

# **INTRODUCTION: ASSEMBLY LANGUAGE X86**

## **TOPICS**

- 1. Basic Concepts:** Applications of assembly language, basic concepts, machine language, and data representation.
- 2. x86 Processor Architecture:** Basic microcomputer design, instruction execution cycle, x86 processor architecture, Intel64 architecture, x86 memory management, components of a microcomputer, and the input-output system.
- 3. Assembly Language Fundamentals:** Introduction to assembly language, linking and debugging, and defining constants and variables.
- 4. Data Transfers, Addressing, and Arithmetic:** Simple data transfer and arithmetic instructions, assemble-link-execute cycle, operators, directives, expressions, JMP and LOOP instructions, and indirect addressing.
- 5. Procedures:** Linking to an external library, description of the book's link library, stack operations, defining and using procedures, flowcharts, and top-down structured design.
- 6. Conditional Processing:** Boolean and comparison instructions, conditional jumps and loops, high-level logic structures, and finite-state machines.
- 7. Integer Arithmetic:** Shift and rotate instructions with useful applications, multiplication and division, extended addition and subtraction, and ASCII and packed decimal arithmetic.
- 8. Advanced Procedures:** Stack parameters, local variables, advanced PROC and INVOKE directives, and

recursion.

**9. Strings and Arrays:** String primitives, manipulating arrays of characters and integers, two-dimensional arrays, sorting, and searching.

**10. Structures and Macros:** Structures, macros, conditional assembly directives, and defining repeat blocks.

**11. MS-Windows Programming:** Protected mode memory management concepts, using the Microsoft-Windows API to display text and colors, and dynamic memory allocation.

**12. Floating-Point Processing and Instruction Encoding:** Floating-point binary representation and floating-point arithmetic. Learning to program the IA-32 floating-point unit. Understanding the encoding of IA-32 machine instructions.

**13. High-Level Language Interface:** Parameter passing conventions, inline assembly code, and linking assembly language modules to C and C++ programs.

**14. 16-Bit MS-DOS Programming:** Memory organization, interrupts, function calls, and standard MS-DOS file I/O services.

**15. Disk Fundamentals:** Disk storage systems, sectors, clusters, directories, file allocation tables, handling MS-DOS error codes, and drive and directory manipulation.

**16. BIOS-Level Programming:** Keyboard input, video text, graphics, and mouse programming.

**17. Expert MS-DOS Programming:** Custom-designed segments, runtime program structure, and Interrupt handling. Hardware control using I/O ports.

# KEYBOARD OPERATIONS

## ASCII CONTROL CHARACTERS

This statement from the x86 Assembly book by Kip Irvine is referring to a list of ASCII control characters that are generated when a control key combination is pressed.

ASCII control characters are special characters that are used to control the formatting and communication of data on a screen or printer.

ASCII	HEX VALUE	MEANING	MNEMONIC	CTRL-
0	00h	Null	NUL	Ctrl-@
1	01h	Start of Heading	SOH	Ctrl-A
2	02h	Start of Text	STX	Ctrl-B
3	03h	End of Text	ETX	Ctrl-C
4	04h	End of Transmission	EOT	Ctrl-D
5	05h	Enquiry	ENQ	Ctrl-E
6	06h	Acknowledge	ACK	Ctrl-F
7	07h	Bell	BEL	Ctrl-G
8	08h	Backspace	BS	Ctrl-H
9	09h	Horizontal Tab	HT	Ctrl-I
10	0Ah	Line Feed	LF	Ctrl-J
11	0Bh	Vertical Tab	VT	Ctrl-K
12	0Ch	Form Feed	FF	Ctrl-L
13	0Dh	Carriage Return	CR	Ctrl-M

14	0Eh	Shift Out	SO	Ctrl-N
15	0Fh	Shift In	SI	Ctrl-O
16	10h	Data Link Escape	DLE	Ctrl-P
17	11h	Device Control 1	DC1 (XON)	Ctrl-Q
18	12h	Device Control 2	DC2	Ctrl-R
19	13h	Device Control 3	DC3 (XOFF)	Ctrl-S
20	14h	Device Control 4	DC4	Ctrl-T
21	15h	Negative Acknowledge	NAK	Ctrl-U
22	16h	Synchronous Idle	SYN	Ctrl-V
23	17h	End of Transmission Block	ETB	Ctrl-W
24	18h	Cancel	CAN	Ctrl-X
25	19h	End of Medium	EM	Ctrl-Y
26	1Ah	Substitute	SUB	Ctrl-Z
27	1Bh	Escape	ESC	Ctrl-[

28	1Ch	File Separator	FS	Ctrl-\
29	1Dh	Group Separator	GS	Ctrl-]
30	1Eh	Record Separator	RS	Ctrl-^
31	1Fh	Unit Separator	US	Ctrl-_

32	20h	Space	Space	Ctrl-@
----	-----	-------	-------	--------

33	21h	Exclamation Mark	!	Ctrl-!
34	22h	Quotation Mark	"	Ctrl-"
35	23h	Number Sign	#	Ctrl-#
36	24h	Dollar Sign	\$	Ctrl-\$
37	25h	Percent Sign	%	Ctrl-%
38	26h	Ampersand	&	Ctrl-&
39	27h	Apostrophe	'	Ctrl-'
40	28h	Left Parenthesis	(	Ctrl-(
41	29h	Right Parenthesis	)	Ctrl-)
42	2Ah	Asterisk	*	Ctrl-*
43	2Bh	Plus Sign	+	Ctrl-+
44	2Ch	Comma	,	Ctrl-,
45	2Dh	Hyphen	-	Ctrl--
46	2Eh	Period	.	Ctrl-.

47	2Fh	Slash	/	Ctrl-/
48	30h	Digit 0	0	Ctrl-0
49	31h	Digit 1	1	Ctrl-1
50	32h	Digit 2	2	Ctrl-2
51	33h	Digit 3	3	Ctrl-3
52	34h	Digit 4	4	Ctrl-4
53	35h	Digit 5	5	Ctrl-5
54	36h	Digit 6	6	Ctrl-6
55	37h	Digit 7	7	Ctrl-7
56	38h	Digit 8	8	Ctrl-8
57	39h	Digit 9	9	Ctrl-9
58	3Ah	Colon	:	Ctrl-:
59	3Bh	Semicolon	;	Ctrl-;
60	3Ch	Less Than Sign	<	Ctrl-<

61	3Dh	Equal Sign	=	Ctrl-=
62	3Eh	Greater Than Sign	>	Ctrl->
63	3Fh	Question Mark	?	Ctrl-?
64	40h	At Sign	@	Ctrl-@

The mnemonics and descriptions mentioned in the statement are simply labels and descriptions given to these control characters to make them easier to identify and use in programming.

These labels and descriptions allow programmers to quickly and easily understand the function of each control character, and use them to format their output or communicate data effectively.

Note that the ASCII code for **Ctrl-Hyphen (-)** mentioned in the question is actually 1Fh, and is included in the table above.

## ALT-KEY COMBINATIONS

Key	Hex Value
Alt+A	1E
Alt+B	30
Alt+C	2E
Alt+D	20
Alt+E	12
Alt+F	21
Alt+G	22
Alt+H	23
Alt+I	17
Alt+J	24
Alt+K	25
Alt+L	26
Alt+M	32

Alt+N	31
Alt+O	18
Alt+P	19
Alt+Q	10
Alt+R	13
Alt+S	1F
Alt+T	14
Alt+U	16
Alt+V	2F
Alt+W	11
Alt+X	2D
Alt+Y	15
Alt+Z	2C

Note that these hexadecimal scan codes are generated by holding down the ALT key and then pressing the corresponding letter key.

These codes are often used in programming to create keyboard shortcuts or to enter special characters that are not available on the keyboard.

characters that are not available on the keyboard.

## **KEYBOARD SCAN CODES**

Scan Code	Hex Value	Description
ESC	01	Escape key
1	02	1 key
2	03	2 key
3	04	3 key
4	05	4 key
5	06	5 key
6	07	6 key
7	08	7 key
8	09	8 key
9	0A	9 key
0	0B	0 key
-	0C	Minus key
=	0D	Equals key

Backspace	0E	Backspace key
Tab	0F	Tab key
Q	10	Q key
W	11	W key
E	12	E key
R	13	R key
T	14	T key
Y	15	Y key
U	16	U key
I	17	I key
O	18	O key
P	19	P key
[	1A	Left bracket
]	1B	Right bracket

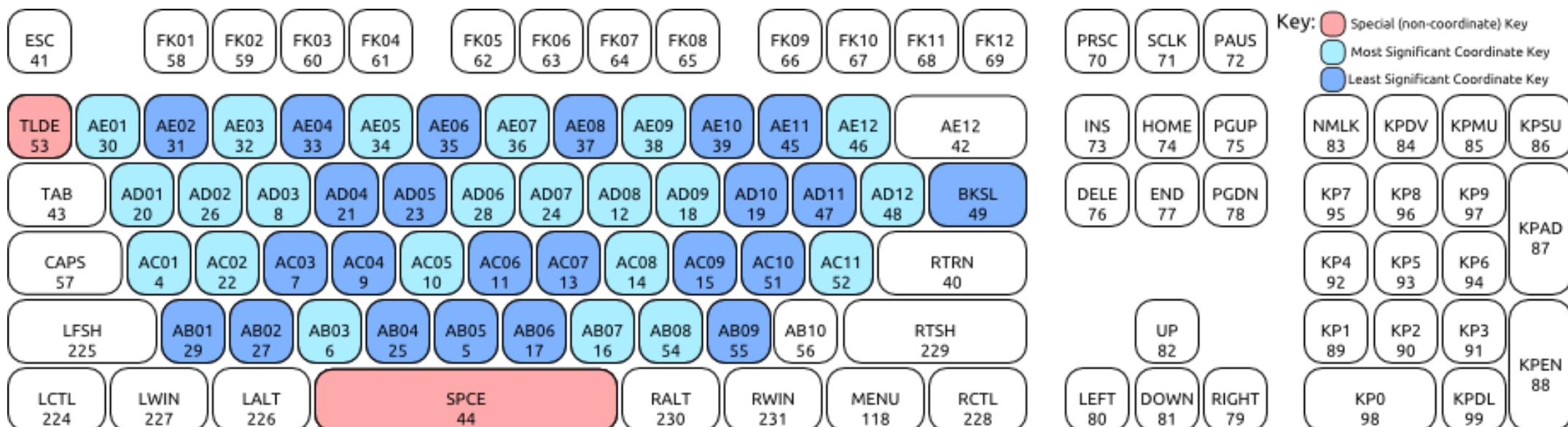
Enter	1C	Enter key
Ctrl	1D	Control key
A	1E	A key
S	1F	S key
D	20	D key
F	21	F key
G	22	G key
H	23	H key
J	24	J key
K	25	K key
L	26	L key
;	27	Semicolon key
'	28	Apostrophe key
`	29	Grave accent key

Shift	2A	Shift key
\	2B	Backslash key
Z	2C	Z key
X	2D	X key
C	2E	C key
V	2F	V key
B	30	B key
N	31	N key
M	32	M key
,	33	Comma key
.	34	Period key
/	35	Slash key
*	37	Asterisk key
Alt	38	Alt key

Space	39	Space bar
Caps Lock		

Keyboard scan codes, ASCII codes, and ALT-key combinations are all related to computer input and keyboard operations.

**Keyboard scan codes** are hexadecimal codes that represent the physical key pressed on a keyboard. When a key is pressed on a keyboard, it generates a keyboard scan code that is interpreted by the computer's hardware and translated into an ASCII code or other character code that is used by software applications.



**ASCII codes** are also hexadecimal codes that represent characters used in the ASCII character set.

This character set includes letters, numbers, punctuation marks, and other special characters. ASCII codes are used to represent text data in computers and are often used in software applications, file formats, and communication protocols.

**ALT-key combinations** are special key combinations that are activated by holding down the ALT key and pressing a specific key on the keyboard. These combinations are often used as shortcuts in software applications to perform specific tasks or commands.

In summary, keyboard scan codes, ASCII codes, and ALT-key combinations are all important aspects of computer input and keyboard operations.

Understanding these codes and combinations is important for developing software applications, working with data files, and communicating with other computer systems.

=====

## **How do programming languages understand ASCII?**

If you don't know the basics of computers, bits, bytes etc, you may find this answer confusing.  
There must be a good indian somewhere teaching this on YouTube.

### **ASCII**

The fundamental character encoding used in most computers.

American Standard Code for Information Interchange.

Character encoding is a method used to represent characters, symbols, and textual information in

computers.

I mean, computers only understand binary's (1 and 0), so we have to map our whole keyboard to numerical codes that the computer can understand. “We map the characters a, b , c, /, ;” to “binary”.

This mapping is what is called character encoding.

So this is the table, check it first then we continue the discussion:

# ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

The character 'A' is represented by the numeric value 65 in ASCII, which is 01000001 in binary.

When you type the letter 'A' on your keyboard, the computer's hardware translates that keystroke into the binary representation 01000001, allowing the computer to understand and process the input.

Hex is another format to represent stuff on the computer, but its just a shorter version of binary. "Binary is too verbose".

Let's get a bit deeper according to that table, character encoding standard that was introduced in the early days of computing.

It uses 7 bits to represent characters, allowing for a total of 128 unique combinations ( $2^7 = 128$ ).

binary number:	0	1	1	1	1	1	1	1
bit values:	128	64	32	16	8	4	2	1
add 'em up:	0	+ 64	+ 32	+ 16	+ 8	+ 4	+ 2	+ 1 = 127

The original ASCII table included control characters (0 to 31) and printable characters (32 to 127).

- Control characters (0 to 31): These are non-printable characters used for various control functions in computing, such as carriage return, line feed, tab, etc.



# ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[END OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]

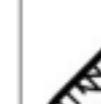
- Printable characters (32 to 127): These represent the visible characters that you see on the screen, including letters (uppercase and lowercase), numbers, punctuation marks, and special symbols. In ASCII, the 128th combination (1111111 in binary or 127 in decimal) is used for the "DEL" (delete) control character.

ASCII has 128 characters represented by 7 bits, but modern character encodings like Unicode have significantly expanded the number of supported characters to meet the needs of global communication and computing.

So we've said ASCII has 33 control characters (0 to 31 including 127) and 95 printable characters (32 to 126).

## UNICODE

When you see this one, you should see “emojis” + “many languages” + “many keyboards”. Its the modern encoding scheme, and uses variable-length encoding.

												
1F926	1F936	1F946	1F956	1F966	1F976	1F986	1F996	1F9A6	1F9B6	1F9C6	1F9D6	1F9E6
												
1F927	1F937	1F947	1F957	1F967		1F987	1F997	1F9A7	1F9B7	1F9C7	1F9D7	1F9E7
												
1F928	1F938	1F948	1F958	1F968		1F988	1F998	1F9A8	1F9B8	1F9C8	1F9D8	1F9E8
												
1F929	1F939	1F949	1F959	1F969		1F989	1F999	1F9A9	1F9B9	1F9C9	1F9D9	1F9E9
												
1F92A	1F93A	1F94A	1F95A	1F96A	1F97A	1F98A	1F99A	1F9AA	1F9BA	1F9CA	1F9DA	1F9EA

Unicode represents a very large character set, currently supporting over 144,000+ characters, including characters from multiple writing systems, symbols, emojis, and special characters.

ASCII allows a total of 128 characters.

ASCII is primarily for representing English characters, but Unicode can represent characters from all writing systems used in the world.

It includes characters from various languages, scripts, and symbols, so its multilingual.

Nepal keyboard unicode:



ASCII uses fixed-length encoding, where each character is represented using 7 bits, Unicode uses variable-length encoding, eg the UTF-8, UTF-16, and UTF-32, to accommodate the larger character set.

These variable-length encodings allows the representation of characters using a variable number of bytes.

NB: UTF-8 is the most widely used encoding for Unicode coz it efficiently represents characters using a variable number of bytes, allowing for compact representation and multilingual support.

using a variable number of bytes, allowing for compact representation and multilingual support.

