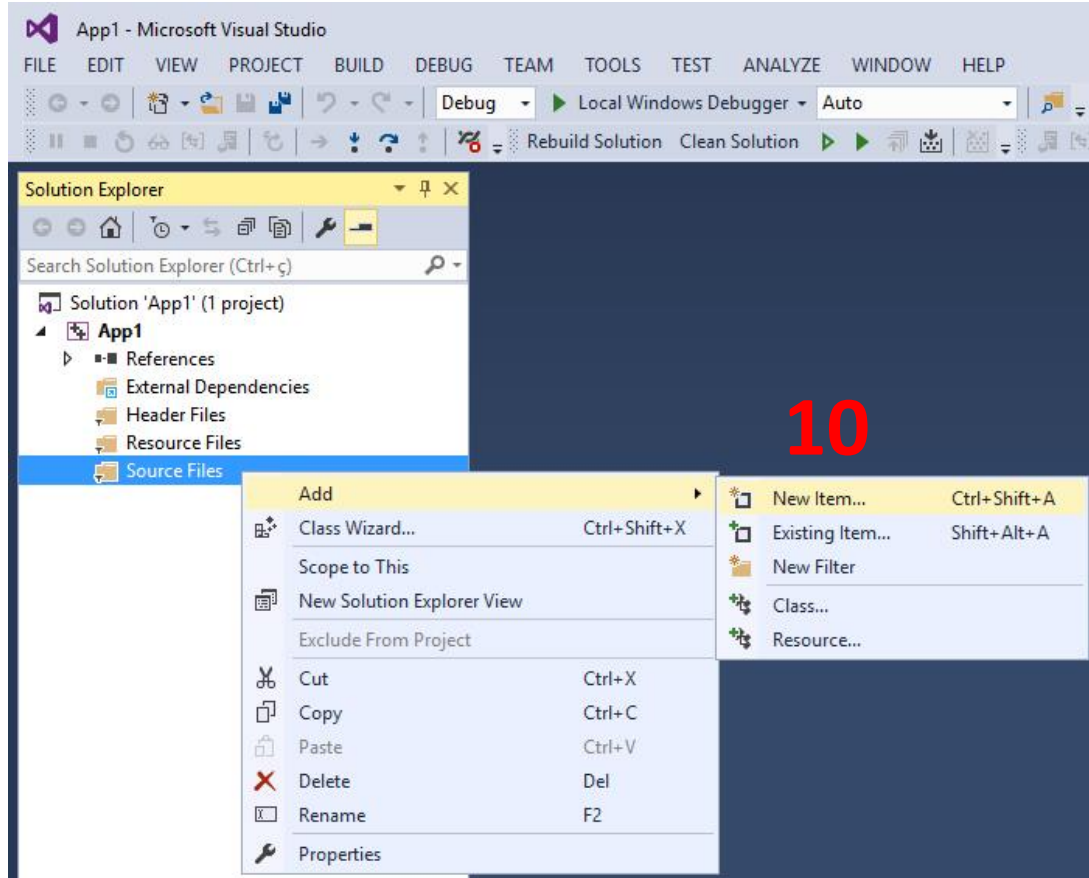
9



Criando Arquivo de Código-Fonte

10. Para adicionar o arquivo do código-fonte, clique com o botão direito sobre a pasta **Source Files**.

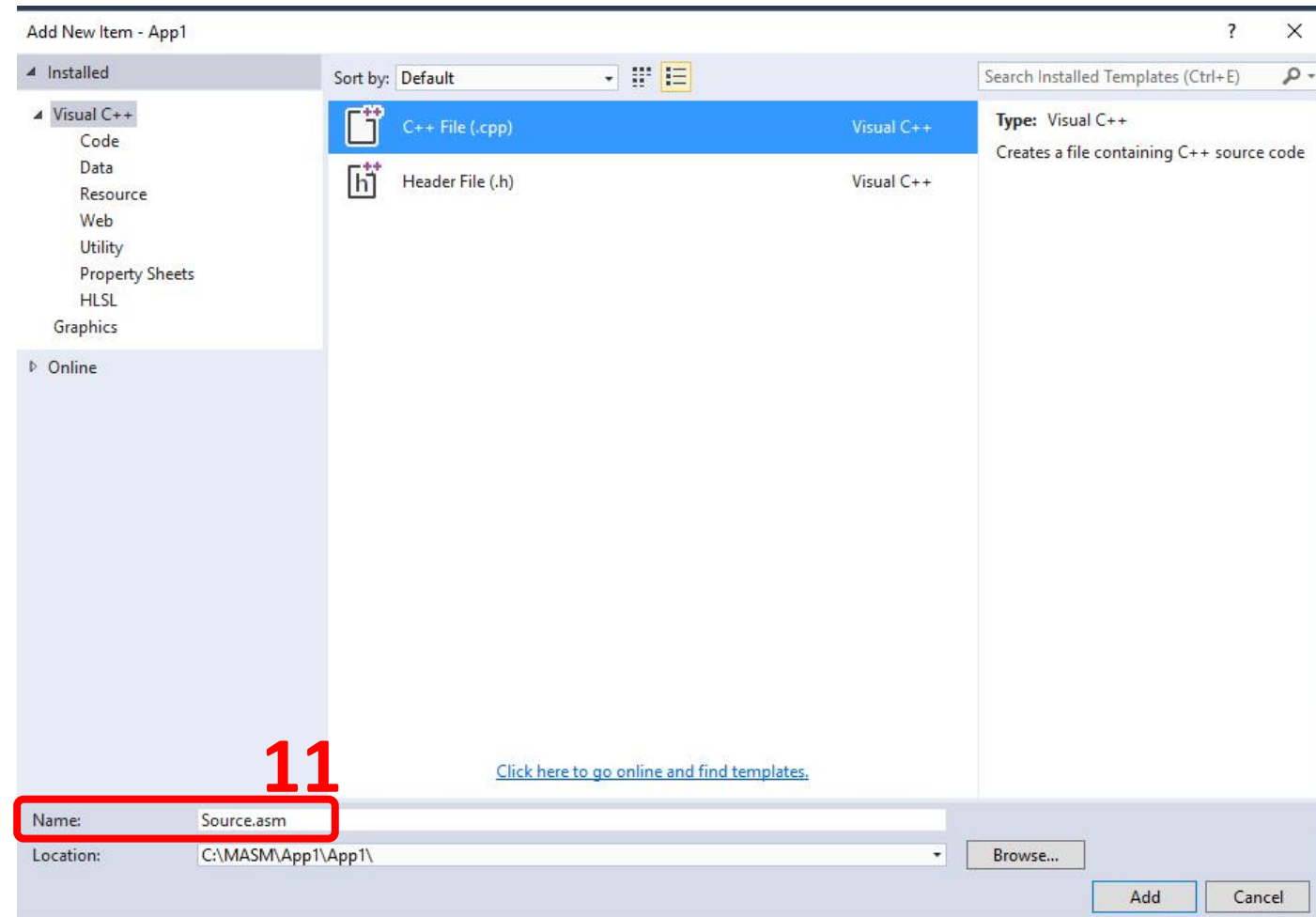
No menu, escolha **Add > New Item...**



11. **ATENÇÃO:**

Mude a extensão do arquivo Source.cpp para Source.asm

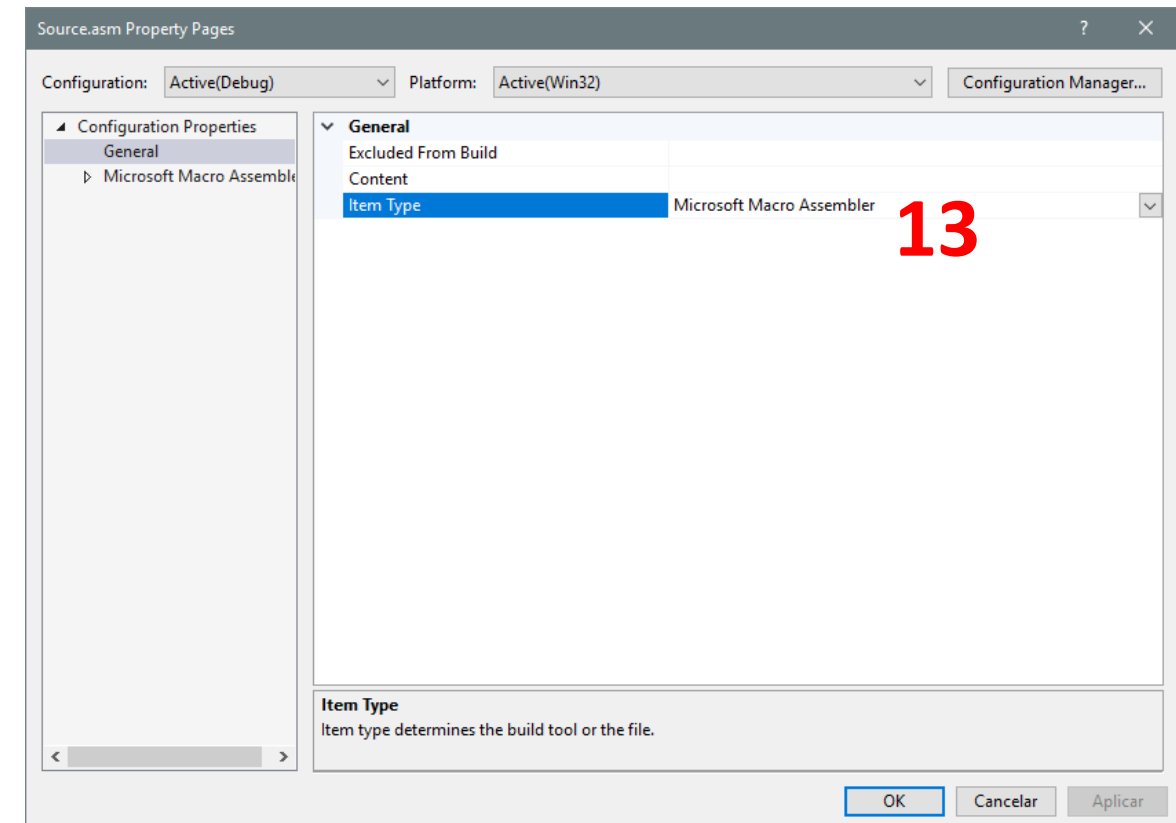
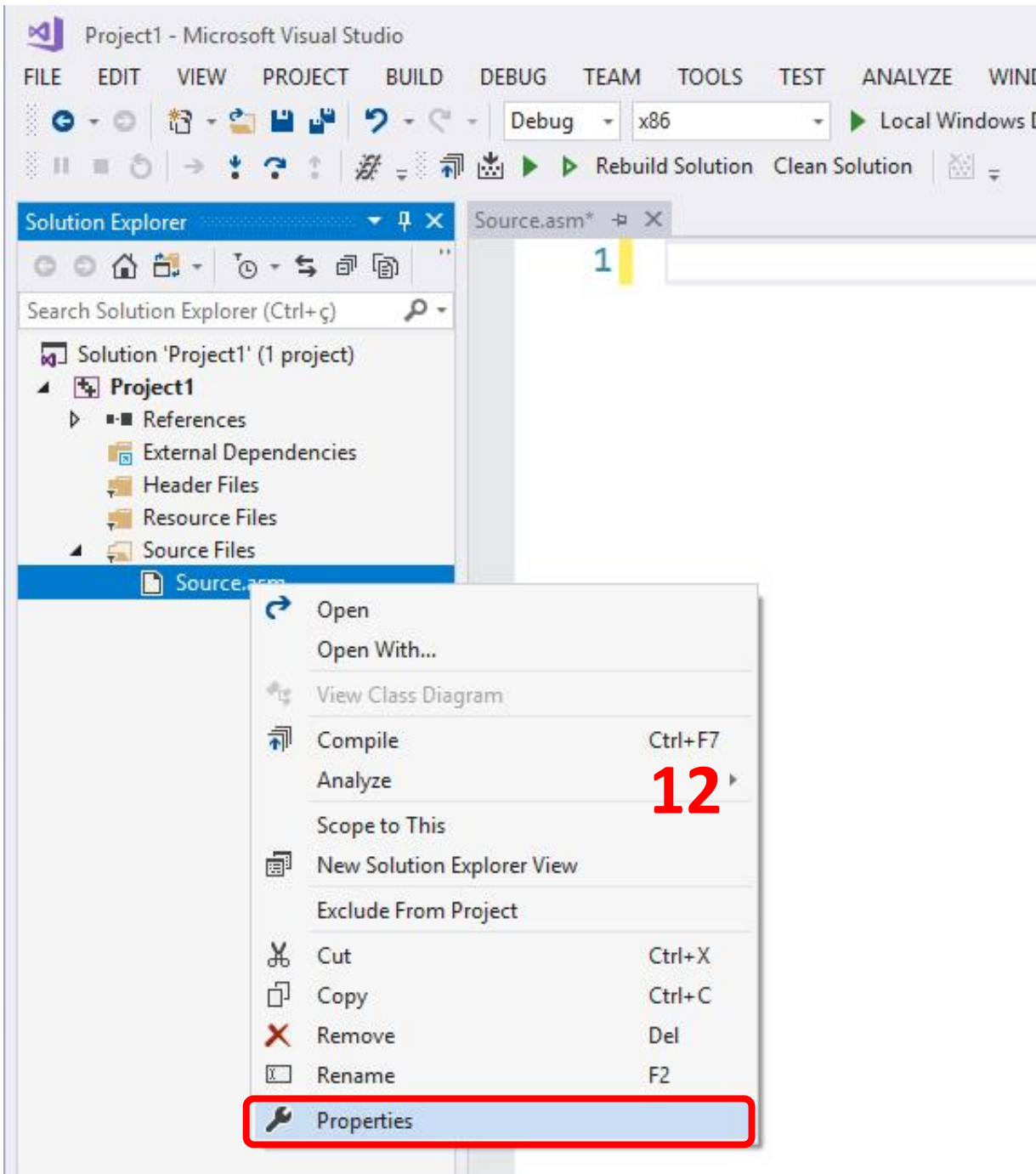
A extensão **asm** identifica um arquivo de código-fonte em Assembly.