# Learn Unicode

It's not too hard!

你 ſ ę é  ດ あ

UTF-8 F0 9F 91 8D F0 9F 8F BE

UTF-32 00 01 F4 4D 00 01 F3 FE

To finalise, ASCII is a simple and limited character encoding standard primarily used for representing characters in the English language.

Unicode is a more comprehensive and universal character encoding standard that supports characters from various languages and scripts, making it suitable for multilingual applications and global communication.

## ANSWER:

(YOU WILL GET MORE KNOWLEDGE AND UNDERSTANDING IF YOU DO COMPILER DESIGN)

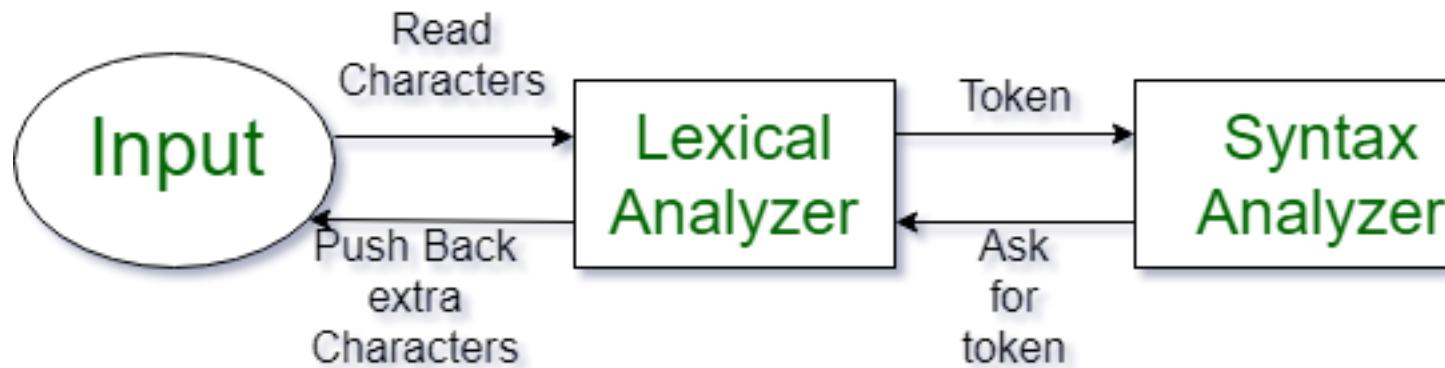
So, you write your code using human-readable characters (letters, numbers, symbols) to represent the instructions and logic for your program.

Text in the source code file is encoded using a specific character encoding scheme.

Lexical analysis is like breaking down a sentence or paragraph into smaller pieces, or "tokens," to make it easier to understand.

During this process, the compiler or interpreter converts the source code into tokens. Tokens are meaningful chunks of the code, like keywords, identifiers, operators, etc.

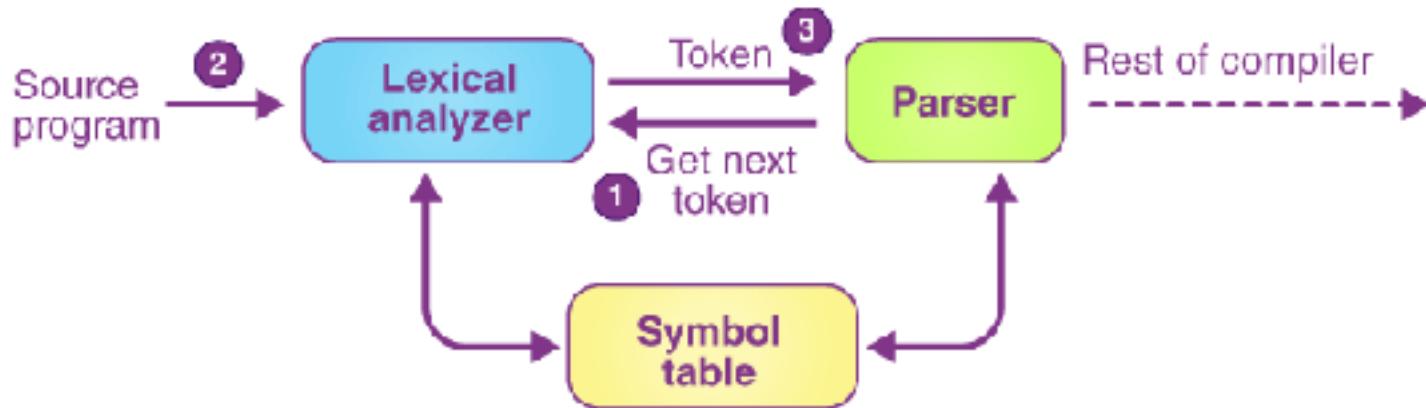
ASCII or UNICODE is used here to map the characters in the source code to their corresponding numerical values.



When you write code, the programming language's lexical analyzer breaks down your code into smaller pieces or "tokens." These tokens can be things like keywords (like "if" or "while"), variable names, numbers, operators.

By breaking down the code into tokens, the programming language can better understand the structure and meaning of your code, just like breaking down a sentence helps you understand its meaning.

"The," "quick," "brown," "fox," "jumps," "over," "the," "lazy," and "dog".



Once the source code is converted into tokens, the compiler (or interpreter in the case of interpreted languages) translates those tokens into machine code or intermediate code, a low-level representation of the program.

Machine code is executed by the computer's CPU, , following the instructions represented by the binary values of the machine code. These binary values are ultimately derived from the ASCII representation of the characters in the source code.

Now you know something new. Not perfect, not full of every detail of the inner workings, but good enough. Bye!

## ***CHAPTER 1: VM PLATFORM/CONVERSIONS/BOOLEAN***

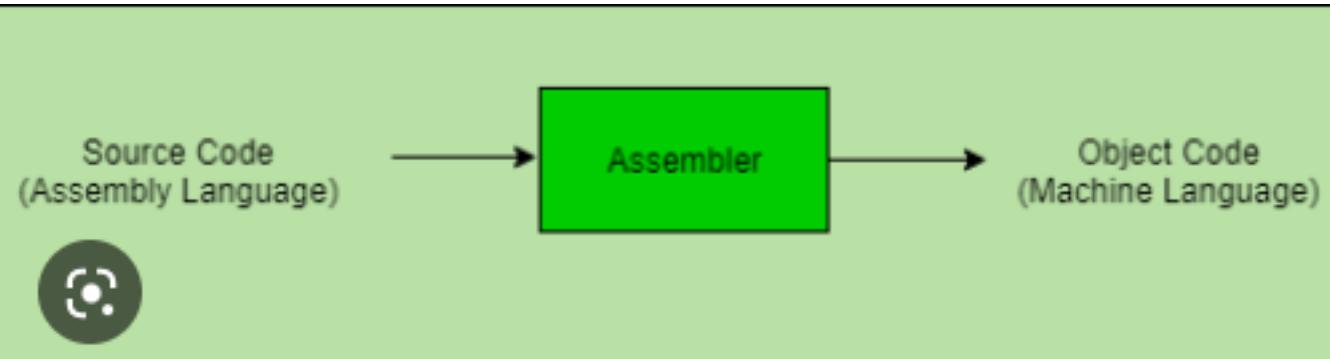
If you are planning to be a C or C++ developer, you need to develop an understanding of how memory, address, and instructions work at a low level.

A lot of programming errors are not easily recognized at the high-level language level.

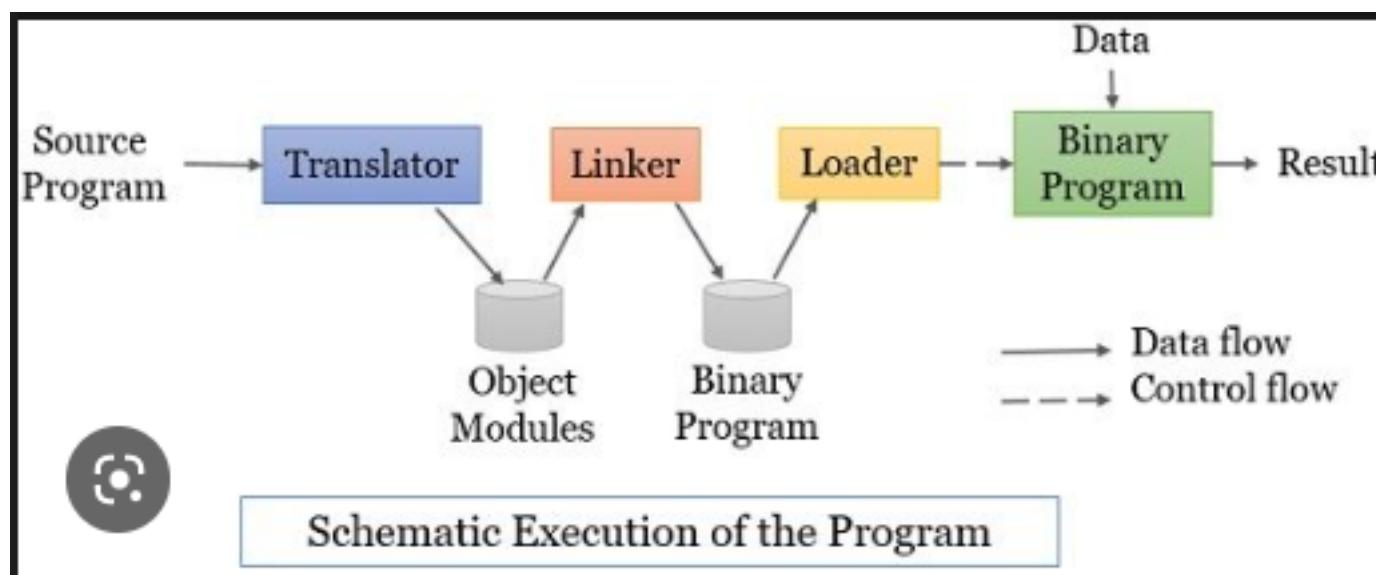
You will often find it necessary to “drill down” into your program’s internals to find out why it isn’t working.

## **ASSEMBLERS, LINKERS, DEBUGGERS**

**Assembler** is a utility program that converts source code programs from assembly language into machine language.



**Linkers** are programs that combine object code files generated by an assembler or compiler into a single executable program. Linkers resolve any external references between the different modules of the program, such as function calls or variable references, and create a single executable file that can be loaded and run by the computer.



**Debuggers**, let you to step through a program while it's running and examine registers and memory. Debuggers are programs that allow developers to find and fix errors in their code. Debuggers provide features such as breakpoints, which allow developers to pause the execution of their program at a

specific point and inspect the state of the program's variables and memory. Debuggers also provide tools for examining the call stack, stepping through code one line at a time, and analyzing crash dumps.



x64\_dbg - File: Sample.exe - PID: 21D4 - Module: sample.exe - Thread: 122C

File View Debug Plugins Options Help



CPU	Log	Breakpoints	Memory Map	Call Stack	Script	Symbols	References	Threads	
RDX → 00000000005AA2A0	55 48 83 EC 20 48 8B EC 90 48 8D 0D 98 49 FF FF E8 EB 68 E6 FF 48 8B 05 44 5F 02 00 48 8B 08 E8 4C 3F FE FF 48 8B 05 35 5F 02 00 48 8B 08 B2 01 E8 0B 6B FE FF 48 8B 05 24 5F 02 00 48 8B 08 48 8B 15 5A 41 FF FF 4C 8B 05 18 62 02 00 E8 4E 3F FE FF 48 8B 05 07 5F 02 00 48 8B 08 E8 4F 41 FE FF E8 AA 06 E6 FF	push rbp sub rsp, 20 mov rbp, rsp nop lea rcx, qword ptr ds:[59EC48] call sample.410BA0 mov rax, qword ptr ds:[5D0200] mov rcx, qword ptr ds:[rax] call sample.58E210 mov rax, qword ptr ds:[5D0200] mov rcx, qword ptr ds:[rax] mov dl, 1 call sample.590DE0 mov rax, qword ptr ds:[5D0200] mov rcx, qword ptr ds:[rax] mov rdx, qword ptr ds:[59E440] mov r8, qword ptr ds:[5D0508] call sample.58E240 mov rax, qword ptr ds:[5D0200] mov rcx, qword ptr ds:[rax] call sample.58E450 call sample.40A9B0	General						

rbp=0

sample.exe[1AA2A0] | ".text":00000000005AA2A0

Address	Hex	ASCII		
000007FCE8221000	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	.....	000000000013FF58	000007FCE686167E
000007FCE8221010	65 48 88 04 25 30 00 00 00 F0 0F BA 71 08 00 48	eH.%0...ð.ºq..H	000000000013FF60	0000000000000000
000007FCE8221020	8B 40 48 0F 83 FE 43 00 00 48 89 41 10 C7 41 0C	.@H..þC..H.A.ÇA.	000000000013FF68	0000000000000000
000007FCE8221030	01 00 00 00 33 C0 C3 90 90 90 90 90 90 90 90 90 90	....3ÃÄ.....	000000000013FF70	0000000000000000
000007FCE8221040	FF F3 48 83 EC 20 FF 49 0C 48 8B D9 75 27 48 C7	ýóH.ì ýI.H.Üü'HC	000000000013FF78	0000000000000000
000007FCE8221050	41 10 00 00 00 48 89 7C 24 30 B8 FE FF FF FF	A....H. \$0.þýy	000000000013FF80	0000000000000000
000007FCE8221060	83 C9 FF F0 OF B1 48 08 8B F8 OF 85 58 8D 01 00	.Éýð.±K..ø.X...	000000000013FF88	000007FCE823C3F1
000007FCE8221070	48 8B 7C 24 30 33 C0 48 83 C4 20 5B C3 90 90 90	H. \$03ÅH.À [Å...	000000000013FF90	0000000000000000
000007FCE8221080	en		000000000013FF98	0000000000000000
			000000000013FFA0	0000000000000000

Command:

Paused

INT3 breakpoint "entry breakpoint" at 00000000005AA2A0!

Together, assemblers, linkers, and debuggers provide essential tools for creating and maintaining executable code, enabling developers to create efficient, reliable software for a wide range of applications.

## FILETYPES CREATED BY MASM

**32-Bit Protected Mode:** 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode programs. From now on, we will simply call this 32-bit mode.

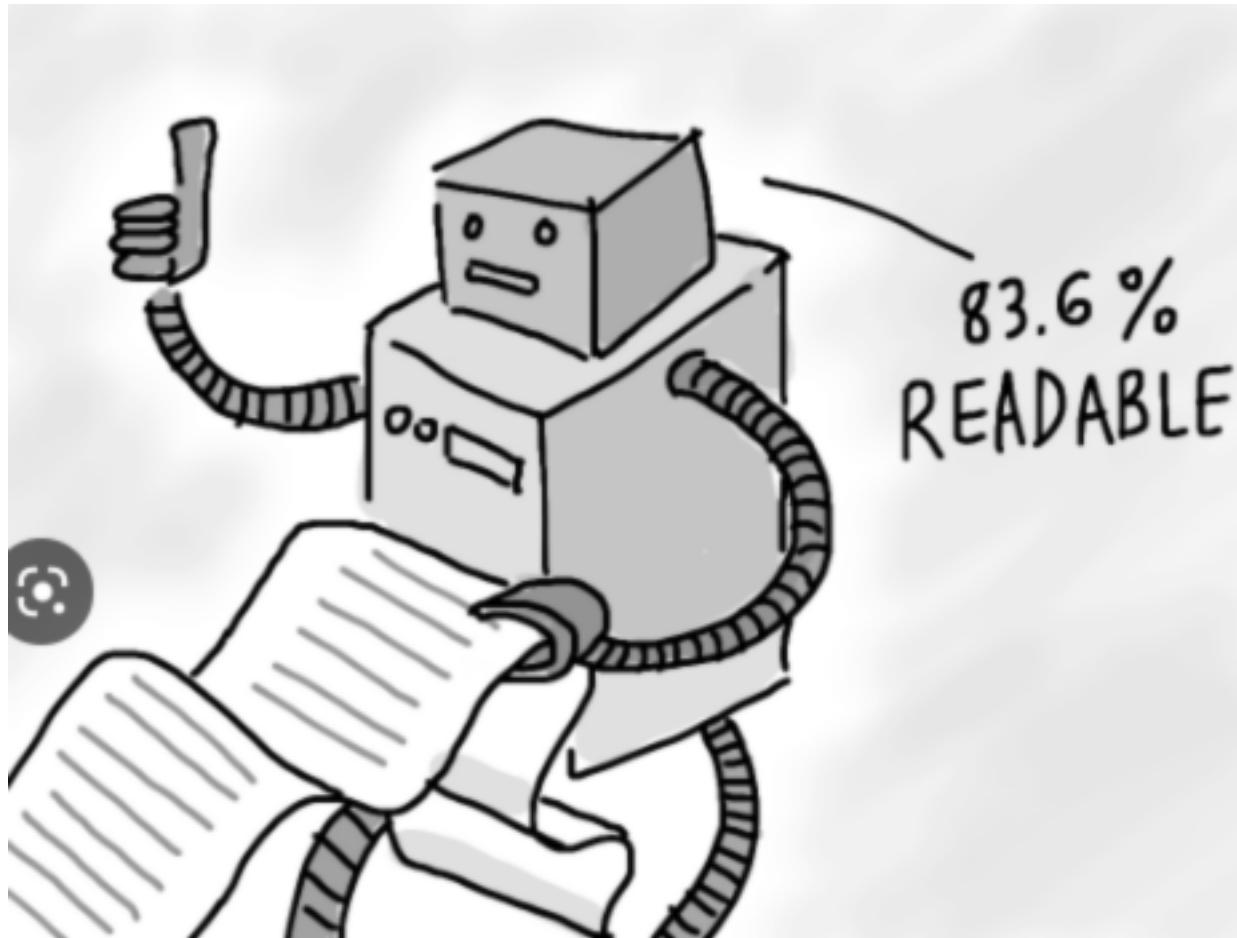
**64-Bit Mode:** 64-bit programs run under all 64-bit versions of Microsoft Windows.

**16-Bit Real-Address Mode:** 16-bit programs run under 32-bit versions of Windows and on embedded systems. Because they are not supported by 64-bit Windows, we will restrict discussions of this mode to Chapters 14 through 17.

## COMPARE AND CONTRAST ASSEMBLY AND MACHINE LANGUAGE

Assembly language and machine language are both low-level programming languages used to interact directly with computer hardware. However, there are some key differences between the two:

- 1. Human readability:** Assembly language is a symbolic representation of machine language instructions that is easier for humans to read and understand, while machine language is a binary code that is only understandable by the computer.



2. **Complexity:** Assembly language is a more complex language than machine language since it involves the use of symbols, labels, and mnemonics to represent the various instructions, registers, and memory addresses. Machine language, on the other hand, is a sequence of binary instructions that are executed by the computer's processor directly.



**3. Portability:** Assembly language is usually specific to a particular processor or architecture, making it less portable than machine language, which can run on any computer that has the same architecture as the code was compiled for.



4. Maintenance: Assembly language code is usually easier to maintain than machine language since it is more human-readable and therefore easier to understand, debug and modify.



**5. Efficiency:** Machine language is the most efficient code as it is the native code that the computer can execute directly. However, assembly language can still be very efficient, and in some cases, it can be more efficient than high-level programming languages.

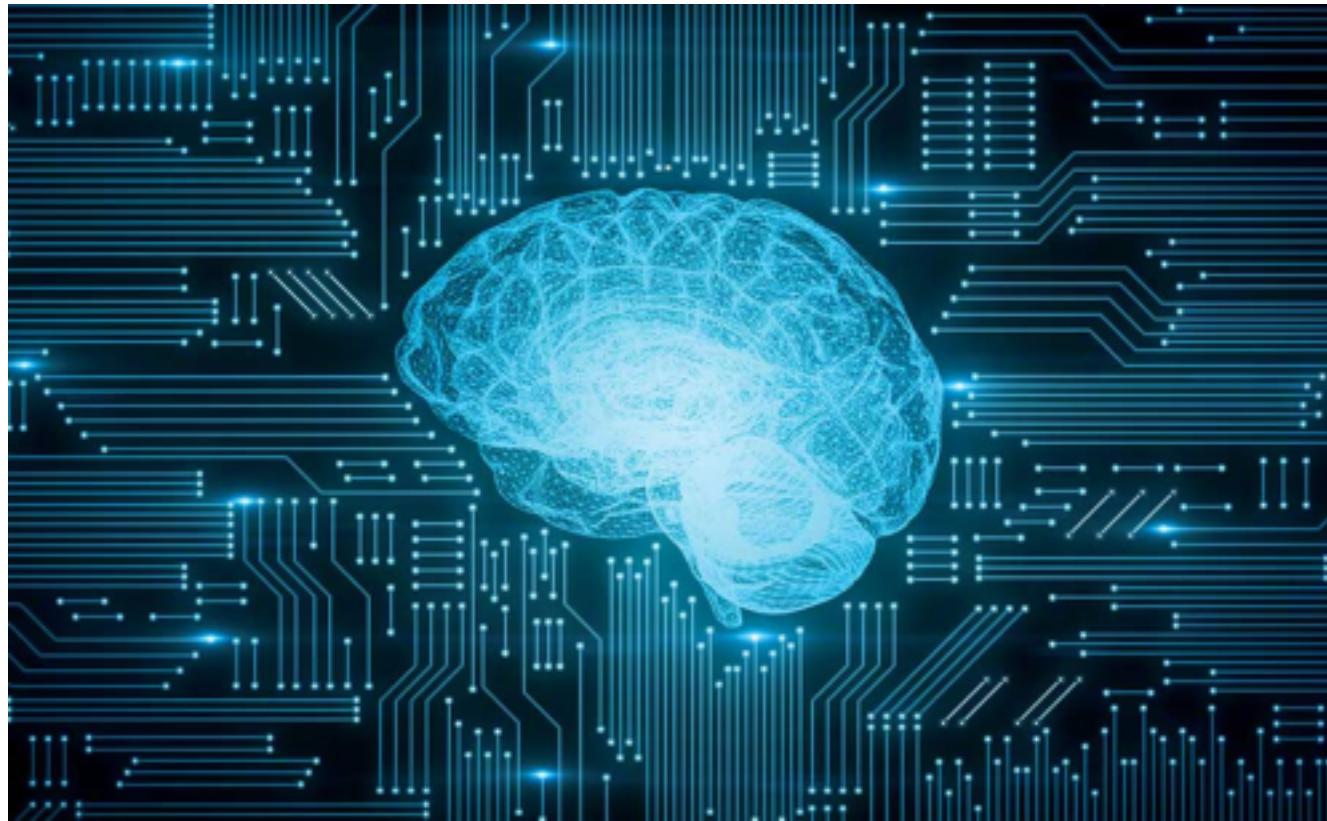


6. Assembly language is **time-consuming** and **error-prone** because it requires manual coding of every single instruction.



# Error-Prone

7. Assembly language is **hardware-dependent**, meaning that code written in one assembly language may not be portable to other systems.



**8. Machine language is the fastest and most efficient way to communicate with computer hardware because it is the language that the CPU understands.**



9. Machine language programs can be more **compact** than assembly language programs because they use binary code, which requires fewer bits than assembly language mnemonics.

compact

A red rectangular background featuring the word "compact" in a large, white, sans-serif font. The word is centered and has a slight shadow or glow effect.

In summary, assembly language is a symbolic representation of machine language that is easier to read and write than machine language, but it requires more effort and time than machine language. Machine language is the fastest and most efficient way to communicate with computer hardware, but it

is difficult to read, write, and maintain.

## HIGH LEVEL vs LOW LEVEL ASSEMBLY

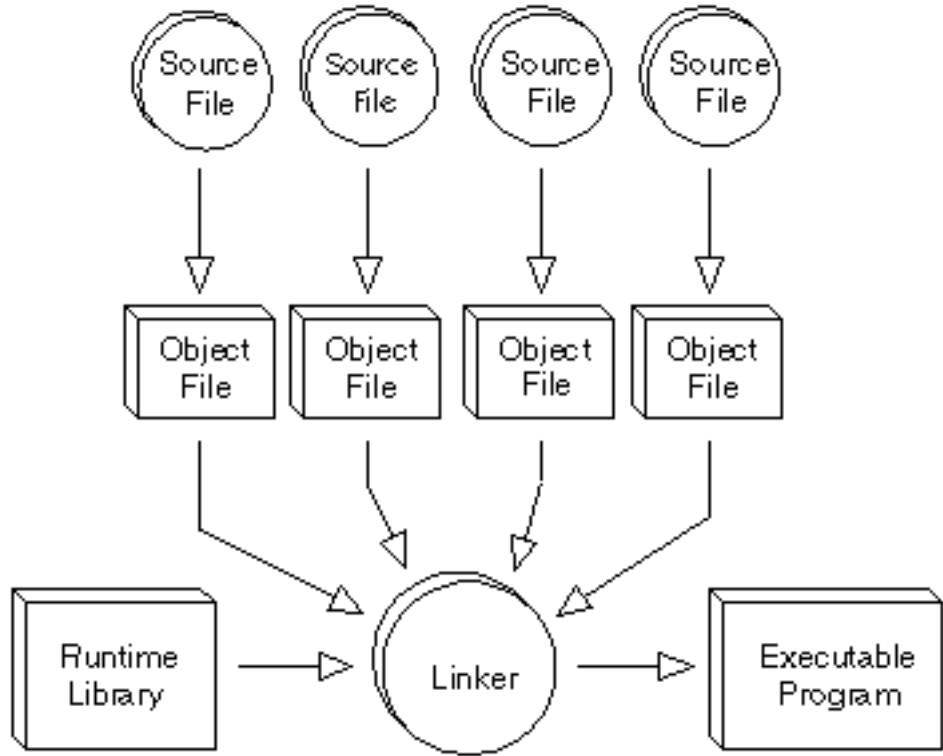
---

C program vs Assembly program to add and multiply(example)

```
;                                int Y;  
;                                int X = (Y+4)*3;  
;  
mov eax, Y                      ;mov variable Y into EAX register  
add eax, 4                      ;add 4 to the variable in EAX register  
mov ebx, 3                      ;mov 3 into EBX register  
imul ebx                         ;multiply EAX and EBX  
mov X, eax                       ;mov contents of EAX into variable X
```

Registers are named storage locations in the CPU that hold intermediate results of operations.

1. **Compilation:** C, C++, and Java are all compiled languages, meaning that the code written in these languages is translated into assembly language or machine language instructions by a compiler before being executed on the computer.



**2. Abstraction:** High-level languages like C, C++, and Java offer a higher level of abstraction than assembly language, allowing programmers to write code that is more readable, maintainable, and easier to understand. These languages provide constructs like variables, functions, and classes that make it easier to manage and manipulate data.



# Abstraction

```
proc initialize_variables()
    set x to 0
    set y to 0
endProc

proc loop_100_times()
    for i=0 to 100
        set x to x+1
        set y to y+1
    endFor
endProc

proc display_variables()
    display x
    display y
endProc
```

3. **Optimization:** Although high-level languages are more abstract than assembly language, modern compilers are capable of generating optimized assembly language code that can execute more efficiently than the equivalent assembly language code written by a human.



**4. Libraries:** High-level languages like C, C++, and Java provide libraries that abstract away many of the low-level details of programming, such as system calls and device drivers. These libraries provide a higher level of abstraction and make it easier to develop complex software applications.



**5. Portability:** One of the key advantages of high-level languages is their portability. Because high-level languages are compiled into machine language or bytecode, they can be executed on any platform that has a compatible runtime environment. This means that code written in C, C++, or Java can be run on a wide range of hardware and software platforms.



**Is Assembly Language Portable?** A language whose source programs can be compiled and run on a wide variety of computer systems is said to be portable. A C++ program, for example, will compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system. Assembly language is not portable, because it is designed for a specific processor family.

Assembly language has fewer syntax rules compared to high-level languages like C++ or Java, but it also requires a lot of **debugging** due to its low-level data access. Assembly language can **access any memory address**, unlike Java, which restricts specific memory addresses.

# DEBUGGING



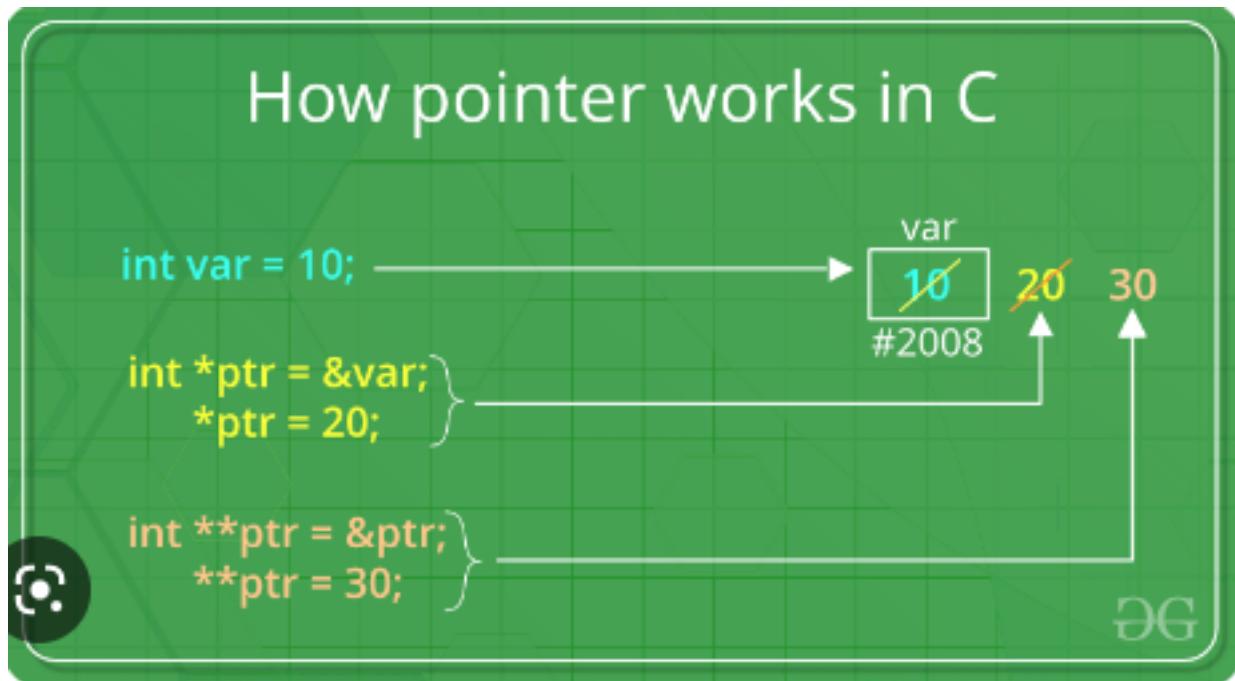
In the early days of programming, **most applications were written partially or entirely in assembly language**, but as programs became more complex, programmers switched to high-level languages. Assembly language is now used to **optimize certain sections of application programs for speed and to access computer hardware**. Overall, assembly language is more adaptable to low-level applications, while high-level languages are better suited for complex programs with a lot of structuring capabilities.

```
01 section .text
02     global_start
03 _start:
04     mov edx, len      ;message length
05     mov ecx, msg      ;message to write
06     mov ebx, 1         ;file descriptor(std_out)
07     mov eax, 4         ;system call number(sys_write)
08     int 0x80          ;call kernel
09
10    mov eax, 1         ;system call number(sys_exit)
11    int 0x80          ;call kernel
12
13 section .data
14
15 msg db 'hello, world!', 0xa ;our dear string
16 len equ $ -msg             ;length of our dear string
```