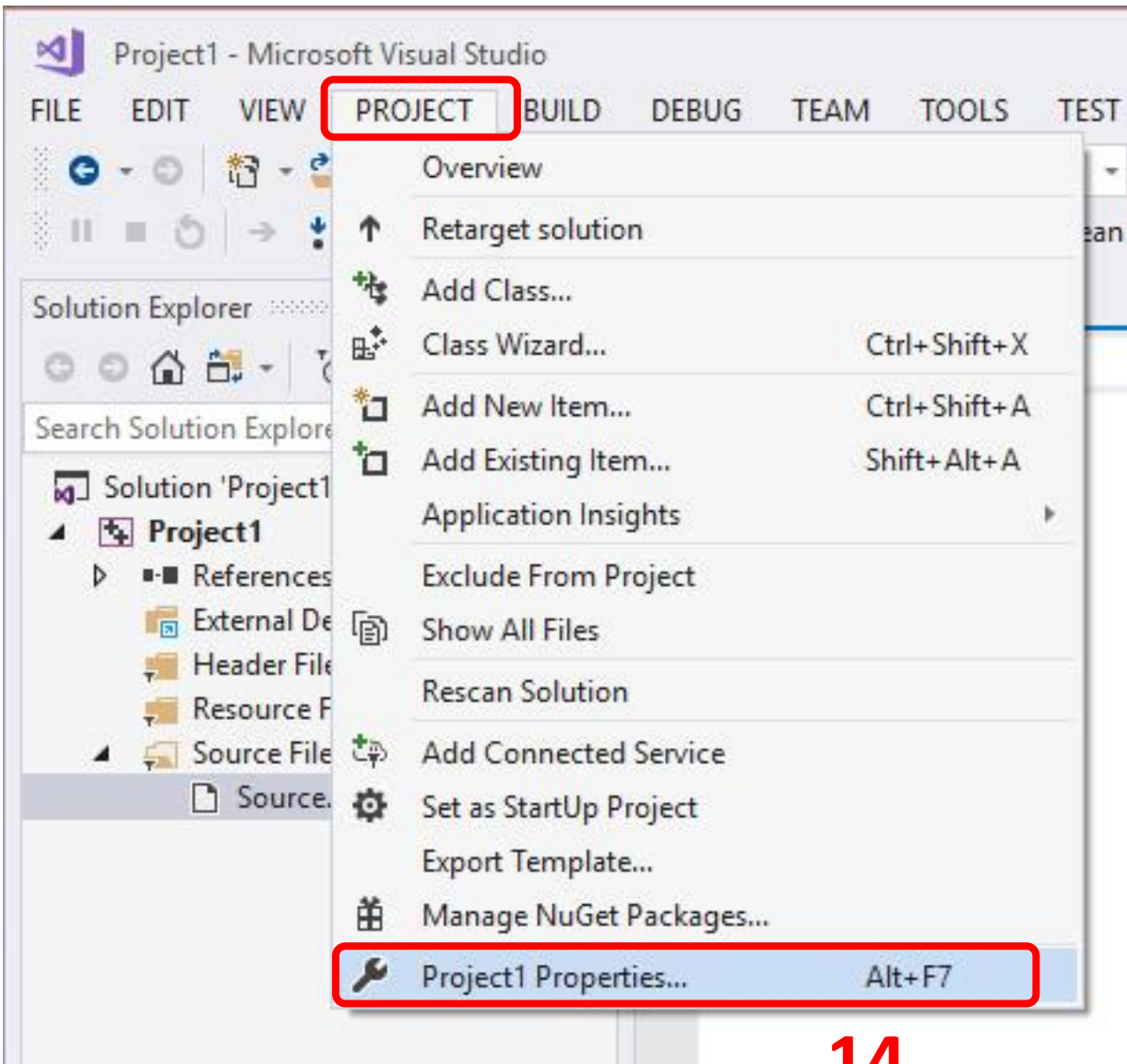


Conferir se o arquivo Source.asm será compilado em MASM

12. Clique com o botão direito sobre o arquivo **Source.asm** e escolha **Properties**

13. Verifique se **Item Type** está configurado para **Microsoft Macro Assembler**

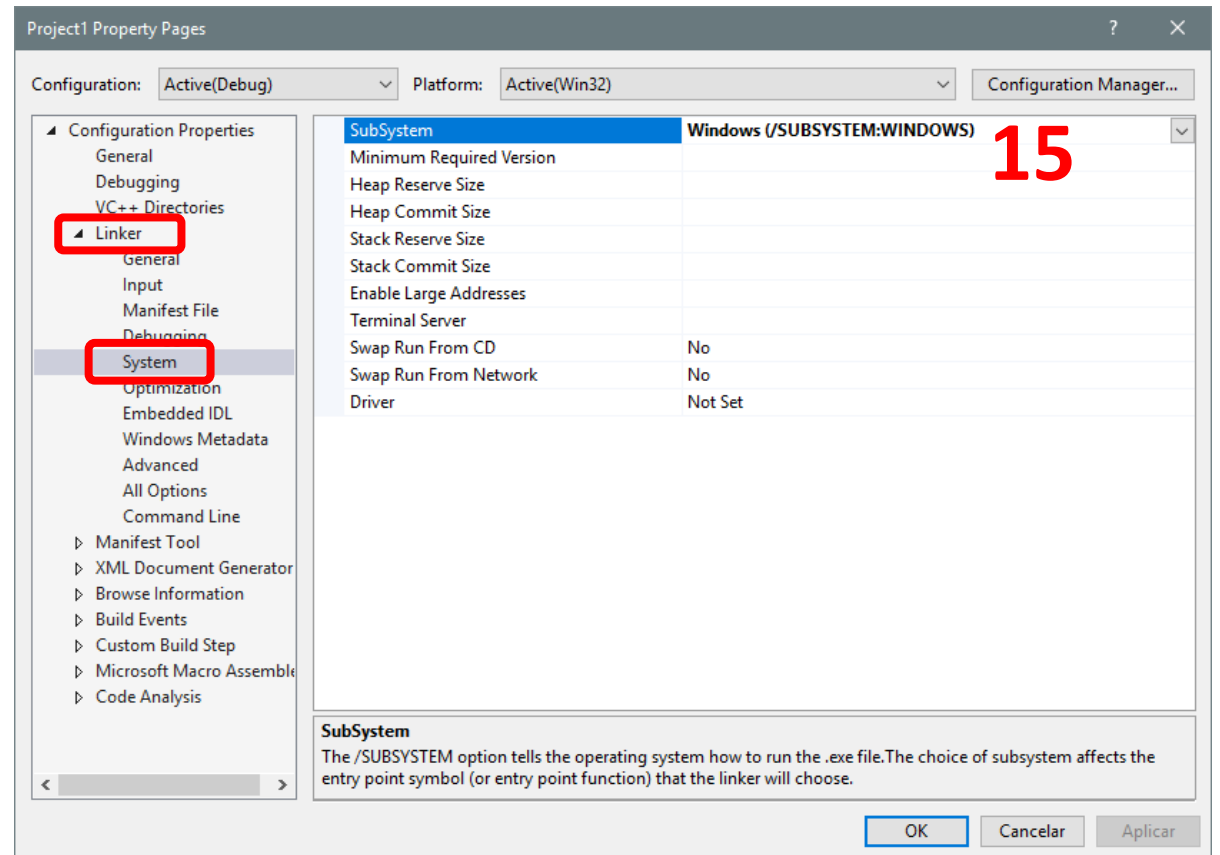




14. Clique em **PROJECT -> Properties**

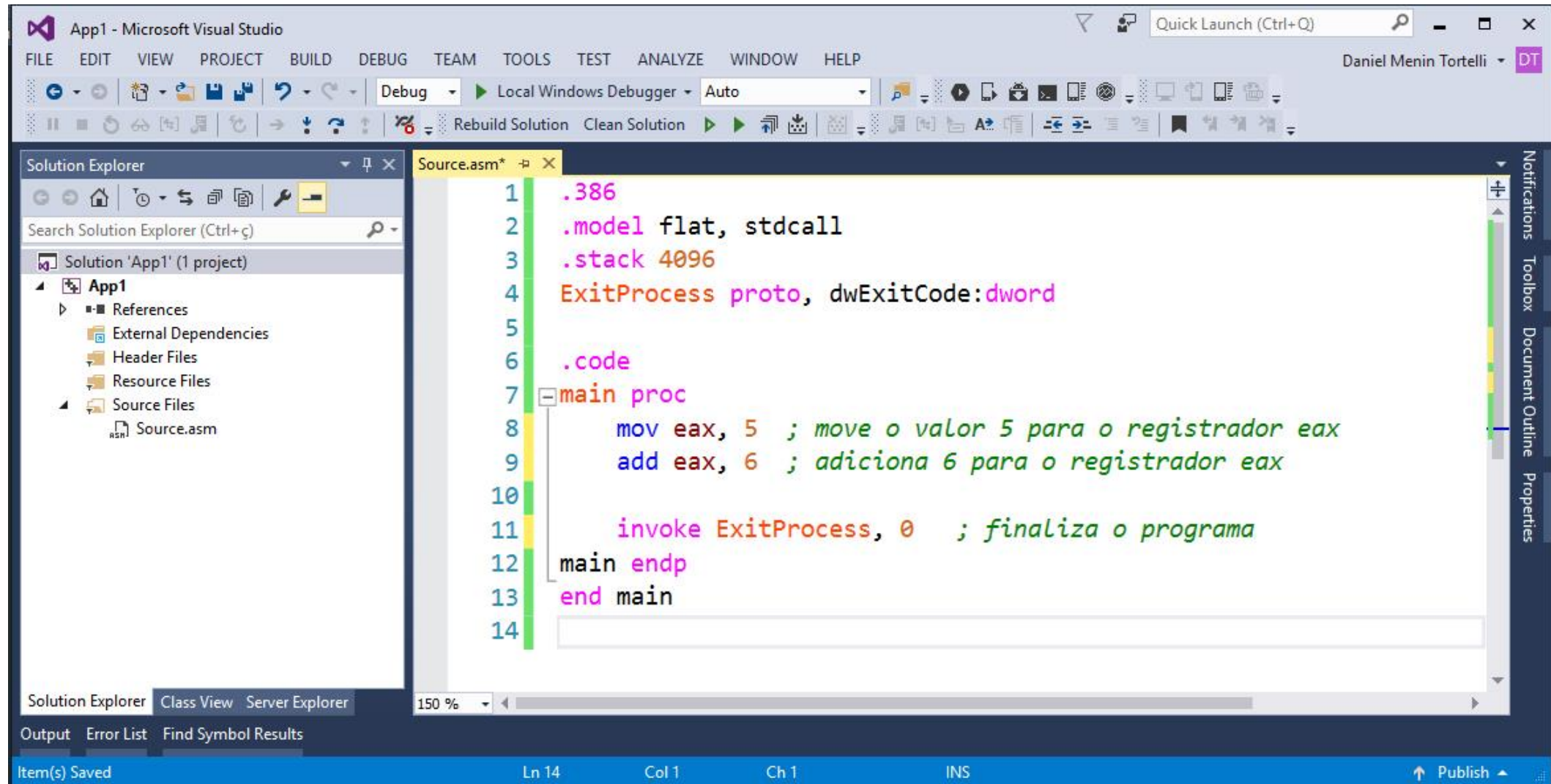
15. Em **Linker -> System -> SubSystem**, escolha a opção

Windows(/SUBSYSTEM:WINDOWS)



Criando Arquivo de Código-Fonte

Se o arquivo **asm** foi criado com sucesso, a seguinte estrutura é mostrada:



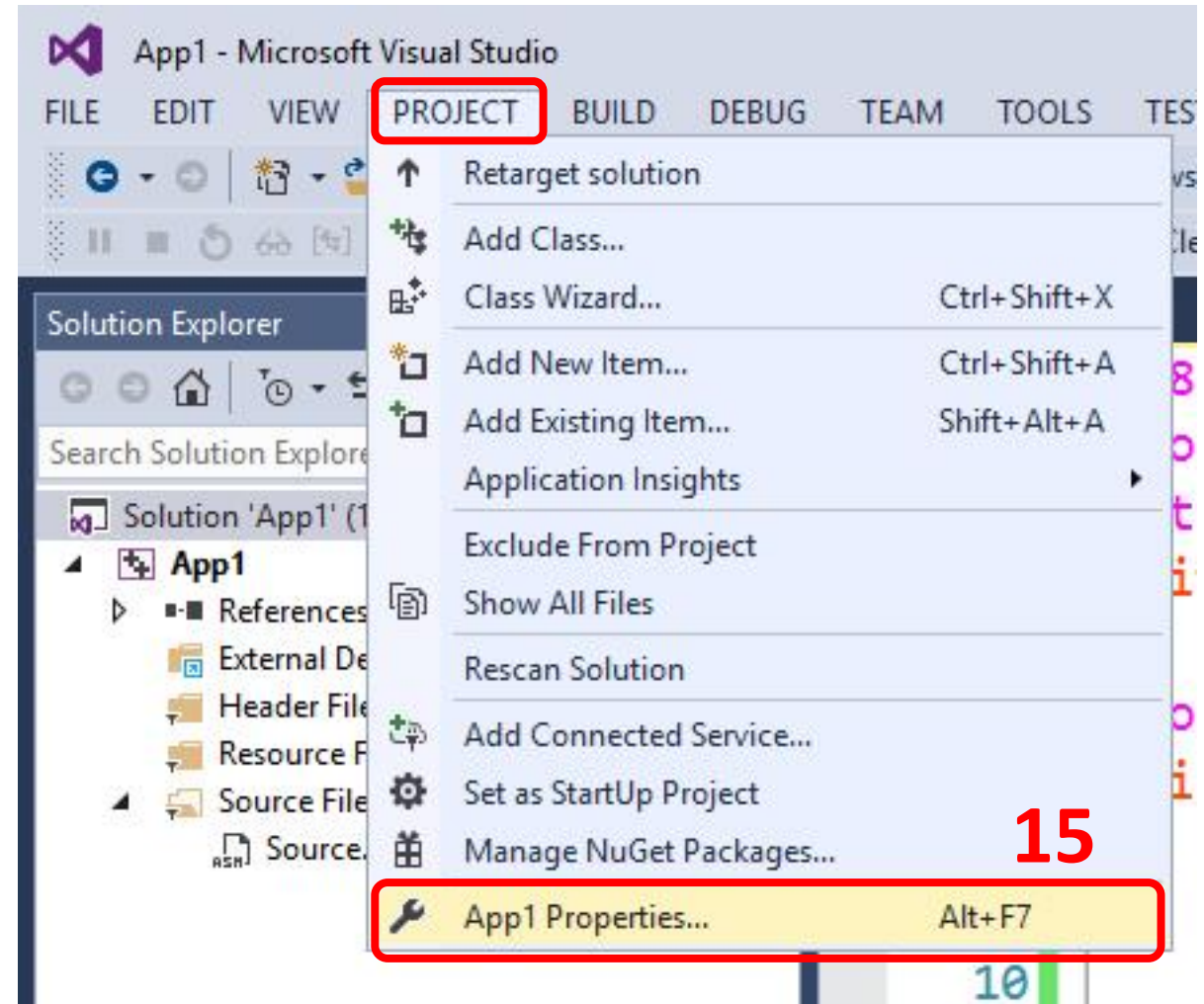
The screenshot shows the Microsoft Visual Studio interface. On the left, the Solution Explorer displays the project