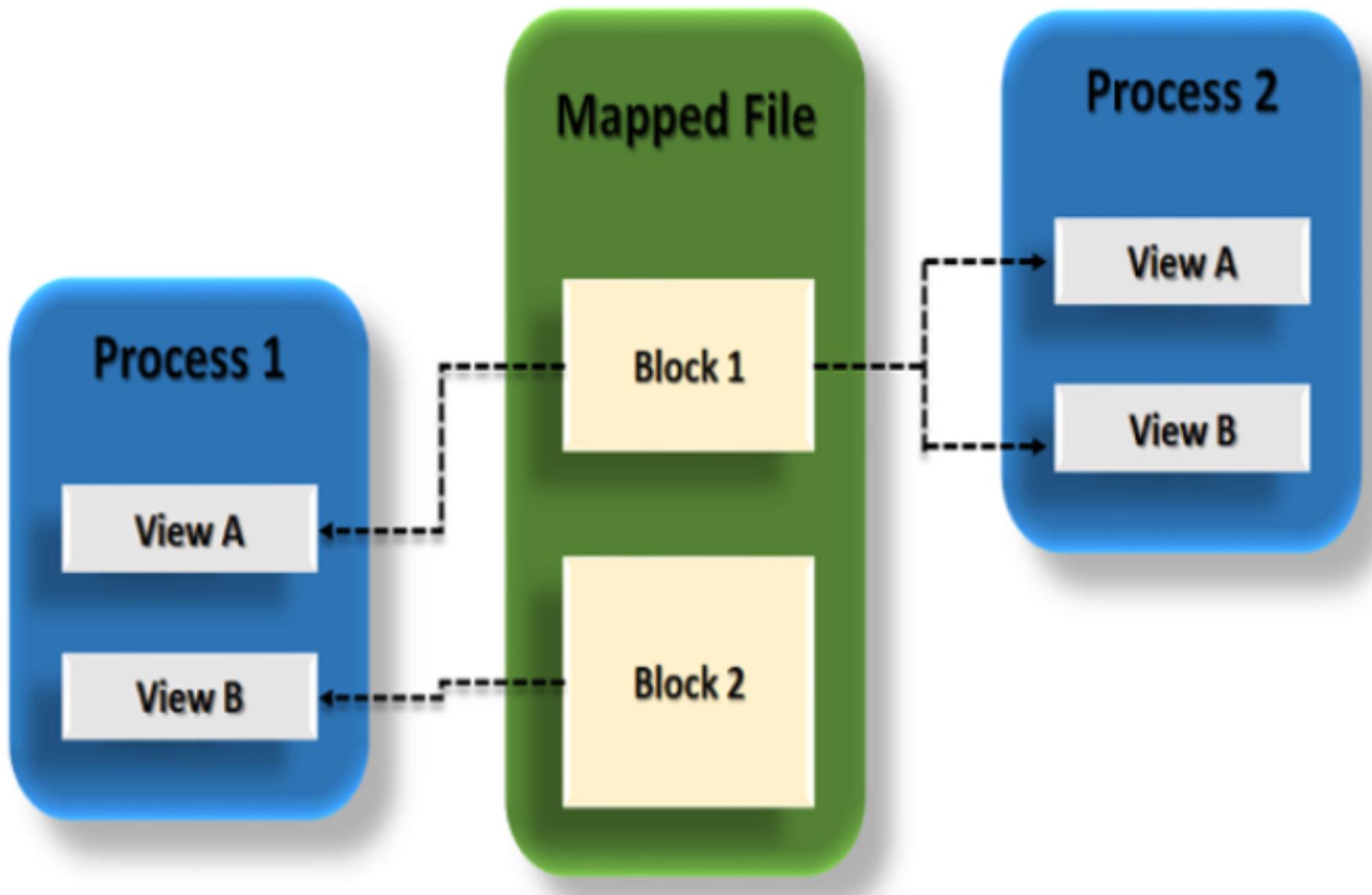
Type of Application	High-Level Languages	Assembly Language
Commercial or scientific application, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	The language may not provide for direct hardware access. Even if it does, awkward coding techniques may be required, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Commercial or scientific application written for multiple platforms (different operating systems).	Usually portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	May produce large executable files that exceed the memory capacity of the device.	Ideal, because the executable code is small and runs quickly.

The C and C++ languages have the unique quality of offering a compromise between high-level structure and low-level details. Direct hardware access is possible but completely non-portable.

In C/C++, direct hardware access is possible through the use of pointers, which allow programmers to directly manipulate memory addresses.

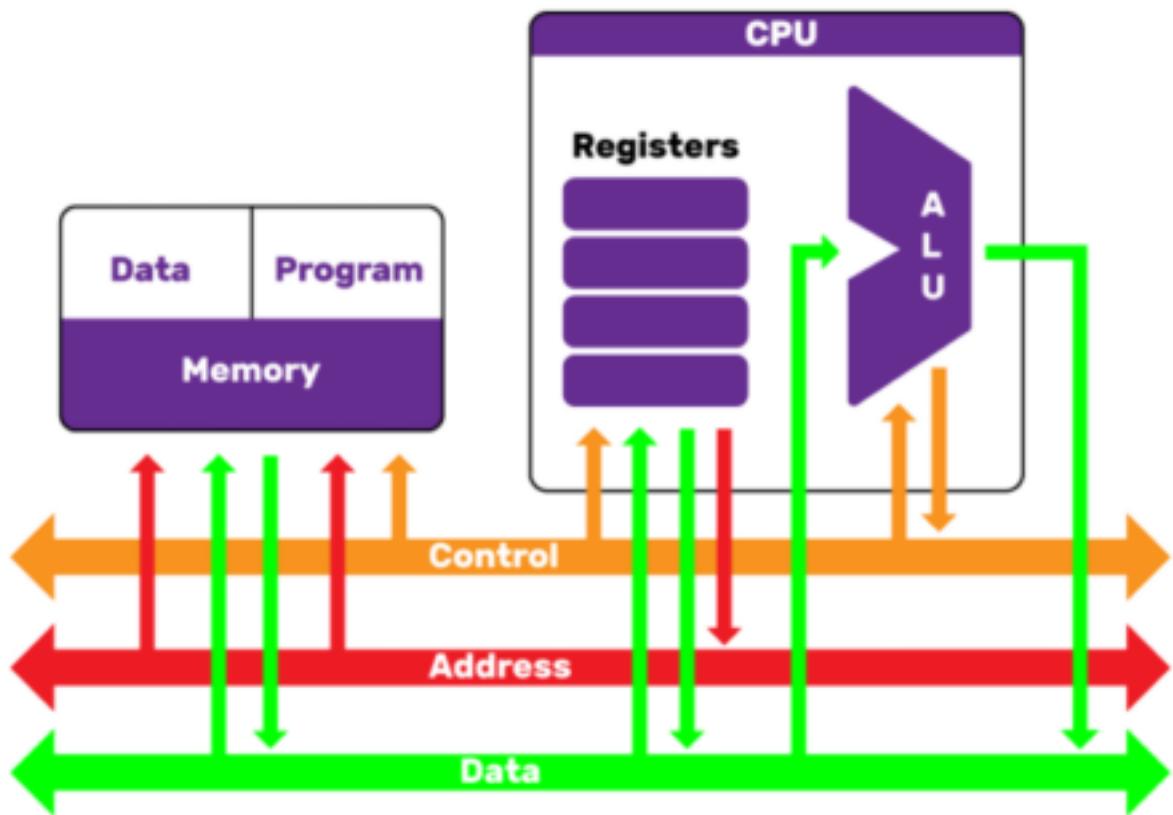


However, this direct hardware access is non-portable because it relies on knowledge of the specific hardware architecture and **memory mapping** of the target system.



Different computer systems have different **memory architectures and arrangements**, and accessing specific memory locations may result in undefined behavior or cause the program to crash on systems.

with different memory layouts.



Additionally, direct hardware access in C/C++ may violate the safety and security of the system, and it may not be allowed in certain environments or operating systems due to security restrictions.



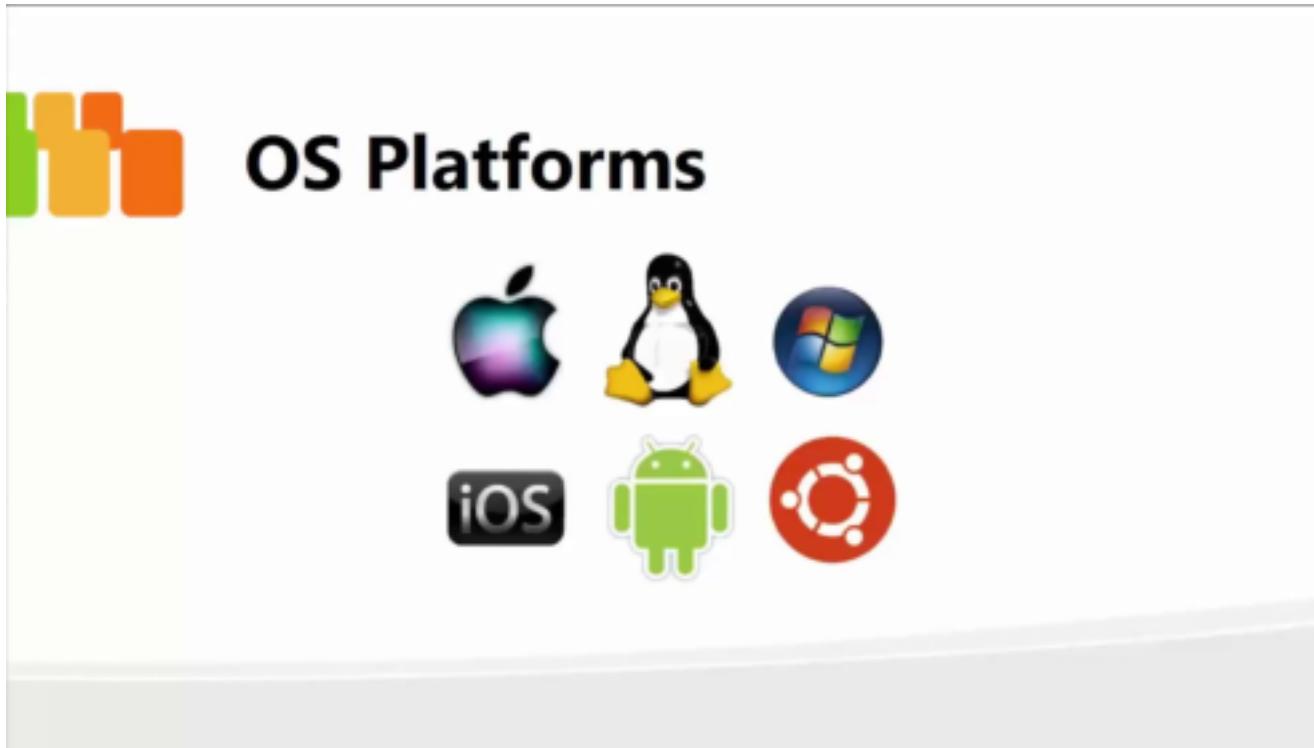
Therefore, to ensure portability and safety, C/C++ provides a set of standard libraries and system calls that **abstract away hardware details** and provide a standardized interface for accessing system resources.

abstraction?

A decorative graphic featuring the word "abstraction?" in a colorful, stylized font. The letters are in various colors: orange, red, brown, yellow, green, blue, and purple. A dashed line with small circular patterns runs horizontally below the text.

This allows programs to be written in a more **portable and platform-independent way**, without having

to rely on specific hardware or memory layouts.



Most C and C++ compilers allow you to **embed assembly language statements** in their code, providing access to hardware details.

The screenshot shows the Microsoft Visual Studio interface. The left pane displays a C++ file named 'stdafx.h' containing assembly code for getting CPU name. The right pane shows the 'Solution Explorer' with two projects: 'MPI' and 'SIMD (Intel C++ 15.0)'. The 'SIMD' project contains files like 'AssemblyOutput.cpp', 'SIMD.cpp', 'stdafx.cpp', and 'ReadMe.txt'. The status bar at the bottom indicates '100 %'.

```
#include "stdafx.h"

string get_cpu_name()
{
    uint32_t data[4] = { 0 };

    __asm
    {
        cpuid;
        mov data[0], ebx;
        mov data[4], edx;
        mov data[8], ecx;
    }

    return string((const char*)data);
}

void assembler()
{
    cout << "CPU is " << get_cpu_name() << endl;
}

int main(int argc, char* argv[])
{
    assembler();
}
```

## ASSEMBLY AND LINKERS WORKING TOGETHER

Assemblers and linkers work together to create executable programs from assembly language source code. Here's how they work together:

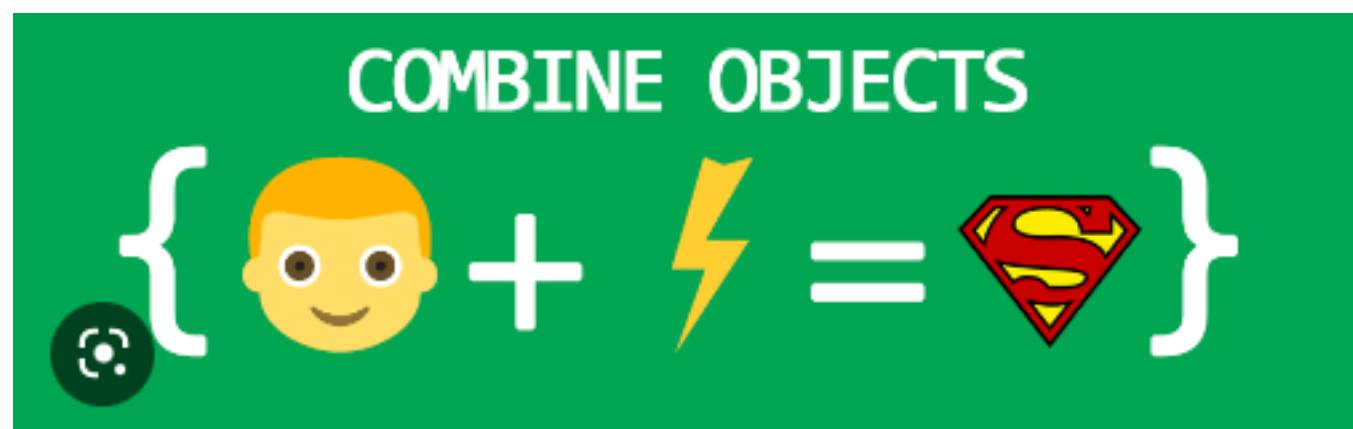
1. Assemblers **convert** assembly language source code into machine language object files that contain executable code and data. During this process, the assembler performs two main tasks: translating the mnemonics and operands of the assembly language instructions into their corresponding machine language opcodes and generating relocation information for the linker.



2. **Relocation information is generated** by the assembler to indicate to the linker where in memory the object code should be loaded and how it should be linked with other object files. This information includes the addresses of external symbols (i.e., symbols that are defined in other object files) that are used in the code, as well as the addresses of any jump or call instructions that reference other parts of the program.



3. The linker takes the object files generated by the assembler and **combines them into a single executable** program. During this process, the linker resolves any external symbol references by finding the addresses of the symbols in other object files and updating the relocation information in the code to reflect the correct addresses.



4. The linker also performs other tasks, such as **merging duplicate sections of code**, resolving references to library functions, and creating the program header and relocation tables.



5. Finally, the linker **produces the executable program**, which can be loaded and run by the operating system.



In summary, assemblers convert assembly language source code into object files, which contain machine language code and relocation information. Linkers then combine these object files into an executable program, resolving any external symbol references and performing other tasks as needed.

## **LEARNING ASSEMBLY HELPS IN UNDERSTANDING OPERATING SYSTEMS**

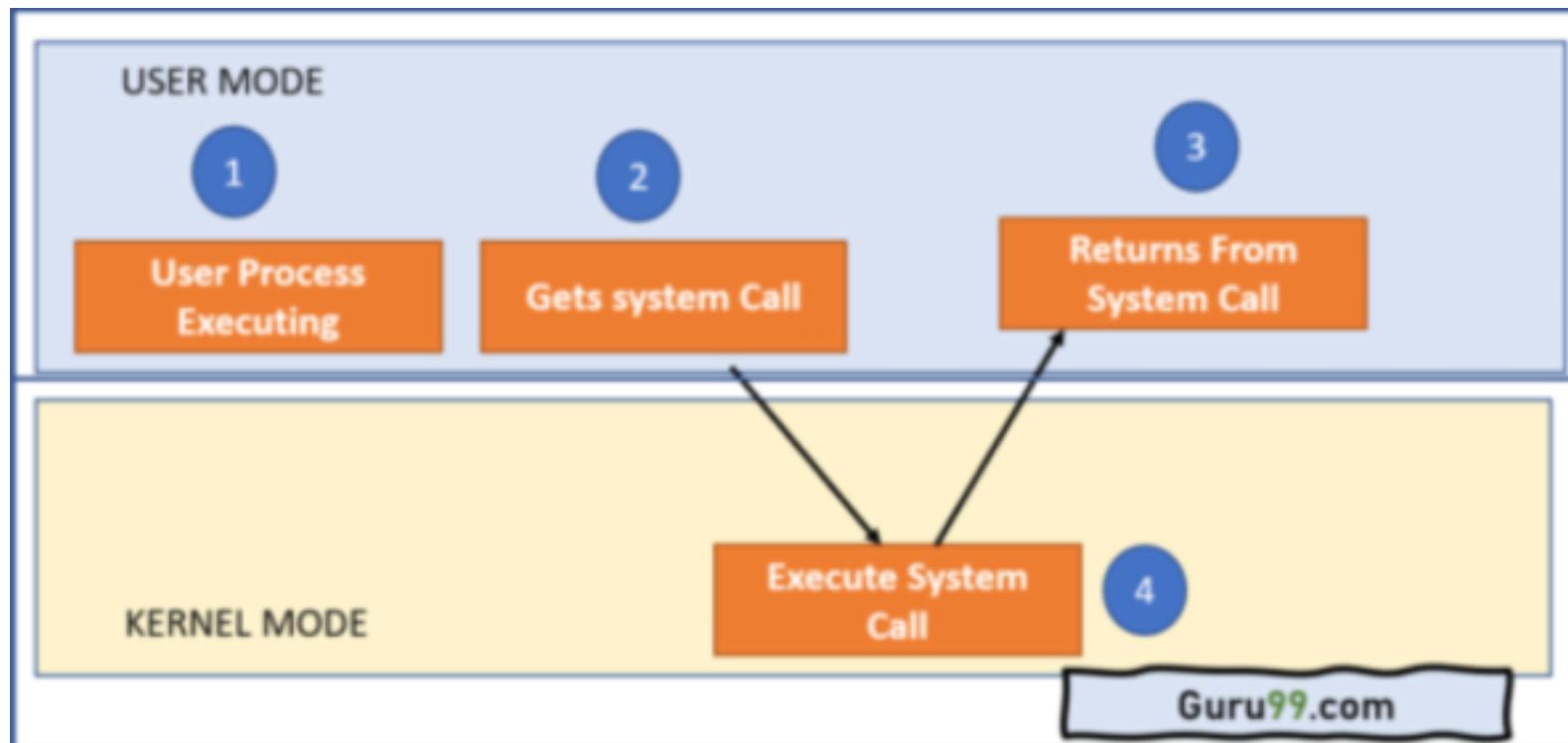
Studying assembly language can enhance your understanding of operating systems in several ways:

- 1. Understanding how hardware interacts with software:** Assembly language provides a low-level understanding of how software interacts with hardware. Since operating systems are responsible for managing hardware resources, understanding how software interacts with hardware is critical to

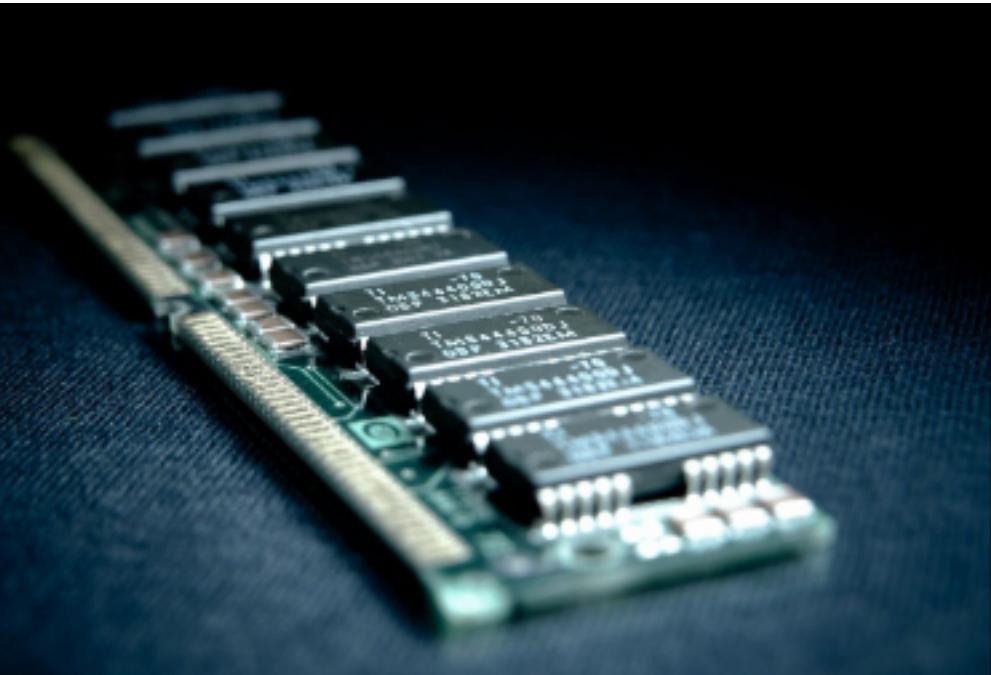
understanding operating systems.



2. **Understanding system calls:** System calls are the primary interface between user-level programs and the operating system. By studying assembly language, you can gain a deeper understanding of how system calls work and how the operating system responds to them.



**3. Understanding memory management:** Memory management is a critical aspect of operating systems, and assembly language provides a deep understanding of how memory is allocated and managed at the lowest level.



**4. Understanding performance optimization:** Assembly language is often used to optimize performance-critical parts of an operating system. By studying assembly language, you can gain insight into how to optimize system performance.



**5. Understanding security:** Understanding assembly language can also help in understanding system vulnerabilities, especially in cases where software exploits hardware or kernel bugs.



In summary, studying assembly language can provide a deeper understanding of how an operating system works at a low level, which can help in various aspects of operating system development, including system calls, memory management, performance optimization, and security.

## ONE-TO-MANY RELATIONSHIPS IN HIGH LEVEL VS LOW LEVEL LANGUAGES

In the context of comparing a high-level language to machine language, a one-to-many relationship refers to the fact that a single instruction in a high-level language may be translated into multiple machine language instructions.

```
100100  
1000000011  
101010101010  
0000000001111
```

VS

```
if(i<5)  
{  
    printf("I am true block ");  
}  
else{  
    printf("I am false block ");  
}
```

Low level language

High level language

High-level languages, such as C++, Java, or Python, are designed to provide a more abstract and human-readable way of writing code, whereas machine language is the binary code that is executed directly by the computer's processor. When a program written in a high-level language is compiled or interpreted, it is translated into machine language instructions that the computer can execute.

In this translation process, a single instruction in a high-level language may be broken down into multiple machine language instructions that perform the same task. For example, a high-level language statement such as "for (i = 0; i < 10; i++)" may be translated into multiple machine language instructions, including load, compare, increment, and jump instructions, to achieve the same result.

Therefore, a one-to-many relationship in this context refers to the fact that a single high-level language instruction may be translated into multiple machine language instructions, which can make the resulting machine code more complex and difficult to read and understand.

## **PORABILITY**

Portability is the ability of a program or application to run on different hardware or software platforms without requiring significant modifications. In the context of programming languages, portability refers to the ability of a programming language to write code that can be compiled or interpreted on different platforms with minimal changes.

# WRITE PORTABLE CODE

A GUIDE TO DEVELOPING SOFTWARE FOR  
MULTIPLE PLATFORMS



Brian Hook



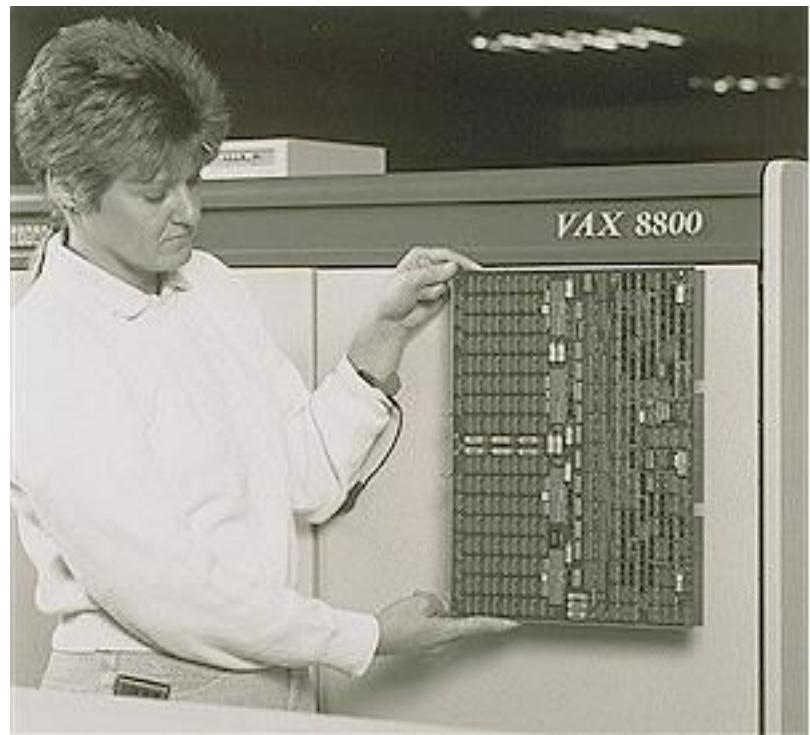
A portable programming language is one that can be used to write code that can run on different

operating systems, hardware architectures, or software environments without requiring significant changes to the code. Portable programming languages achieve this by providing a standardized syntax, data types, and libraries that are common across different platforms. This allows developers to write code that can be compiled or interpreted on different platforms without requiring modifications to the code.

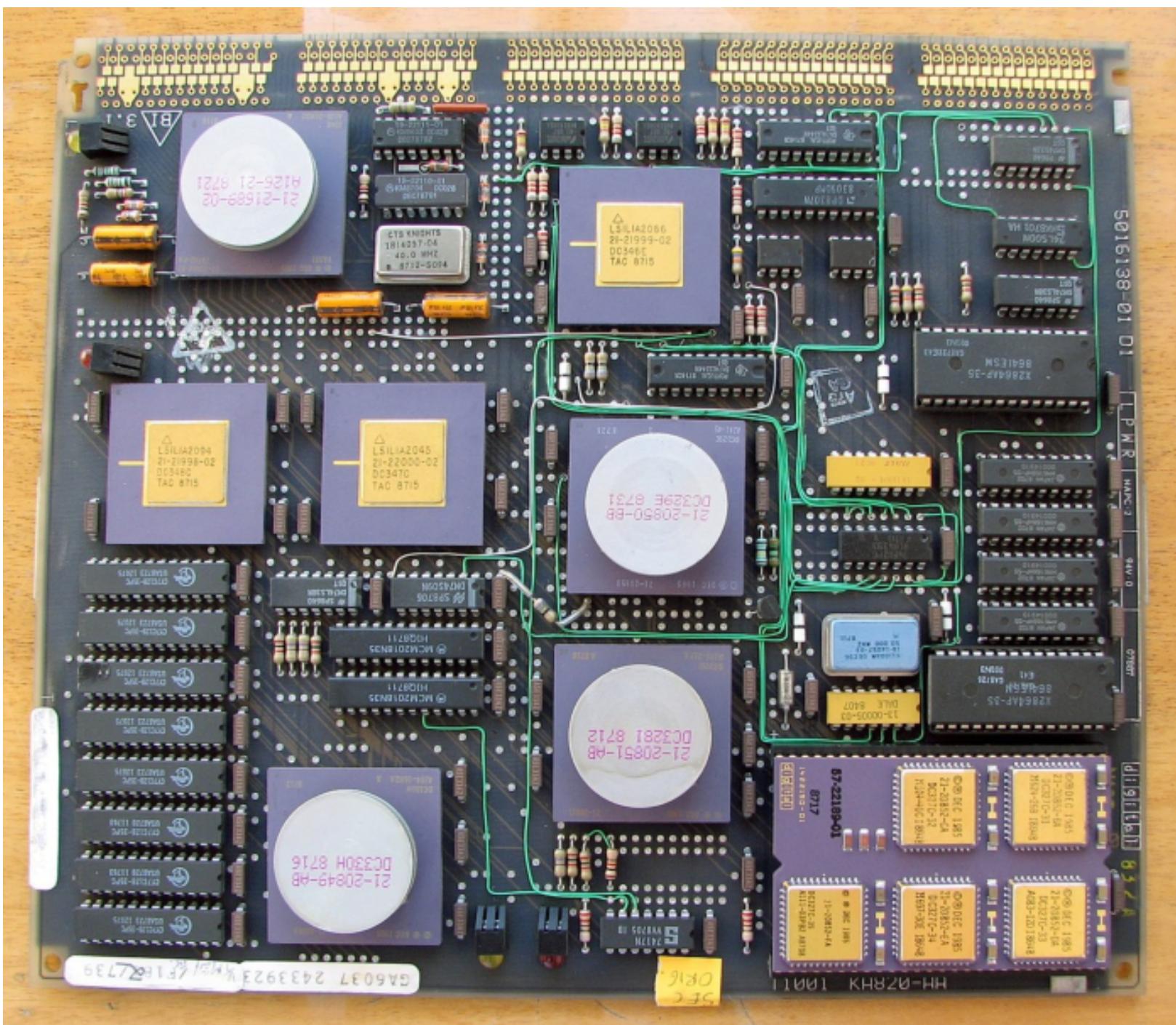
Some examples of portable programming languages include Java, Python, Ruby, and C++. These languages have a large user base and are supported by different hardware and software vendors, which makes them available on different platforms. They also provide a standard set of libraries that can be used across different platforms, which simplifies the development process and reduces the likelihood of errors due to platform-specific dependencies.

In summary, portability is an important consideration when selecting a programming language for a project, as it can reduce development time and cost by allowing code to be used across different platforms with minimal changes. Portable programming languages provide a standard set of syntax, data types, and libraries that can be used across different platforms, which makes them a popular choice for developing cross-platform applications.

**Is the assembly language for x86 processors the same as those for computer systems such as the Vax or Motorola 68x00?**



No, the assembly language for x86 processors is not the same as those for computer systems such as the VAX or Motorola 68x00. While assembly language is a low-level programming language that is specific to the processor architecture, the instructions and syntax used by different processors can vary significantly.



The x86 processor family is used by most personal computers and laptops, while VAX processors were used primarily in large mainframe systems, and the Motorola 68x00 processors were used in many embedded systems and personal computers in the 1980s and 1990s. Each of these processor families has its own unique instruction set and assembly language syntax.



Therefore, if you want to write assembly language code for a specific processor, you need to learn the instruction set and syntax for that processor. While there may be similarities between different assembly languages, it is important to be familiar with the specific instruction set and syntax of the processor you are working with in order to write efficient and effective code.