structure for 'App1', including 'Source Files' and 'Source.asm'. The main editor window shows the assembly code for 'Source.asm' with the following content:

```
1 .386
2 .model flat, stdcall
3 .stack 4096
4 ExitProcess proto, dwExitCode:dword
5
6 .code
7 main proc
8     mov eax, 5 ; move o valor 5 para o registrador eax
9     add eax, 6 ; adiciona 6 para o registrador eax
10
11     invoke ExitProcess, 0 ; finaliza o programa
12 main endp
13 end main
14
```

The status bar at the bottom indicates 'Item(s) Saved', 'Ln 14', 'Col 1', 'Ch 1', and 'INS'.

Configurações do Projeto (inserindo libs e paths)

15. Entre no menu **PROJECT** e escolha **App1 Properties...**

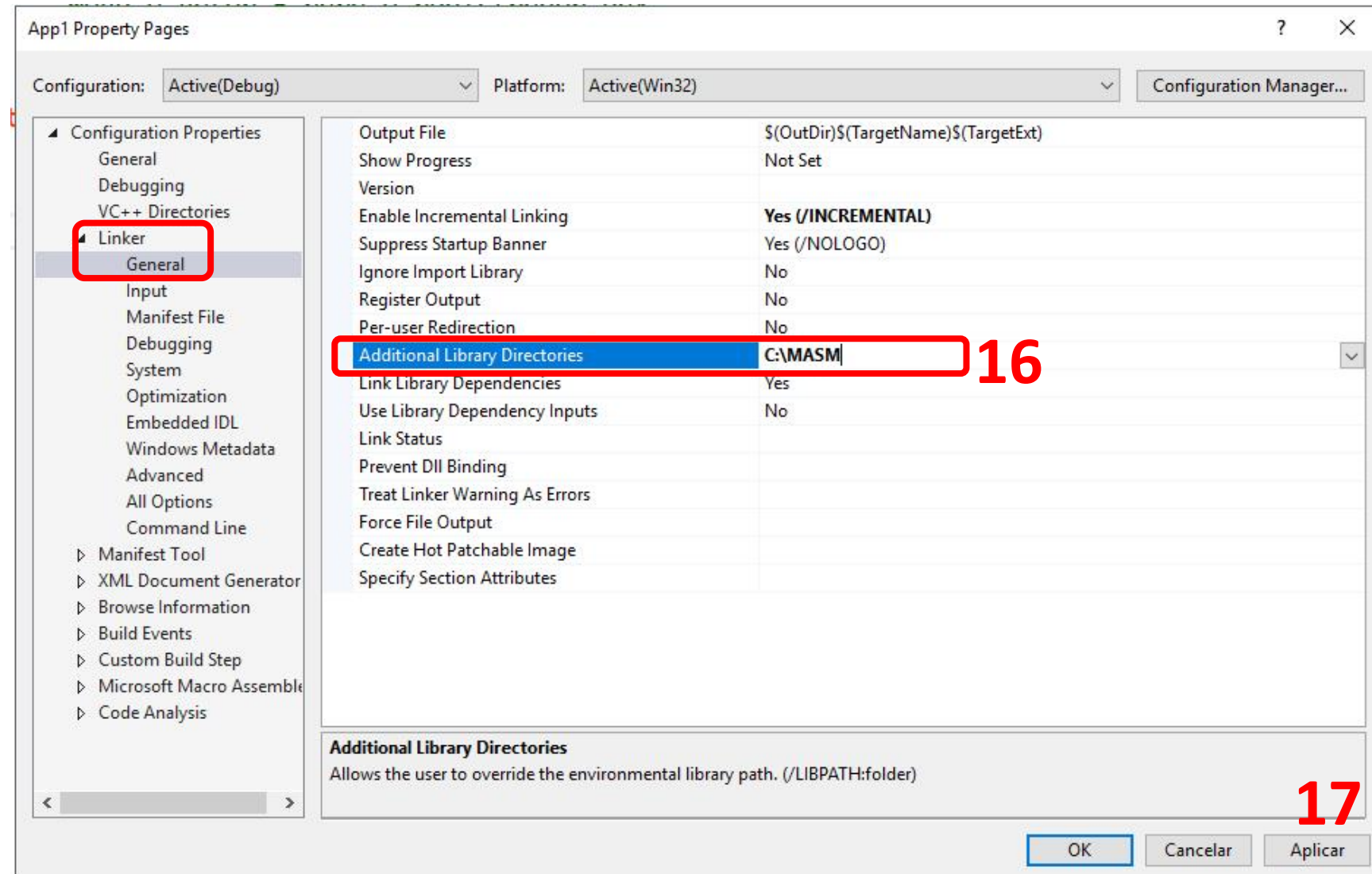


Configurações do Projeto (inserindo libs e paths)