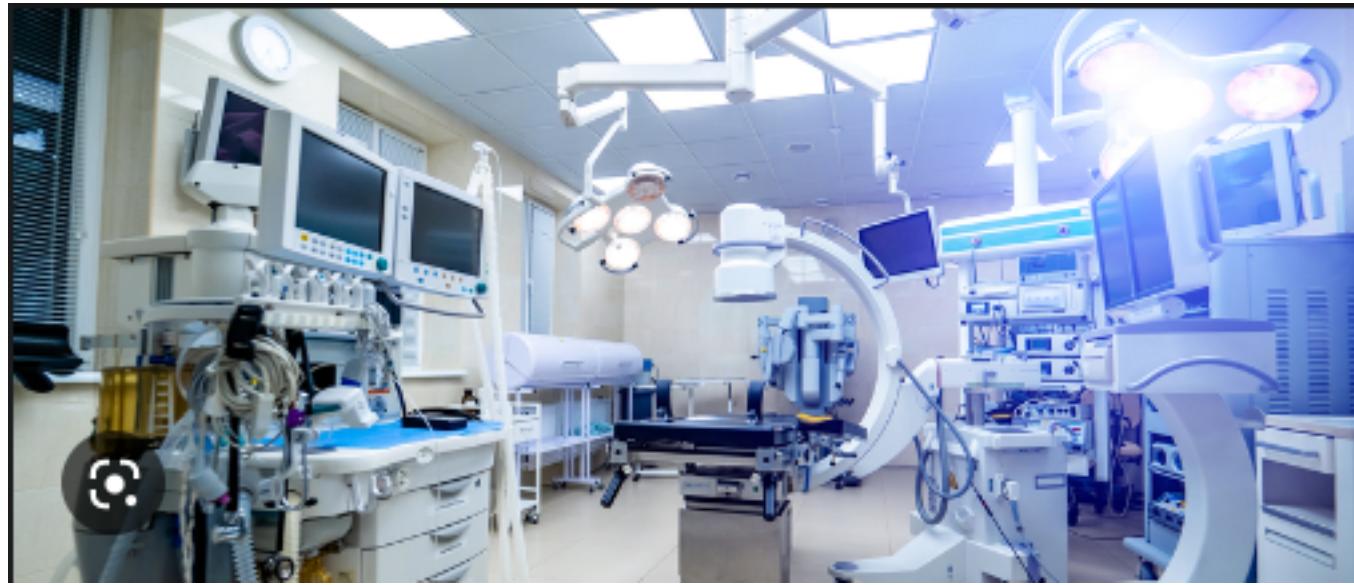
## EMBEDDED SYSTEMS APPLICATION

There are many examples of embedded systems applications. Here are a few:

**Smart home devices:** Many smart home devices, such as thermostats, security systems, and door locks, are powered by embedded systems. These systems enable the devices to communicate with each other and with the internet, and to perform complex functions in real time.

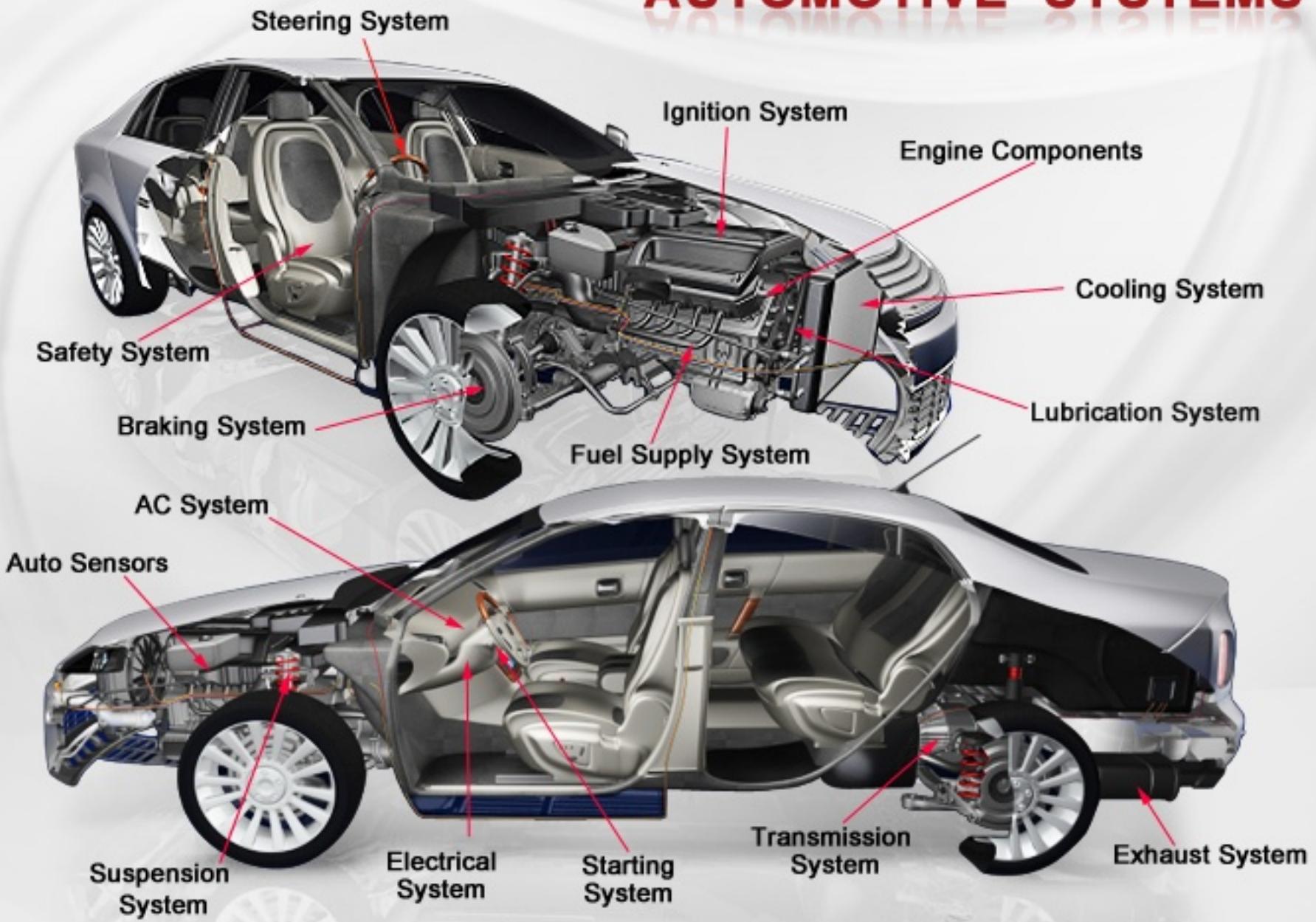


**Medical devices:** Many medical devices, such as blood glucose monitors, pacemakers, and infusion pumps, are powered by embedded systems. These systems enable the devices to monitor and adjust a patient's vital signs and deliver precise amounts of medication.



**Automotive systems:** Modern cars contain numerous embedded systems, including those that control the engine, brakes, and entertainment system. These systems enable the car to operate efficiently and safely, and to provide a comfortable and enjoyable driving experience.

# AUTOMOTIVE SYSTEMS



**Industrial control systems:** Many manufacturing and industrial processes rely on embedded systems to control and automate various tasks, such as monitoring temperature and pressure, controlling motors and actuators, and collecting and analyzing data.





**Consumer electronics:** Many consumer electronics products, such as digital cameras, smartphones, and gaming consoles, are powered by embedded systems. These systems enable the devices to perform complex functions and provide a seamless user experience.



## DEVICE DRIVERS

A **device driver** is a software program that allows an operating system to communicate with a specific hardware device, such as a printer, scanner, or keyboard.

The device driver acts as a translator between the operating system and the hardware device, allowing the operating system to send commands and receive data from the device.

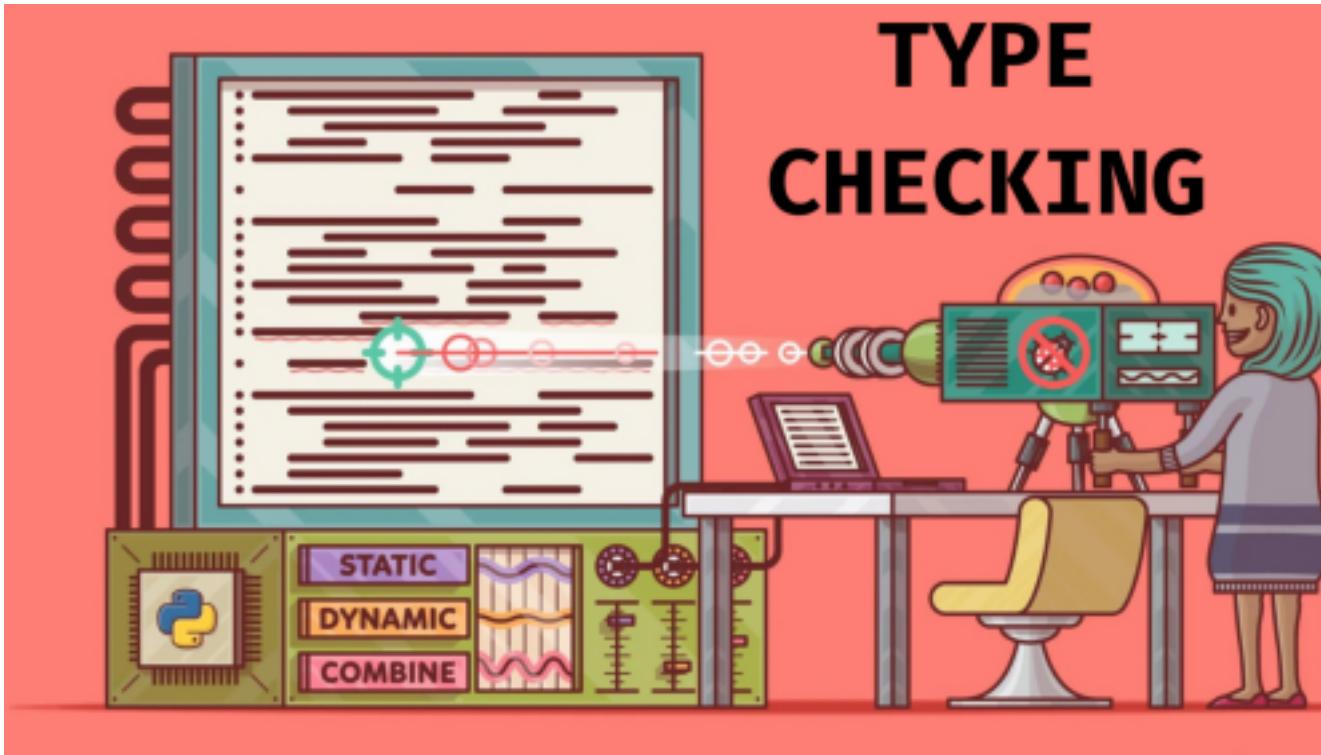


Device drivers are typically written by hardware manufacturers and are specific to the hardware they produce.

The device driver must be compatible with the operating system and version that it is intended to work with.

Without a device driver, the operating system would not be able to communicate with the hardware device, and the device would not function correctly.

Do you suppose type checking on pointer variables is stronger (stricter) in assembly language, or in



## TYPE CHECKING

Type checking on pointer variables is generally stronger (stricter) in C and C++ compared to assembly language.

In C and C++, pointer variables are strongly typed, which means that the compiler enforces strict rules about what type of data can be stored in a pointer variable.

For example, if a pointer variable is declared to point to an integer, then the compiler will prevent the program from assigning a pointer to a different type of data to that variable.

This helps to prevent programming errors and improve program reliability.

In assembly language, pointer variables are typically represented as memory addresses and are not strongly typed.

The assembly language programmer has to manually manage the memory locations where data is stored, and it is up to them to ensure that the correct data types are stored in the correct memory locations.

This can make it easier to introduce programming errors related to type mismatches, such as accessing memory locations using the wrong data type.

Overall, while assembly language provides greater control and flexibility over memory management, it does not have the same level of type checking and safety features that are present in C and C++.

## TWO APPLICATIONS BETTER SUITED WITH ASSEMBLY

Assembly language is a low-level programming language that provides direct control over the hardware of a computer. It is generally more difficult to work with than high-level languages such as C and Java, but it can be advantageous in certain situations. Here are two types of applications that might be better suited to assembly language:

- 1. Operating system components:** Certain parts of an operating system, such as device drivers or system utilities, require low-level access to computer hardware. Assembly language can provide the level of control and performance needed for these components, as they need to be efficient and interact directly with the underlying hardware.

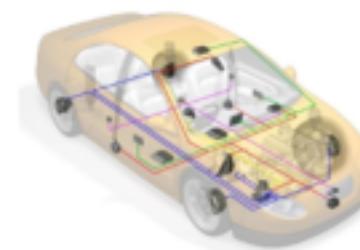
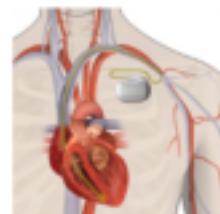


---

**2. Real-time systems:** Real-time systems, such as those used in robotics or industrial control applications, require very fast and precise responses to inputs. Assembly language can provide the level of control and performance needed for these systems, as they need to respond quickly and deterministically to changes in the environment.

# Real-Time Systems

- Controllers in planes, cars, plants, etc. are expected to finish their tasks within **reliable time bounds**
- It is essential that an upper bound on the execution times of all **tasks** is known: commonly called the **worst-case execution time (WCET)**, computed at the code level
- WCET of tasks prerequisite for scheduling analysis at **system level** (e.g. SymTA/S from Symtavision)



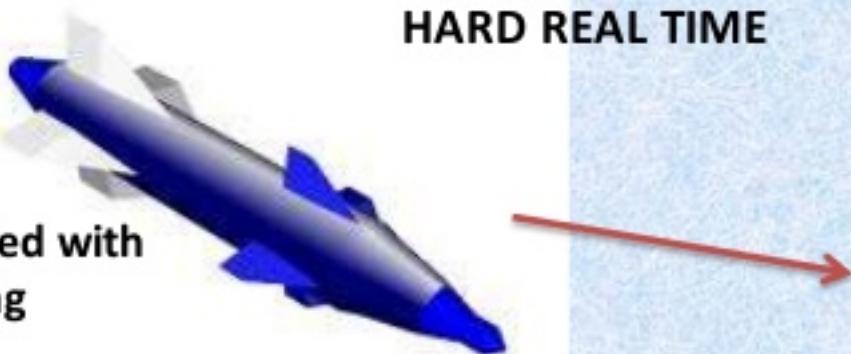
In both of these cases, the benefits of using assembly language come from the level of control and performance it offers, as well as the ability to optimize code for specific hardware platforms. However, it's worth noting that assembly language is generally less productive than high-level languages, and can be more difficult to debug and maintain.

# *Real Time Embedded Systems*

A system in which work has to be done in a specific time period.

HARD REAL TIME

Missile  
embedded with  
a tracking  
system



Aircraft

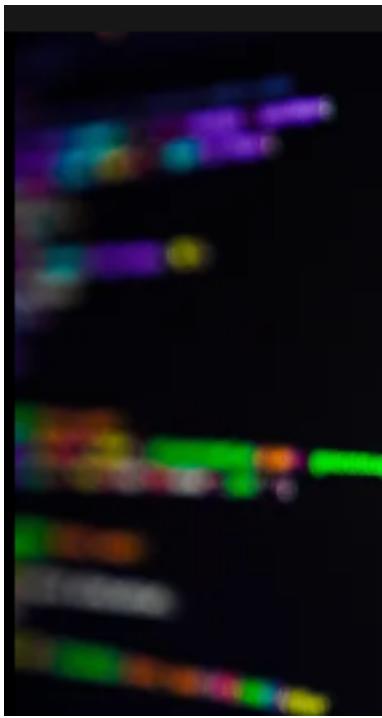
SOFT REAL TIME



DVD PLAYER

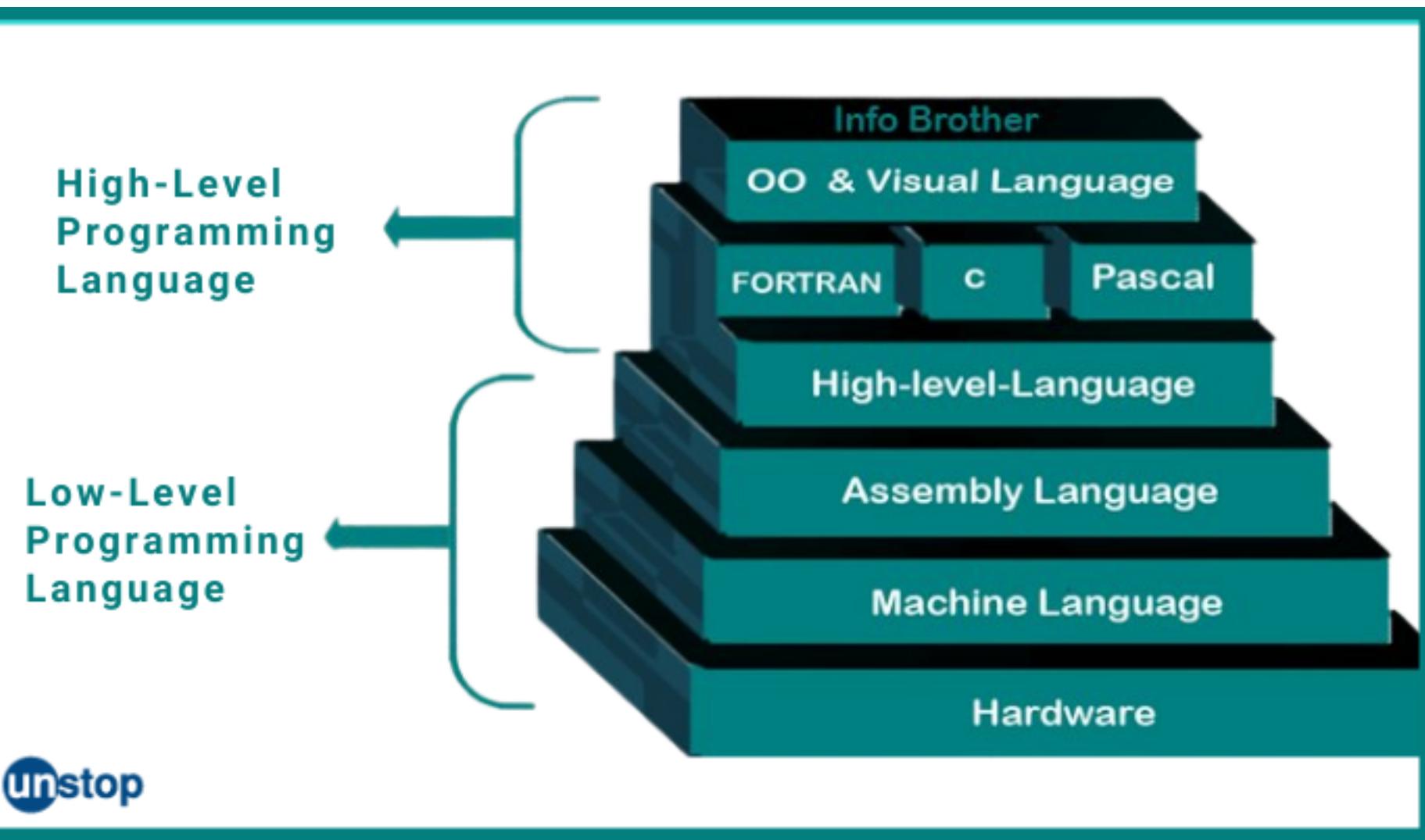
WHY HLL IS NOT SUITED FOR PROGRAM ACCESSING PRINTERS

A high-level language is designed to provide an abstraction layer between the programmer and the hardware, which makes it easier to write portable and maintainable code. However, this abstraction also comes at a cost, as it can **limit the level of control and access** that the programmer has over the hardware.



```
1 const fetch = require('node-fetch');
2 const log = require('loglevel');
3 let embed;
4
5 function transformHTML(html) {
6   // Promise.resolve() is used here to return a promise
7   return new Promise((resolve, reject) => {
8     resolve(html);
9   });
10 }
11
12 function resolveHeaders(headers) {
13   return prev => {
14     const header = headers[0];
15     const children = headers[1];
16     if (!header) {
17       return prev;
18     }
19     const heading = header.toLowerCase();
20     if (heading === 'content-type') {
21       prev.setHeader('Content-Type', children[0]);
22     } else if (heading === 'content-length') {
23       prev.setHeader('Content-Length', children[0]);
24     }
25   };
26 }
27
28 function handleResponse(response) {
29   const { status, headers, body } = response;
30   const statusText = statusTextMap.get(status);
31   const statusLine = `${status} ${statusText} ${headers}`;
32   log.info(`Status Line: ${statusLine}`);
33   log.info(`Headers: ${headers}`);
34   log.info(`Body: ${body}`);
35 }
```

Directly accessing a **printer port** is an example of a task that **requires low-level access to the hardware**, which is not well-suited to high-level languages. High-level languages such as Java or Python provide a high level of abstraction, which makes it difficult to directly manipulate hardware devices such as printer ports. In contrast, assembly language or a low-level language such as C provide a much lower level of abstraction, allowing for direct access to hardware ports and registers.



Directly accessing a printer port also involves working with specific I/O addresses, interrupts, and other low-level hardware details. These details are generally not exposed in high-level languages, as the language abstracts away these low-level details in favor of providing higher-level abstractions.

In summary, a high-level language is not ideal for writing a program that directly accesses a printer port because it **lacks the low-level access and control required to directly manipulate**

**hardware devices.** Instead, a low-level language such as assembly or C would be better suited for this task, as it provides the necessary low-level access and control required to work with hardware devices.

## LARGE APPLICATIONS DEVELOPMENT AVOIDS ASSEMBLY

Assembly language is a low-level programming language that provides direct control over computer hardware. While it has certain advantages over higher-level languages, it is not typically used when writing large application programs for the following reasons:

- **Complexity:** Assembly language programming is much more complex and time-consuming than programming in higher-level languages. Writing large applications in assembly language requires a great deal of skill, effort, and attention to detail.



- **Maintainability:** Assembly language code is difficult to read, understand, and maintain. This is because it is written in terms of low-level hardware operations, rather than high-level abstractions that are easier to reason about.

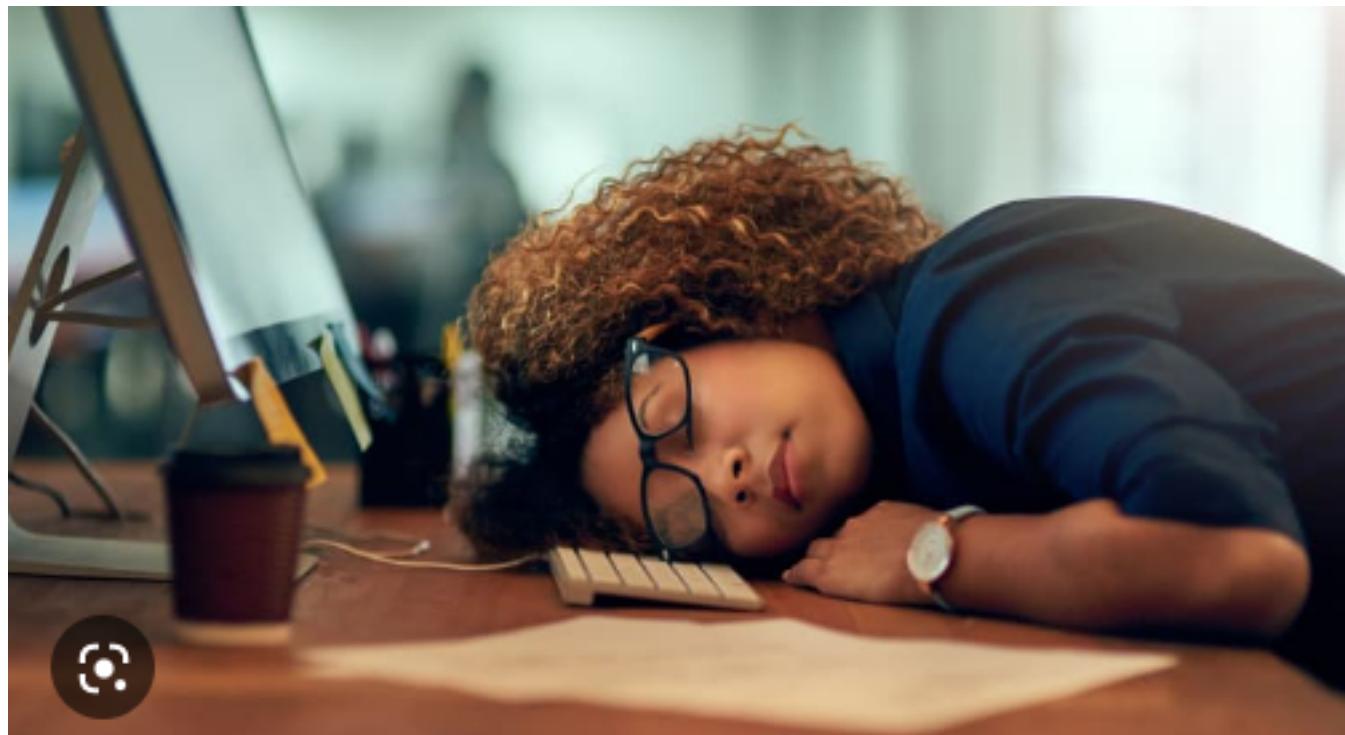


- **Portability:** Assembly language code is not portable across different hardware platforms. This means that code written for one type of hardware may not work on another type of hardware, which makes it more difficult to develop and maintain large applications.



# Portability

- **Productivity:** Assembly language programming is not as productive as programming in higher-level languages. This is because it requires more code to accomplish the same task, and because it lacks the high-level abstractions and features that make programming in higher-level languages more efficient.



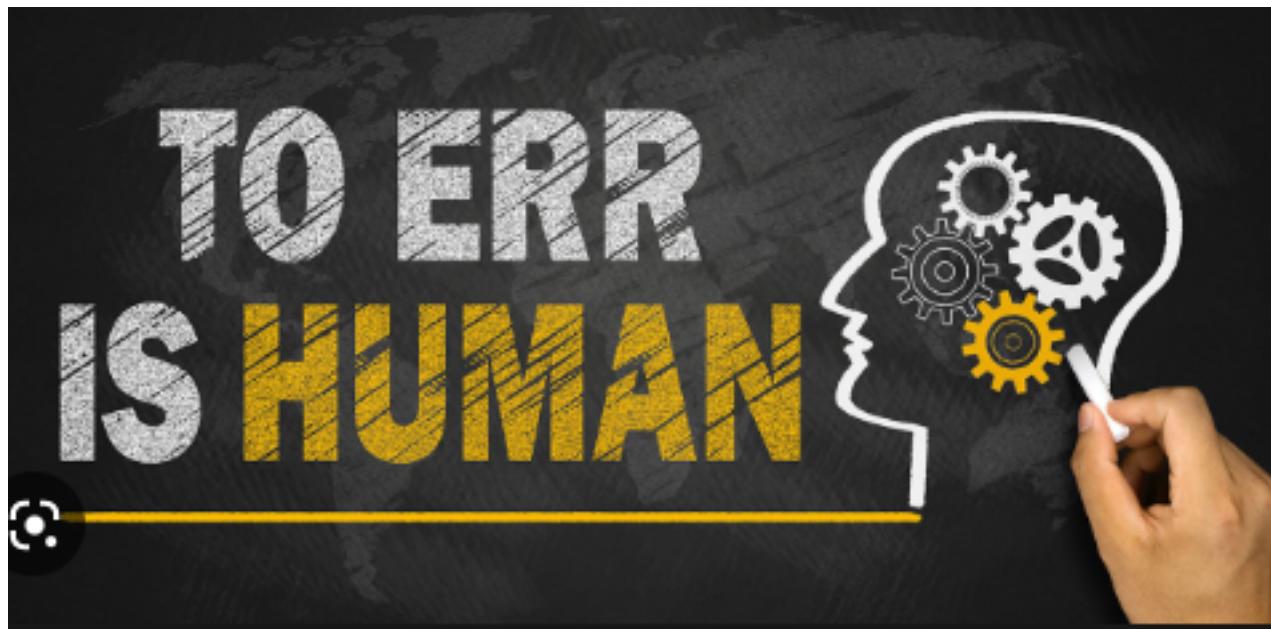


- **Debugging:** Debugging assembly language code can be difficult and time-consuming, especially in large applications. This is because there are no high-level abstractions or debugging tools available to help with the process.

The screenshot shows the Visual Studio Code interface with the 'app.js' file open in the editor. The 'Debug' sidebar on the left is expanded, showing sections for 'RUN', 'VARIABLES', 'WATCH', 'CALL STACK', 'LOADED SCRIPTS', and 'BREAKPOINTS'. The 'CALL STACK' section shows a single entry: 'Launch Program: www [10868] RUNNING'. The main editor area shows the code for 'app.js', with line 10 highlighted as the current line of execution. The status bar at the bottom indicates the file is 'master', has 0 changes, and is set to 'Launch Program (myExpressApp)'. The top navigation bar includes 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', and 'Help'. The top right corner shows the title 'app.js - myExpressApp - Visual Studio Code' and standard window controls. A red box highlights the 'Launch Program' button in the top-left of the Debug sidebar, the toolbar icons above the editor, the 'DEBUG CONSOLE' tab, and the 'Debug side bar'.

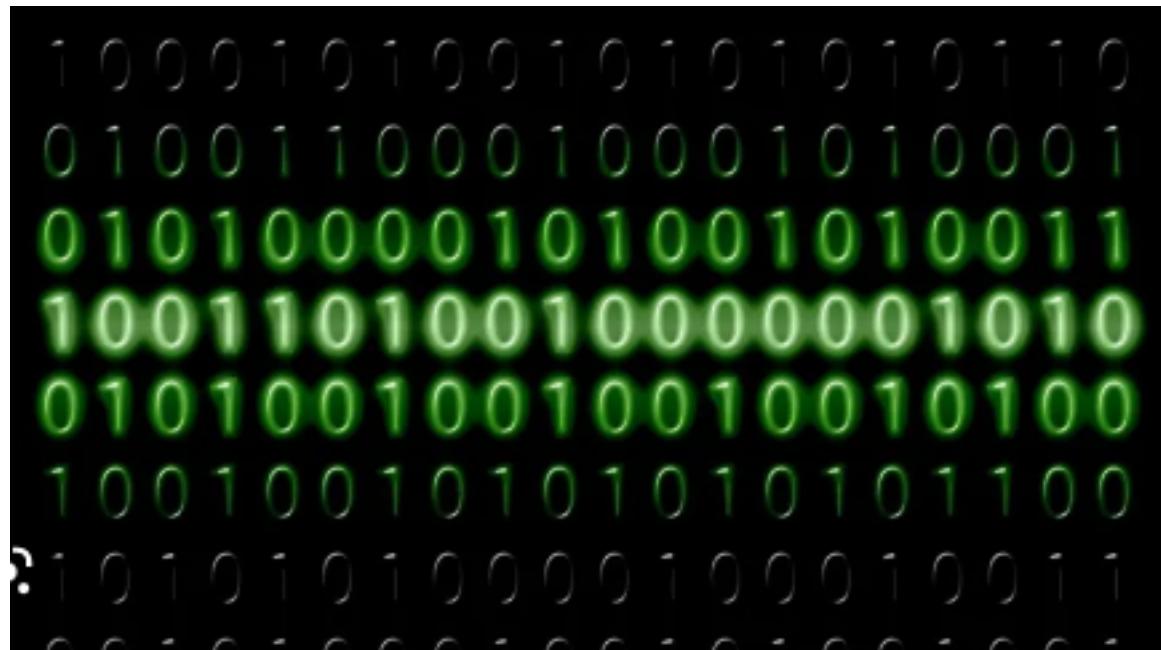
```
1 var createError = require('http-errors');
2 var express = require('express');
3 var path = require('path');
4 var cookieParser = require('cookie-parser');
5 var logger = require('morgan');
6
7 var indexRouter = require('./routes/index');
8 var usersRouter = require('./routes/users');
9
10 var app = express();
11
12 // view engine setup
13 app.set('views', path.join(__dirname, 'views'));
14 app.set('view engine', 'nunjucks');
```

- **Human Error:** Assembly language is more susceptible to human error, as the programmer is responsible for managing memory and performing other low-level operations that are typically handled automatically in higher-level languages. This can lead to bugs and other issues that can be difficult to identify and fix.