16. Em **Linker (General)**, adicione o path das bibliotecas que serão ligadas ao programa.

C:\MASM

17. Clique em **Aplicar**

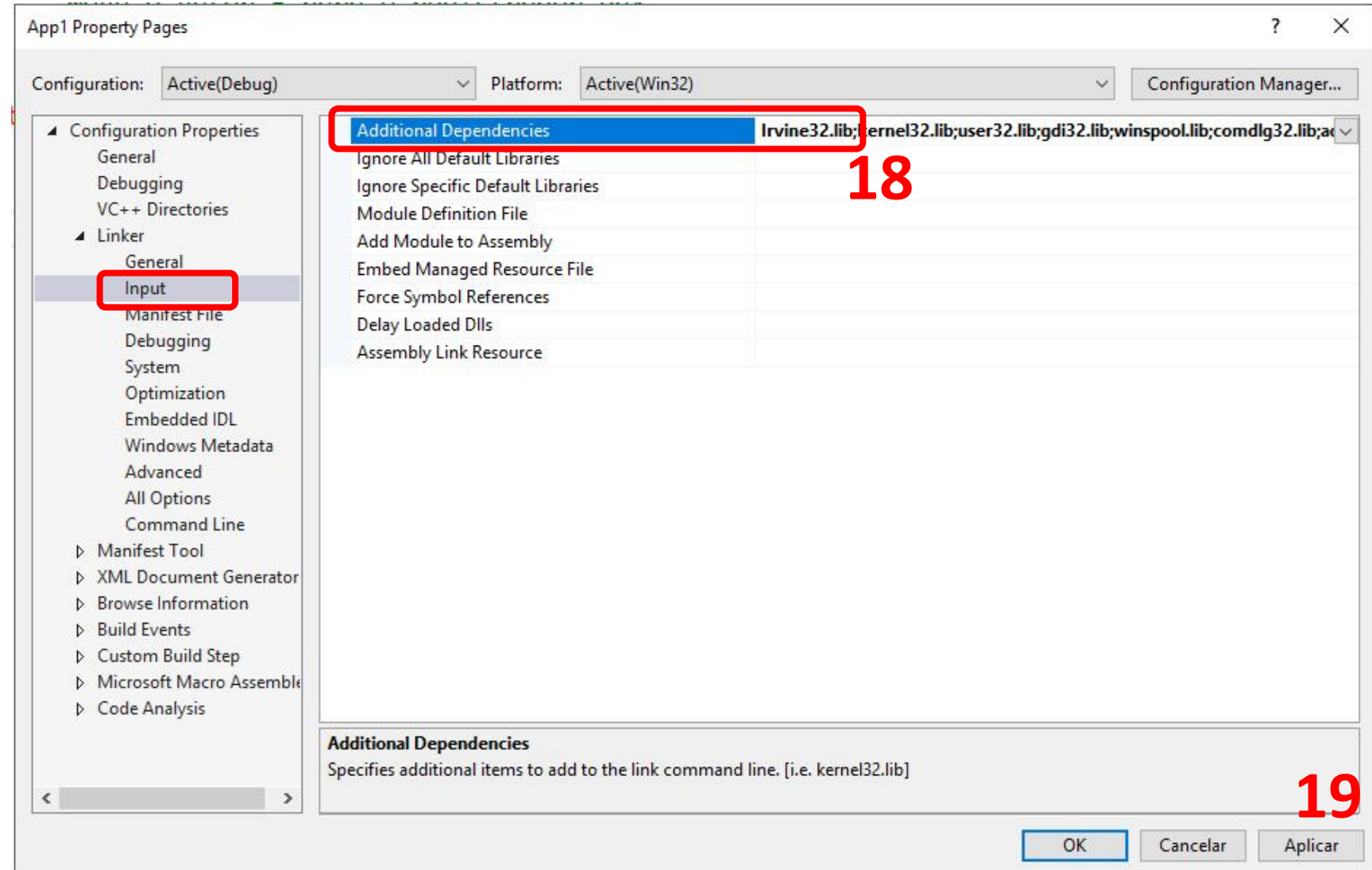


Configurações do Projeto (inserindo libs e paths)

18. Em **Linker (Input)**,
adicione a biblioteca.

Irvine32.lib

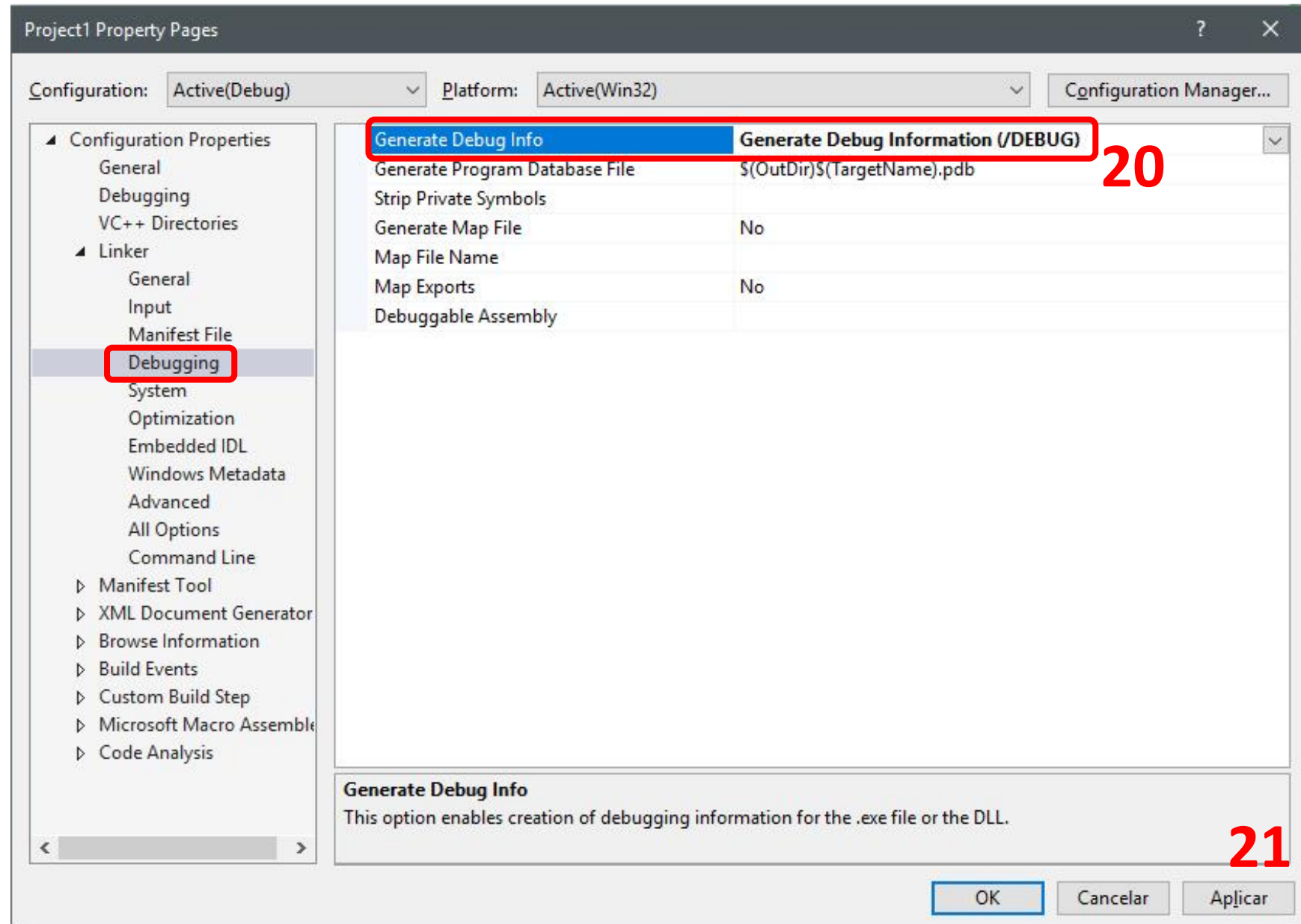
19. Clique em **Aplicar**



Configurações do Projeto (inserindo libs e paths)

20. Em **Linker** (**Debugging**), ative o modo para depuração da aplicação.

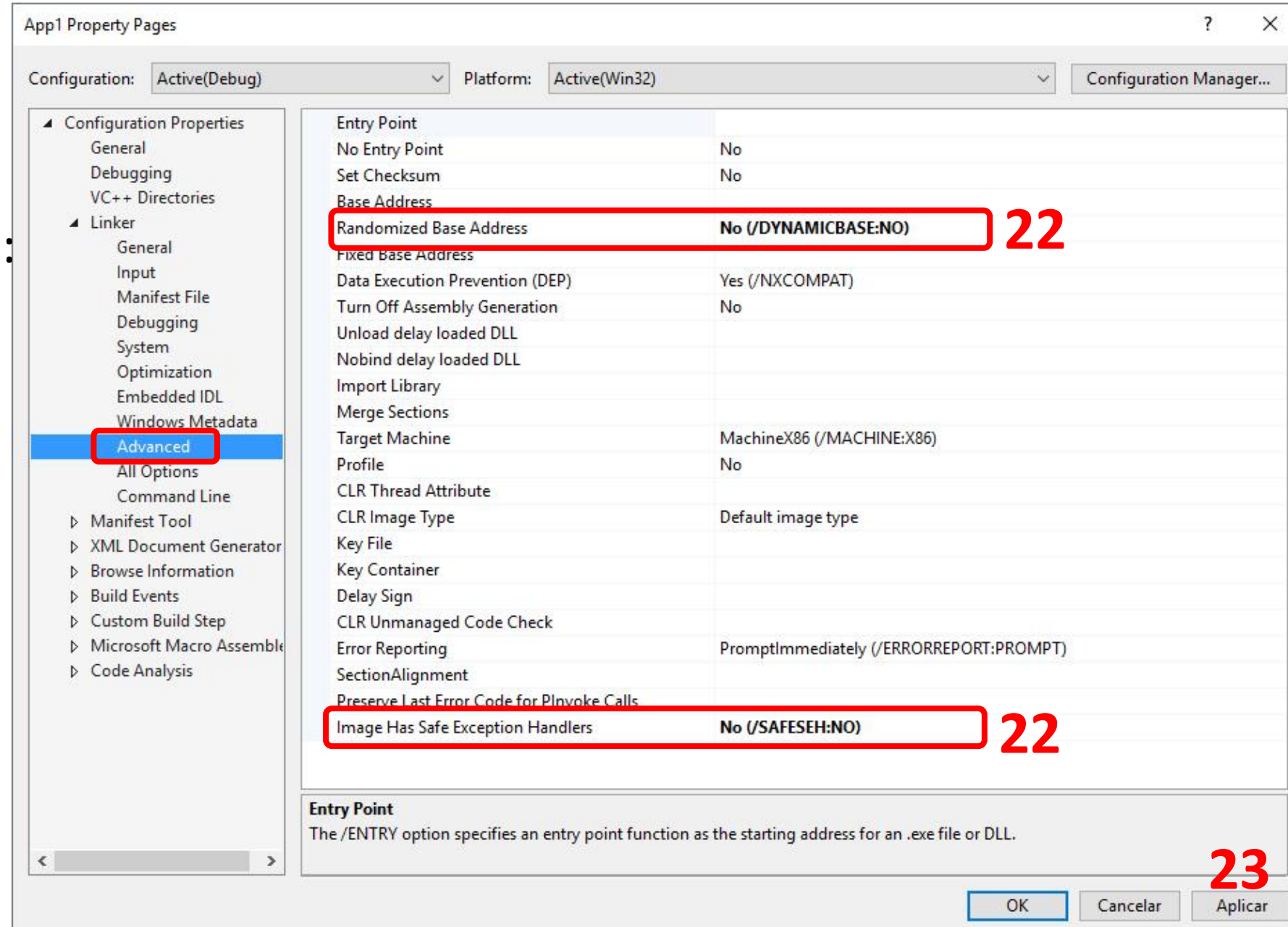
21. Clique em **Aplicar**



Configurações do Projeto (inserindo libs e paths)

22. Em **Linker** (**Advanced**), modifique as opções conforme mostra a figura ao lado:

23. Clique em **Aplicar**

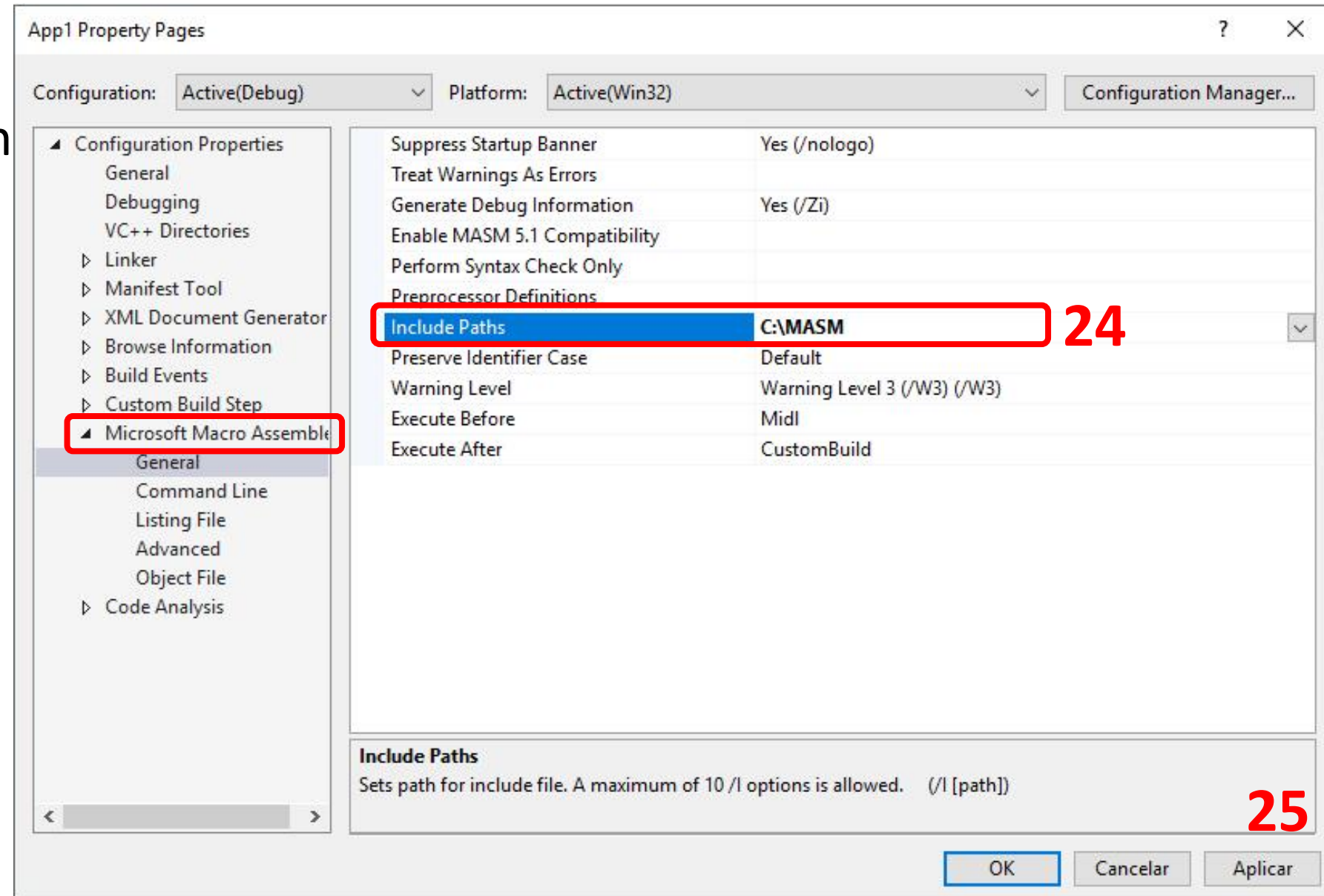


Configurações do Projeto (inserindo libs e paths)

24. Em **Microsoft Macro Assembly (General)**, insira o path para as bibliotecas estáticas.

C:\MASM

25. Clique em **Aplicar**



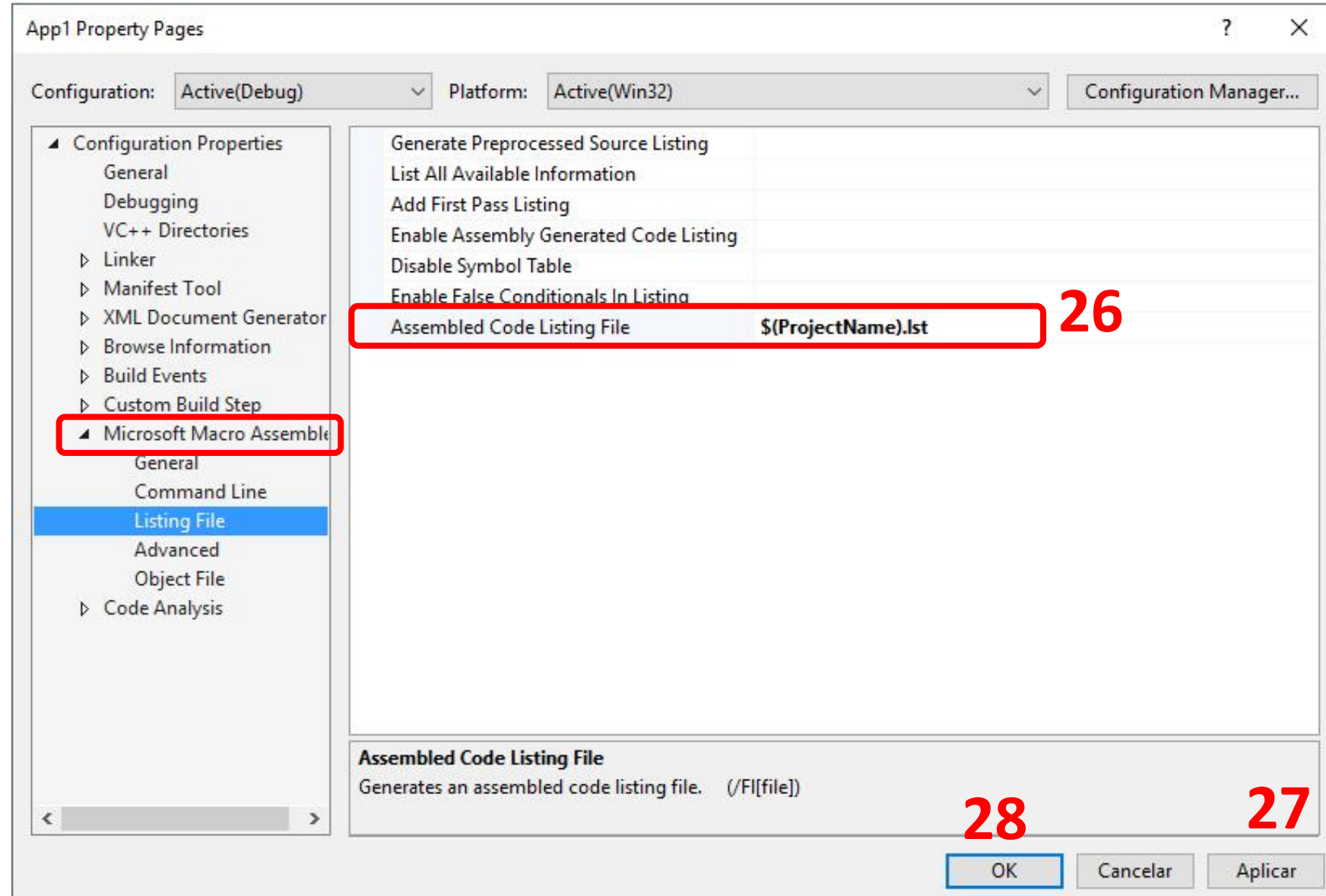
Configurações do Projeto (inserindo libs e paths)

26. Em **Microsoft Macro Assembly (Listing File)**, insira o nome do arquivo para listagem do código de máquina gerado após a compilação:

\$(ProjectName).lst

27. Clique em **Aplicar**

28. Clique **OK** para finalizar o procedimento de configuração do projeto



Elementos Básicos da Linguagem Assembly

Elementos Básicos da Linguagem Assembly

- O programa ao lado adiciona dois números inteiros e salva o resultado em um registrador:

```
1  COMMENT !
2  MASM01
3  Somando dois inteiros de 32 - bit
4  !
5
6  .386
7  .model flat, stdcall
8  .stack 4096
9  ExitProcess proto, dwExitCode:dword
10
11 .code
12 main proc
13     mov eax, 5      ; move o valor 5 para o registrador eax
14     add eax, 6      ; adiciona 6 para o registrador eax
15
16     invoke ExitProcess, 0      ; finaliza o programa
17 main endp
18 end main
```


Elementos Básicos da Linguagem Assembly

- A linha 12 inicia a **função principal** (entry point) do programa.
- Na linha 13, o número inteiro 5 é colocado no registrador geral **eax**.
- Na linha 14 , é adicionado 6 ao valor já presente no registrador eax, totalizando 11.
- Na linha 16 é chamado um serviço/função do Windows denominada **ExitProcess**. Essa função encerra o programa e devolve o controle ao Sistema Operacional.
- A linha 17 marca o fim da função principal do programa.
- As linha 1 a 4 são a forma de fazer comentários em grupo. Comentários individuais usam o caracter ;

Adicionando uma variável

- O programa ao lado modifica o exemplo anterior acrescentando uma variável que salva o resultado da soma na memória RAM.

```
1  COMMENT !
2      MASM02
3      Somando dois inteiros de 32 - bit
4      e armazena resultado em uma variável
5  !
6
7  .386
8  .model flat, stdcall
9  .stack 4096
10 ExitProcess proto, dwExitCode:dword
11
12 .data                                ; esse é o segmento dos dados (variáveis)
13 sum DWORD 0                        ; declara e inicializa a variável sum
14
15 .code                                ; esse é o segmento do código-fonte
16 main proc
17     mov eax, 5                      ; move o valor 5 para o registrador eax
18     add eax, 6                      ; adiciona 6 para o registrador eax
19     mov sum, eax                    ; move o conteúdo do registrador eax para a variável sum
20
21     invoke ExitProcess, 0           ; finaliza o programa
22 main endp
23 end main
```

Adicionando uma variável

- A variável **sum** é declarada e inicializada na linha 13. Seu tamanho é 32 bits, usando o tipo de dado **DWORD**.
 - Existem outros tipos de dados para declarar variáveis em Assembly. Porém, eles são diferentes dos tipos de dados que estamos familiarizados, tais como: **int**, **float**, **double**...
 - **Em Assembly, os tipos de dados apenas especificam o TAMANHO.** Não há uma checagem do conteúdo que é inserido no registrador. VOCÊ ESTÁ NO CONTROLE TOTAL!
😊
- O programa também separa o código-fonte das declarações de variáveis através de **Segmentos**:
 - O segmento **.CODE** fica a função principal e o código do programa.
 - O segmento **.DATA** são declaradas e inicializadas as variáveis alocadas na memória RAM.

Inteiros (constantes)

- Para declarar um valor constante, é necessário especificar:
 1. (Opcional) Um sinal que indica se o número é positivo ou negativo;
 2. Um ou mais dígitos inteiros;
 3. (Opcional) Um caracter RADIX que indica a base do número.

[{ + | - }] digits [radix]

h	hexadecimal	r	encoded real
q/o	octal	t	decimal (alternate)
d	decimal	y	binary (alternate)
b	binary		

26 ; decimal
26d ; decimal
11010011b ; binary
42q ; octal
42o ; octal
1Ah ; hexadecimal
0A3h ; hexadecimal

Um literal hexadecimal que começa com uma letra deve iniciar com um zero

Operadores e Expressões Matemáticas

- Os seguintes operadores aritméticos são válidos no Assembly:

Operator	Name	Precedence Level
()	Parentheses	1
+, −	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, −	Add, subtract	4

- A procedência dos operadores é a ordem das operações quando uma expressão contém dois ou mais operandos.

- Usar parênteses ajuda a manter a ordem das operações...

4 + 5 * 2	Multiply, add
12 - 1 MOD 5	Modulus, subtract
-5 + 2	Unary minus, add
(4 + 2) * 6	Add, multiply

Número Real (floating-point)

- Para declarar um valor Real, é necessário especificar:
 1. (Opcional) Um sinal que indica se o número é positivo ou negativo;
 2. Um ou mais dígitos inteiros;
 3. Um ponto decimal;
 4. (Opcional) Um inteiro que expressa uma fração;
 5. (Opcional) Um expoente.

[sign] integer . [integer] [exponent]

- Formatos para o sinal e o expoente:

sign {+, -}

exponent E[+, -] *integer*

- Exemplos de números Reais válidos:

2.

+3.0

-44.2E+05

26.E5

Caracteres e Strings

- **Caracteres** podem ser definidos com o uso de aspas simples ou duplas. Ex.: 'A' – "A".

```
'ABC'  
'X'  
"Good night, Gracie"  
'4096'
```

- **Strings** são sequências de caracteres (incluindo espaço em branco):

```
"This isn't a test"  
'Say "Good night," Gracie'
```

- *Caracteres e Strings são armazenados como inteiros, usando uma sequência codificada ASCII. Então, quando se escreve o caracter "A", ele é armazenado na memória com o número 65 (ou 41 hexadecimal).*

Tabela ASCII

ASCII control characters			ASCII printable characters									Extended ASCII characters											
DEC	HEX	Simbolo ASCII	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
00	00h	NULL (carácter nulo)	32	20h	espacio	64	40h	@	96	60h	`	128	80h	Ç	160	A0h	á	192	C0h	Ł	224	E0h	Ó
01	01h	SOH (inicio encabezado)	33	21h	!	65	41h	A	97	61h	a	129	81h	ü	161	A1h	í	193	C1h	ł	225	E1h	ô
02	02h	STX (inicio texto)	34	22h	"	66	42h	B	98	62h	b	130	82h	é	162	A2h	ó	194	C2h	Ł	226	E2h	Ô
03	03h	ETX (fin de texto)	35	23h	#	67	43h	C	99	63h	c	131	83h	â	163	A3h	ú	195	C3h	ł	227	E3h	Ò
04	04h	EOT (fin transmisión)	36	24h	\$	68	44h	D	100	64h	d	132	84h	ä	164	A4h	ñ	196	C4h	Ł	228	E4h	ö
05	05h	ENQ (enquiry)	37	25h	%	69	45h	E	101	65h	e	133	85h	à	165	A5h	Ñ	197	C5h	ł	229	E5h	Õ
06	06h	ACK (acknowledgement)	38	26h	&	70	46h	F	102	66h	f	134	86h	á	166	A6h	ª	198	C6h	Ł	230	E6h	µ
07	07h	BEL (timbre)	39	27h	'	71	47h	G	103	67h	g	135	87h	ç	167	A7h	º	199	C7h	ł	231	E7h	þ
08	08h	BS (retroceso)	40	28h	(72	48h	H	104	68h	h	136	88h	ê	168	A8h	¿	200	C8h	Ł	232	E8h	þ
09	09h	HT (tab horizontal)	41	29h)	73	49h	I	105	69h	i	137	89h	ë	169	A9h	®	201	C9h	ł	233	E9h	Û
10	0Ah	LF (salto de línea)	42	2Ah	*	74	4Ah	J	106	6Ah	j	138	8Ah	è	170	AAh	¬	202	CAh	Ł	234	EAh	Ü
11	0Bh	VT (tab vertical)	43	2Bh	+	75	4Bh	K	107	6Bh	k	139	8Bh	ï	171	ABh	½	203	CBh	ł	235	EBh	Ý
12	0Ch	FF (form feed)	44	2Ch	,	76	4Ch	L	108	6Ch	l	140	8Ch	î	172	ACH	¼	204	CCh	Ł	236	ECh	ÿ
13	0Dh	CR (retorno de carro)	45	2Dh	-	77	4Dh	M	109	6Dh	m	141	8Dh	ì	173	ADh	»	205	CDh	ł	237	EDh	Ÿ
14	0Eh	SO (shift Out)	46	2Eh	.	78	4Eh	N	110	6Eh	n	142	8Eh	Ā	174	AEnh	«	206	CEh	Ł	238	EEnh	˘
15	0Fh	SI (shift In)	47	2Fh	/	79	4Fh	O	111	6Fh	o	143	8Fh	Ă	175	AFh	»	207	CFh	ł	239	EFh	˙
16	10h	DLE (data link escape)	48	30h	0	80	50h	P	112	70h	p	144	90h	Ė	176	B0h	⋮	208	D0h	Ł	240	F0h	˚
17	11h	DC1 (device control 1)	49	31h	1	81	51h	Q	113	71h	q	145	91h	æ	177	B1h	⋮	209	D1h	ł	241	F1h	±
18	12h	DC2 (device control 2)	50	32h	2	82	52h	R	114	72h	r	146	92h	Æ	178	B2h	⋮	210	D2h	Ł	242	F2h	̄
19	13h	DC3 (device control 3)	51	33h	3	83	53h	S	115	73h	s	147	93h	ô	179	B3h	⋮	211	D3h	ł	243	F3h	¼
20	14h	DC4 (device control 4)	52	34h	4	84	54h	T	116	74h	t	148	94h	ò	180	B4h	⋮	212	D4h	Ł	244	F4h	¶
21	15h	NAK (negative acknowle.)	53	35h	5	85	55h	U	117	75h	u	149	95h	ò	181	B5h	⋮	213	D5h	ł	245	F5h	§
22	16h	SYN (synchronous idle)	54	36h	6	86	56h	V	118	76h	v	150	96h	û	182	B6h	⋮	214	D6h	Ł	246	F6h	÷
23	17h	ETB (end of trans. block)	55	37h	7	87	57h	W	119	77h	w	151	97h	ù	183	B7h	⋮	215	D7h	ł	247	F7h	ˆ
24	18h	CAN (cancel)	56	38h	8	88	58h	X	120	78h	x	152	98h	ÿ	184	B8h	©	216	D8h	Ł	248	F8h	˚
25	19h	EM (end of medium)	57	39h	9	89	59h	Y	121	79h	y	153	99h	Ō	185	B9h	⋮	217	D9h	ł	249	F9h	˙
26	1Ah	SUB (substitute)	58	3Ah	:	90	5Ah	Z	122	7Ah	z	154	9Ah	Ü	186	BAh	⋮	218	DAh	Ł	250	FAh	˘
27	1Bh	ESC (escape)	59	3Bh	;	91	5Bh	[123	7Bh	{	155	9Bh	ø	187	BBh	⋮	219	DBh	ł	251	FBh	˙
28	1Ch	FS (file separator)	60	3Ch	<	92	5Ch	\	124	7Ch		156	9Ch	£	188	BCh	⋮	220	DCh	Ł	252	FCh	˚
29	1Dh	GS (group separator)	61	3Dh	=	93	5Dh]	125	7Dh	}	157	9Dh	Ø	189	BDh	¢	221	DDh	ł	253	FDh	²
30	1Eh	RS (record separator)	62	3Eh	>	94	5Eh	^	126	7Eh	~	158	9Eh	×	190	BEh	¥	222	DEh	Ł	254	FEh	■
31	1Fh	US (unit separator)	63	3Fh	?	95	5Fh	-				159	9Fh	f	191	BFh	¬	223	DFh	ł	255	FFh	

Palavras Reservadas

- Palavras reservadas possuem seu próprio significado/função e só podem ser usadas no contexto correto.
- ***Palavras reservadas não são case sensitive.*** Ex.: **MOV** é o mesmo que **mov**
- Existem diferentes tipos de palavras reservadas:
 - Instrução mnemônica, tais como: MOV, ADD, MUL, SUB;
 - Nomes de Registradores. EAX, AIP, AX, RAX...
 - Diretivas da linguagem Assembly, que dizem como os programas devem ser montados.
 - Atributos, que fornecem tamanhos e informações de uso para variáveis e operandos. Ex. BYTE, DWORD...
 - Operadores aritméticos.
 - Símbolos pré-definidos, tais como @data, que retorna um valores inteiros constantes.

Identificadores

- Um *identificador* é um nome definido pelo programador. É usado para identificar uma variável, uma constante, uma função, etc.
- Algumas regras para se criar um identificador:
 - Deve conter de 1 a 247 caracteres.
 - Não devem ser case sensitive.
 - O primeiro caracter deve ser uma letra (A...Z, a...z), underline (_), @, ? Ou \$. Caracteres subsequentes também podem conter dígitos.
 - Um identificador não pode ser uma palavra reservada da linguagem Assembly.

Diretivas

- Uma diretiva é um comando incorporado no código-fonte que é reconhecido pelo assembler.
- Diretivas não executam em runtime, mas elas são usadas para definir variáveis, macros e procedimentos.
- As diretivas podem especificar nomes para segmentos de memória e realizar outras tarefas relacionadas com o montador específico.
- Diretivas não são, por padrão, case sensitive. Ex.: **.data** **.DATA** **.Data** são equivalentes.

Diretivas

- O exemplo abaixo mostra a diferença entre uma diretiva e uma instrução:

myVar DWORD 26

mov eax, myVar

- A diretiva **DWORD** diz ao montador para reservar espaço na memória para armazenar uma variável, chamada **myVar**, e atribui o valor inteiro 26.
- A instrução **MOV**, executa em runtime, copia o conteúdo de **myVar** para o registrador **EAX**.
- *Apesar de todos os montadores (assemblers) para processadores Intel compartilharem o mesmo conjunto de instruções, eles podem possuir um conjunto de diretivas diferentes.*

Diretivas (Definindo Segmentos)

- Uma função importante das diretivas é definir as seções dos programas assembly, denominadas de **Segmentos**.

- *Alguns segmentos importantes são:*

.DATA - Diretiva usado para declarar e inicializar variáveis.

.CODE – Diretiva usada para identificar a área do programa contendo as instruções executáveis.

.STACK 4096 – Diretiva que diz ao assembler quantos bytes de memória devem ser alocados para a pilha de execução do programa.

Instruções

- Uma **instrução** é uma declaração que se torna executável quando um programa é montado.
- Instruções são traduzidas pelo montador assembler em bytes de linguagem de máquina, que são carregadas e executadas pela CPU em tempo de execução.
- Uma instrução contém quatro partes básicas:
 - Label (opcional)
 - Mnemônico da Instrução (requerido)
 - Operando(s) (usualmente requerido)
 - Comentários (opcional)

```
[label:] mnemonic [operands] [;comment]
```

Instruções (Label)

- Uma **Label** é um identificador que atua como uma localização para instruções e dados.
- Uma Label inserida antes de uma instrução se torna o endereço da instrução.
- Uma Label inserida antes de uma variável se torna o endereço da variável.
- Existem dois tipos de Labels:
 - **Data Labels**
 - **Code Labels**

Instruções (Label)

- Uma **Data Label** identifica a localização de uma variável, proporcionando uma maneira conveniente de referenciar a variável no código.
- O montador designa um endereço numérico para cada label. O código abaixo define uma variável chamada *count*:

count DWORD 100

- O código abaixo mostra a label *array* definindo a localização do primeiro número 1024. Os outros números são armazenados em sequência na memória.

array DWORD 1024, 2048

DWORD 4096, 8192

Instruções (Label)

- Uma label em uma área de código do programa (onde as instruções estão localizadas) devem terminar com o caracter dois pontos (:)
- **Code Labels** são usadas como pontos iniciais para desvios e looping de instruções.
- O código abaixo mostra a instrução **JMP** (jump) transferindo a execução do código para a label **target**:

```
target:  
    mov    ax, bx  
    ...  
    jmp    target
```

```
L1:  mov    ax, bx  
L2:
```

Instruções (Mnemônicos)

- Um **Mnemônico** de instrução é uma palavra que identifica uma instrução. Ou seja, são os tipos de operações que podem ser realizadas.
- Alguns mnemônicos da linguagem Assembly são:

Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

Instruções (Operandos)

- Um **operando** é um valor que é usado para entrada ou saída para uma instrução.
- As instruções da linguagem Assembly podem ter de zero a três operandos, onde podem ser tanto um registrador, operando de memória, expressão de inteiros ou uma porta de entrada/saída.

stc	; Seta a Carry Flag
inc eax	; incrementa 1 em EAX
mov count, ebx	; move EBX para count

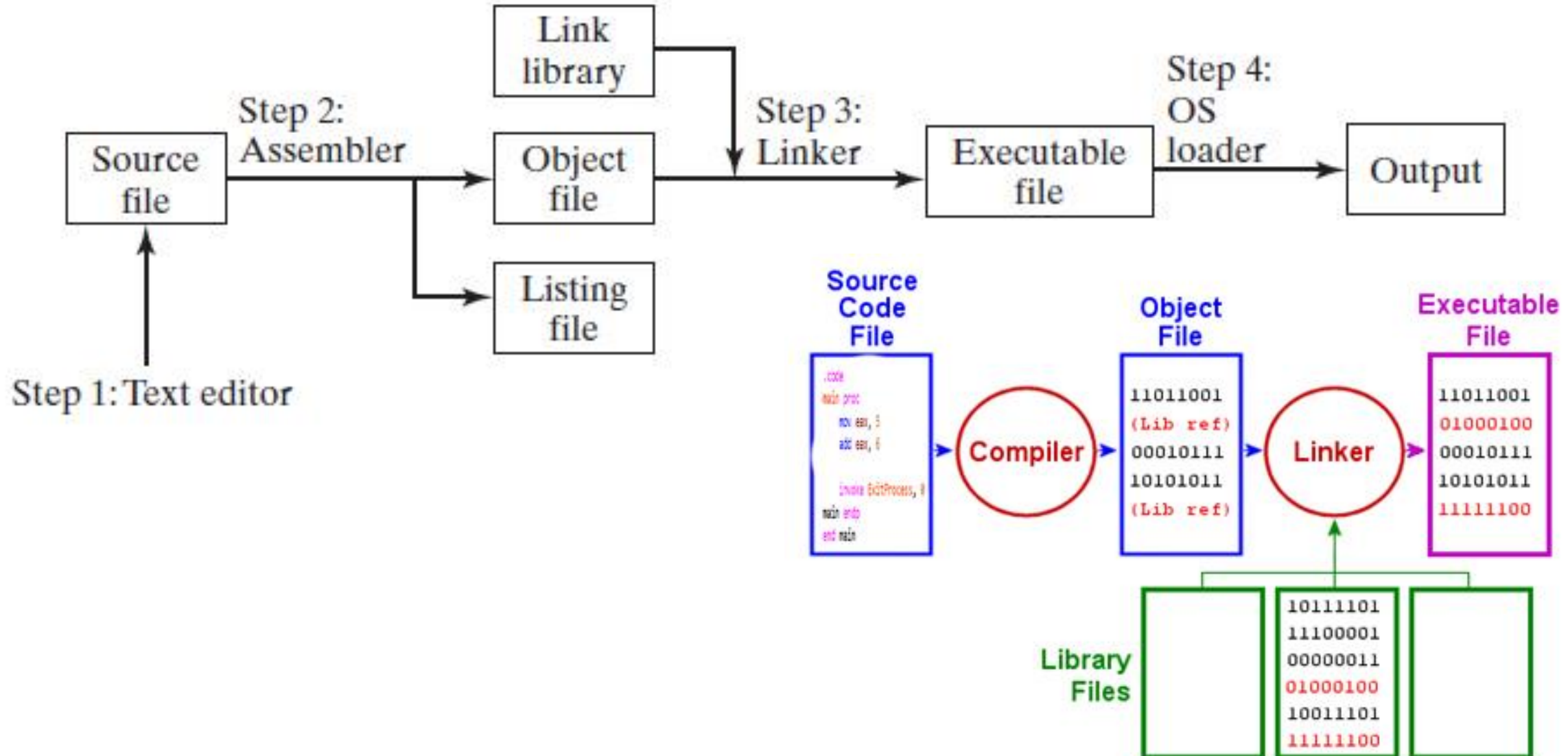
Example	Operand Type
96	<i>Integer literal</i>
2 + 4	<i>Integer expression</i>
eax	<i>Register</i>
count	<i>Memory</i>

Assembling, Linking and Running Programs

Ciclo de Assemble-Link-Execute

1. O programador usa um **editor de texto** para criar um arquivo de código-fonte escrito em Assembly.
2. O **Assembler** (MASM) lê o arquivo e produz um *arquivo Objeto*, que é a tradução do programa em linguagem de máquina.
3. O **Linker** lê o *arquivo Objeto* e checa se o programa contém alguma chamada de procedimentos em alguma biblioteca externa.
 - Se existe, todos os procedimentos necessários são copiados da biblioteca e combinados com o arquivo Objeto.
 - Logo após, é produzido o *arquivo Executável*.
4. O **Loader** do Sistema Operacional lê o arquivo Executável na memória e aponta à CPU para o endereço inicial do programa. O programa inicia sua execução...

Ciclo de Assemble-Link-Execute



Tipos de Dados

Tipos de Dados

1. O Assembler reconhece um conjunto básico de Tipos de Dados Intrinsicos:

- a) Que definem tipos em termos de seus tamanhos (byte, word, doubleword...);
- b) Se os números possuem sinal (positivo, negativo)
- c) Se os números são inteiros ou reais.

`[name] directive initializer [, initializer] . . .`

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

`count`

`DWORD`

`12345`

Tipos de Dados

Name: um nome para a variável

Directive: tipo de dado que pode ser qualquer nome listado na tabela do slide anterior. Ou também são válidos os tipos de dados legados, mostrados na tabela ao lado.

Inicializer: pelo menos um inicializador é necessário para a definição dos dados, mesmo se for zero.

Todos os inicializadores, independente do seu formato, são convertidos para dados binários pelo montador assembler.

Legacy Data Directives.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

Definindo BYTE e SBYTE

```
value1 BYTE    'A' ; character literal
value2 BYTE     0 ; smallest unsigned byte
value3 BYTE    255 ; largest unsigned byte
value4 SBYTE -128 ; smallest signed byte
value5 SBYTE +127 ; largest signed byte
value6 BYTE ?    ; not initialized
```

```
list BYTE 10,20,30,40
```

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

Nem todas as definições de dados necessitam de labels. Por exemplo, para continuar o array de bytes iniciado com **list**, é necessário apenas definir bytes adicionais nas linhas subsequentes:

```
list BYTE 10,20,30,40
        BYTE 50,60,70,80
        BYTE 81,82,83,84
```

Dentro de uma mesma definição, os inicializadores podem usar radices diferentes. Caracteres e strings podem ser misturados livremente.

```
list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h
```

Definindo STRINGS

O tipo mais comum de STRING termina com um byte nulo (contendo zero)

```
greeting1 BYTE "Good afternoon",0  
greeting2 BYTE 'Good night',0
```







Cada caractere usa 1 byte de armazenamento

```
greeting1 BYTE 'G','o','o','d'....etc.
```

Os códigos hexadecimais **0Dh** e **0Ah** são chamados de **CR/LF** (carriage-return line-feed) ou **caracteres de fim da linha**.

```
greeting1 BYTE "Uma STRING pode ser dividida em múltiplas linhas "  
            BYTE "sem a necessidade de inserir uma label para cada linha.",0dh,0ah  
            BYTE "If you wish to modify this program, please "  
            BYTE "send me a copy.",0dh,0ah,0
```

```

1  COMMENT ! MASM03 !
2
3   ; Identifica este como um programa 32-bit que
4   ; pode acessar registradores e endereços de 32-bit
5  .386
6
7  .model flat, stdcall
8
9  ; Reserva 4096 bytes para a pilha do programa em tempo de execução
10 .stack 4096
11
12  ; Declara um protótipo para a função ExitProcess (que é um serviço do windows)
13  ; Um protótipo consiste em um NOME para a função seguido da palavra-chave PROTO,
14  ; de uma VÍRGULA e, em seguida, uma LISTA DE PARÂMETROS.
15 ExitProcess proto, dwExitCode:dword
16
17 .data                ; esse é o segmento dos dados (variáveis)
18 sum DWORD 15        ; declara e inicializa a variável sum
19
20 .code                ; esse é o segmento do código-fonte
21  main proc          ; início da função/procedimento principal
22     mov eax, sum      ; passa o valor da variável sum para o registrador eax
23     add eax, (5 + 5) * 2 ; faz uma expressão matemática
24     add eax, 10       ; adiciona 6 para o registrador eax
25     sub eax, 2        ; subtrai 2 do valor no registrador eax
26     inc eax          ; incrementa 1 ao valor no registrador eax
27     mov sum, eax      ; move o conteúdo do registrador eax para a variável sum
28
29     invoke ExitProcess, 0 ; finaliza o programa
30 main endp
31 end main            ; fim da função principal

```

O Operador DUP

- O operador DUP aloca espaço de armazenamento para múltiplos itens de dados, usando uma expressão inteira como um contador).
- É útil quando há a necessidade de alocar espaço para uma string ou array.
- Pode ser usado com inicialização ou apenas declaração.

```
BYTE 20 DUP(0) ; 20 bytes, all equal to zero
```

```
BYTE 20 DUP(?) ; 20 bytes, uninitialized
```

```
BYTE 4 DUP("STACK") ; 20 bytes: "STACKSTACKSTACKSTACK"
```

Definindo WORD e SWORD

São diretivas que definem inteiros de 16-bit. WORD (sem sinal) - SWORD (com sinal)

```
word1 WORD 65535 ; largest unsigned value
word2 SWORD -32768 ; smallest signed value
word3 WORD ? ; uninitialized, unsigned
```

Um array de inteiros de 16-bit positivos pode ser criado listando os elementos ou usando o operador DUP.

```
myList WORD 1,2,3,4,5      array WORD 5 DUP(?) ; 5 values, uninitialized
```

Um array de inteiros de 16-bit positivos pode ser criado listando os elementos ou usando o operador DUP.

Offset	Value
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

Memory layout, 16-bit word array.

Definindo DWORD e SDWORD

São diretivas que definem inteiros de 32-bit. DWORD (sem sinal) - SDWORD (com sinal)

```
val1 DWORD 12345678h ; unsigned
val2 SDWORD -2147483648 ; signed
val3 DWORD 20 DUP(?) ; unsigned array
```

DWORD pode ser usado para declarar uma variável que contém um offset de 32-bit de outra variável. Abaixo, **pVal** contém o offset de **val3**.

pVal DWORD val3

Um array de inteiros de 32-bit positivos pode ser criado listando os elementos ou usando o operador DUP.

```
myList DWORD 1,2,3,4,5
```

Diagrama do array na memória, assumindo que **myList** começa no offset 0000

Memory layout, 32-bit doubleword array.

Offset	Value
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

Definindo QDWORD

São diretivas que definem inteiros de 64-bit (8-byte).

```
quad1 QWORD 1234567812345678h
```

Legacy Data Directives.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

Diretivas antigas também são válidas para declarar variáveis:

```
val1 DB 255 ; unsigned byte
val2 DB -128 ; signed byte

val1 DW 65535 ; unsigned word
val2 DW -32768 ; signed word

val1 DD 12345678h ; unsigned dword
val2 DD -2147483648 ; signed dword

quad1 DQ 1234567812345678h ; signed qword
```

Definindo Floating-Point

REAL4 define uma variável **float** de 4-byte. **REAL8** define uma variável **double** de 8-byte. **REAL10** define uma variável **double** de 10-byte.

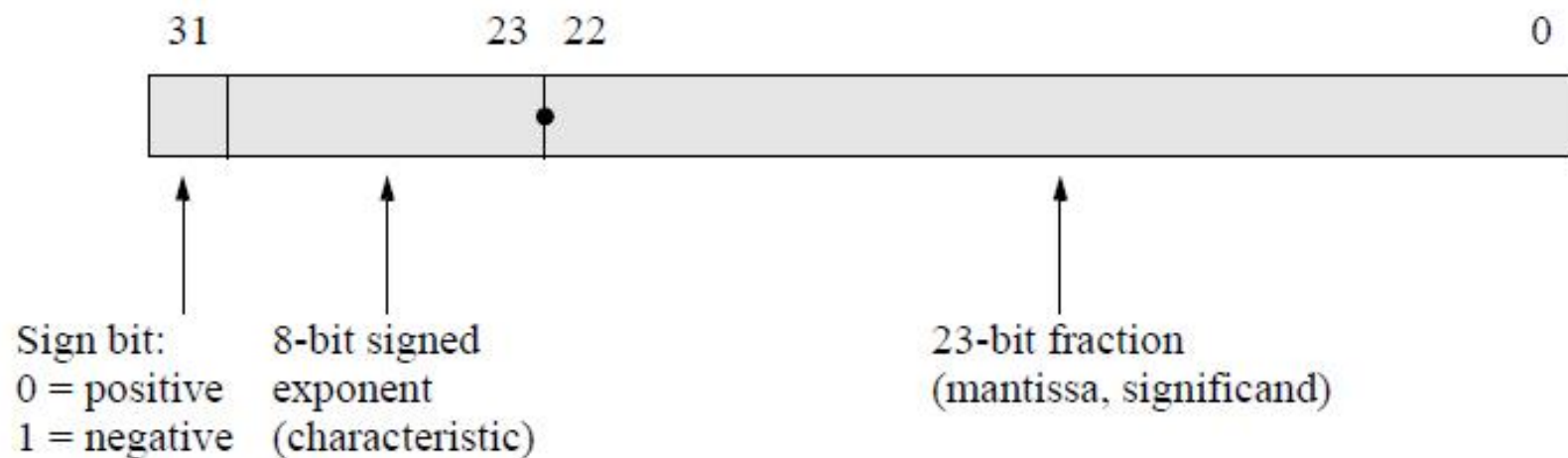
```
rVal1 REAL4 -1.2           ; short real
rVal2 REAL8 3.2E-260        ; long real
rVal3 REAL10 4.6E+4096      ; extended-precision real
ShortArray REAL4 20 DUP(0.0)
```

Standard Real Number Types.

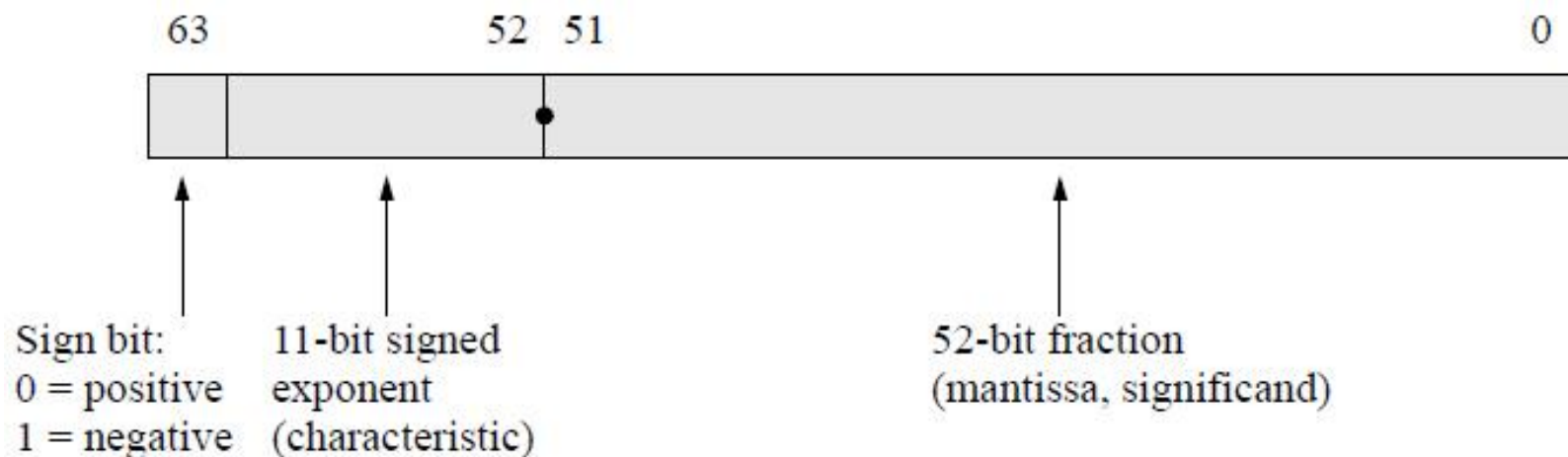
Data Type	Significant Digits	Approximate Range
Short real	6	1.18×10^{-38} to 3.40×10^{38}
Long real	15	2.23×10^{-308} to 1.79×10^{308}
Extended-precision real	19	3.37×10^{-4932} to 1.18×10^{4932}

Definindo Floating-Point

Formato de um ponto flutuante segundo o IEEE (32-bit)



Formato de um ponto flutuante segundo o IEEE (64-bit)



```

1  COMMENT ! MASM03 !
2
3  □; Identifica este como um programa 32-bit que
4  |; pode acessar registradores e endereços de 32-bit
5  |.386
6
7  .model flat, stdcall
8
9  ; Reserva 4096 bytes para a pilha do programa em tempo de execução
10 .stack 4096
11
12 □; Declara um protótipo para a função ExitProcess (que é um serviço do windows)
13 |; Um protótipo consiste em um NOME para a função seguido da palavra-chave PROTO,
14 |; de uma VÍRGULA e, em seguida, uma LISTA DE PARÂMETROS.
15 ExitProcess proto, dwExitCode:dword
16
17 .data ; esse é o segmento dos dados (variáveis)
18 val1 DWORD 00000101h ; declara e inicializa a variável val1
19 val2 DWORD 3 ; declara e inicializa a variável val2
20 val3 DWORD -60 ; declara e inicializa a variável val3
21 sum DWORD 0 ; declara e inicializa a variável sum
22
23 .code ; esse é o segmento do código-fonte
24 □main proc ; início da função/procedimento principal
25
26     mov eax, val1 ; move val1 para o registrador eax
27     add eax, val2 ; soma val2 ao valor já existente em eax (EAX = EAX + val2)
28     add eax, val3 ; soma val3 ao valor já existente em eax (EAX = EAX + val3)
29     mov sum, eax ; armazena o conteúdo do registrador EAX na variável sum na memória RAM
30
31     invoke ExitProcess, 0 ; finaliza o programa
32 main endp
33 end main ; fim da função principal

```

Exercícios

1 – Crie um programa em Assembly que calcule a seguinte expressão:

$$\text{RESULT} = ((A + B) - (C + D))^2 \quad \Rightarrow \quad \mathbf{3636649}$$

2 – Crie um programa em Assembly que calcule a seguinte expressão:

$$A = ((A * B) - (C + D)) + A \quad \Rightarrow \quad \mathbf{361599}$$

Converta os números abaixo e utilize números hexadecimais para as variáveis A, B, C, D:

$$A = 2543$$

$$B = 143$$

$$C = 1050$$

$$D = 3543$$

Exercícios

3 – Crie um programa em Assembly que calcule a seguinte expressão:

$$\text{RESULT} = ((A + B)^2 - (C + D)^3) + (A + B + C + D). \Rightarrow \mathbf{264}$$

$$A = 10$$

$$B = 5$$

$$C = -2$$

$$D = -1$$

4 - Crie um programa em Assembly que calcule a seguinte expressão:

$$\text{RESULT} = (((A * 2) - B) + (D + B)^2 - (C^3)) \Rightarrow \mathbf{17}$$

$$A = 10$$

$$B = 20$$

$$C = 2$$

$$D = -15$$

Exercícios

5 – Verdadeiro ou Falso:

- () Um arquivo Objeto é produzido pelo Linker
- () Uma biblioteca é adicionada pelo Linker antes da produção do arquivo Executável.
- () Um identificador não pode começar com um dígito.
- () Um hexadecimal literal pode ser escrito como 0x3A.
- () As diretivas da linguagem Assembly executam em tempo de execução (runtime).
- () As diretivas da linguagem Assembly podem ser escritas com qualquer combinação de letras maiúsculas e minúsculas.
- () MOV é um exemplo de instrução mnemônica.