VIRTUAL MACHINE CONCEPT

- The **virtual machine concept** explains how a computer's hardware and software are related.
- The most basic function of a computer is executing programs in its **native machine language (L0)**.



```
10001010010101010110  
01001100010001010001  
01010000101001010011  
10011010010000001010  
01010010010010010100  
10010010101010101100  
?10101010000100010011  
00100010010100100100
```

- L0 is difficult to use, so a **new language, L1**, could be constructed that is easier to use.

```
* 0 - deliver
* 1 - block
*/
static int icmp_filter(const struct sock *sk, const struct sk_buff *skb)
{
    struct icmphdr *_hdr;
    const struct icmphdr *hdr;
    __u32 data;

    hdr = skb_header_pointer(skb, skb_transport_offset(skb),
                             sizeof(_hdr), &_hdr);
    if (!hdr)
        return 1;

    if (hdr->type < 32) {
        __u32 data = raw_sk(sk)->filter.data;

        return ((1U << hdr->type) & data) != 0;
    }

    /* Do not block unknown ICMP types */
    return 0;
}
```

- L1 can be executed by a virtual machine (VM1) that emulates the functions of a physical or virtual computer.

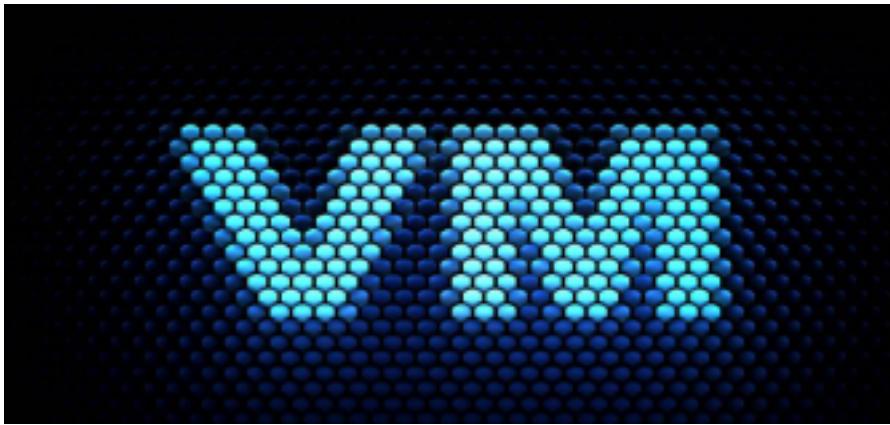
JDK (javac, jar, debugging tools, APIs)

JRE (java, javaw, libraries)

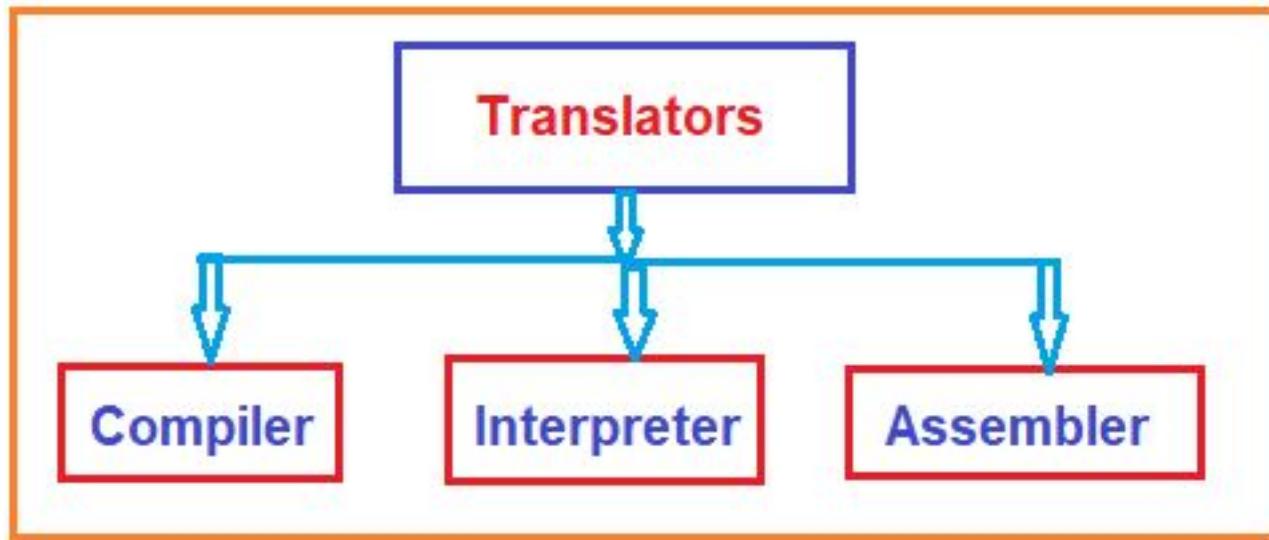
JVM

JIT Compiler

- VM1 can be constructed of either hardware or software, and programs can be written for it.
- If VM1 is practical to implement as an actual computer, programs can be executed directly on the hardware.



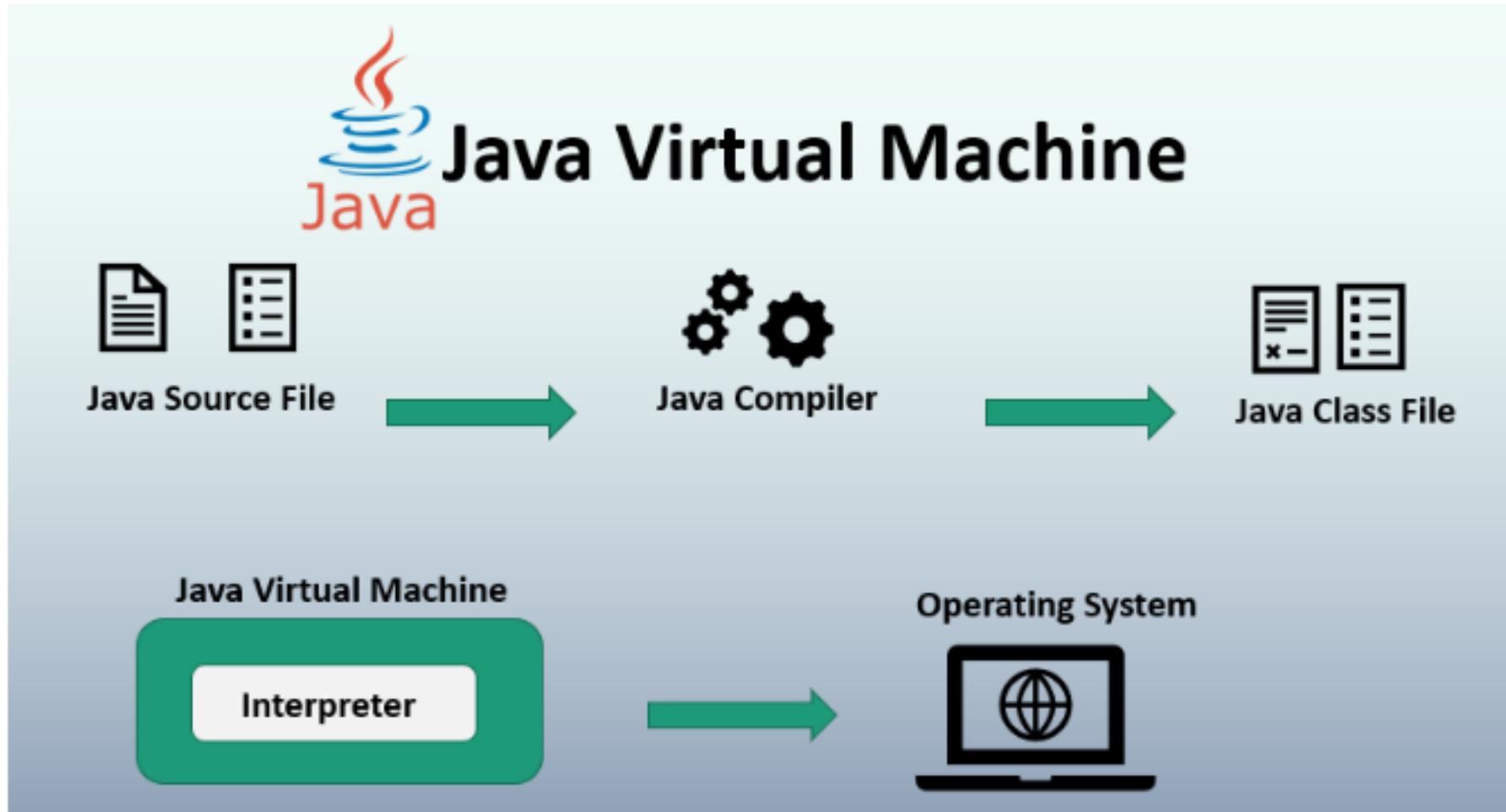
- Otherwise, programs written in VM1 can be interpreted/translated and executed on another virtual machine (VM0) that can execute commands written in L0.

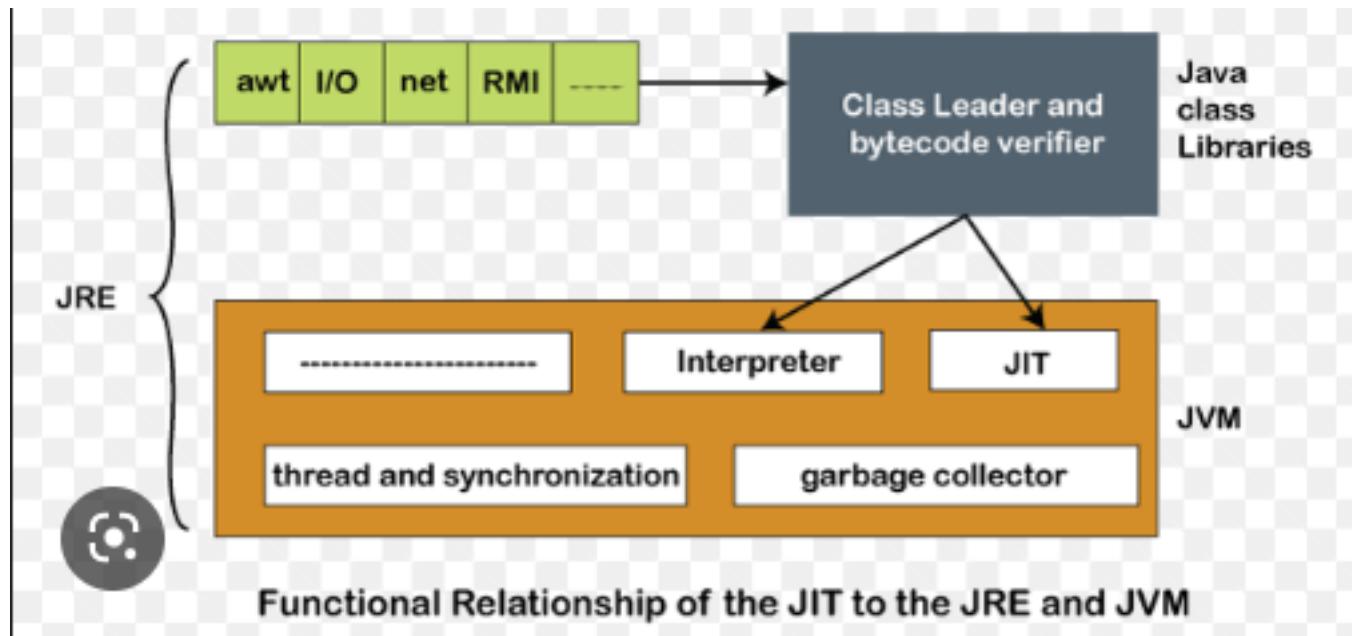


- If the language supported by VM1 is still not easy enough to use, another virtual machine (VM2) can be designed.
- This process can be repeated until a virtual machine (VMn) can be designed to support a powerful,

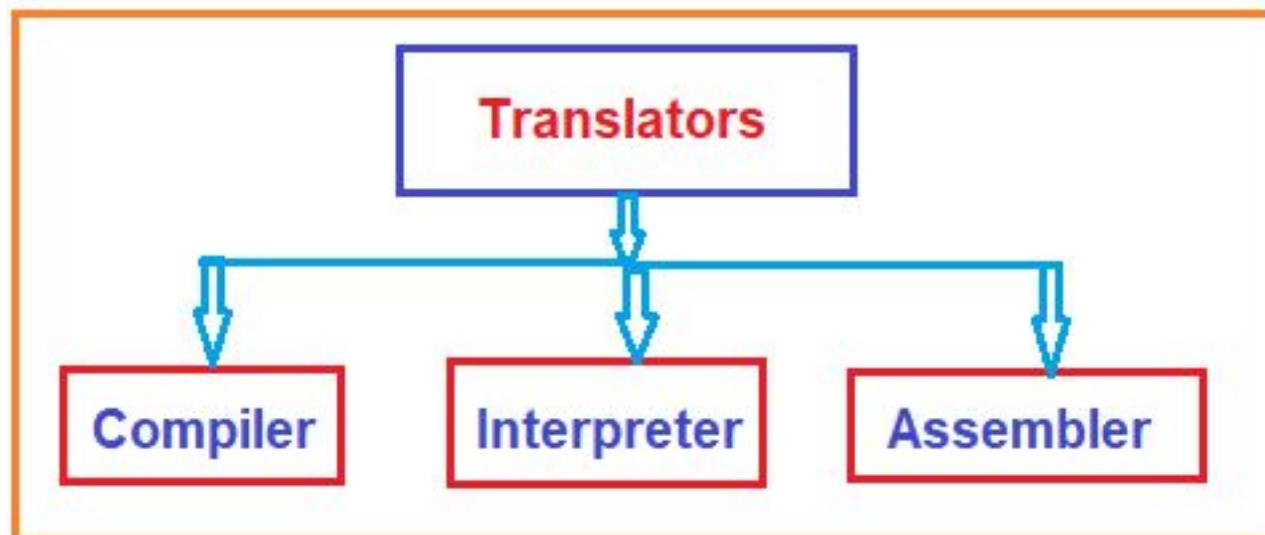
easy-to-use language.

- The Java programming language is based on the virtual machine concept, where programs are translated into Java byte code and executed at runtime by a Java virtual machine (JVM).



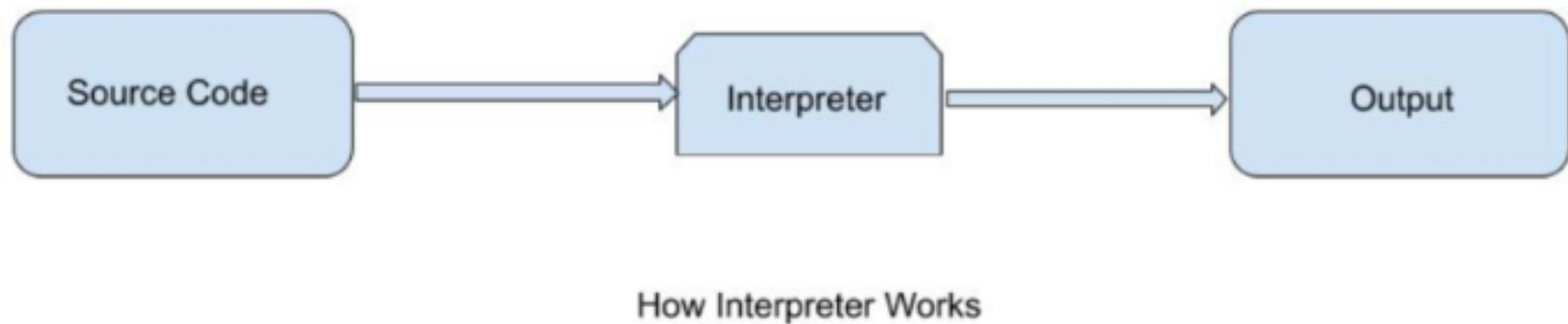


## INTERPRETERS, COMPILERS AND TRANSLATORS



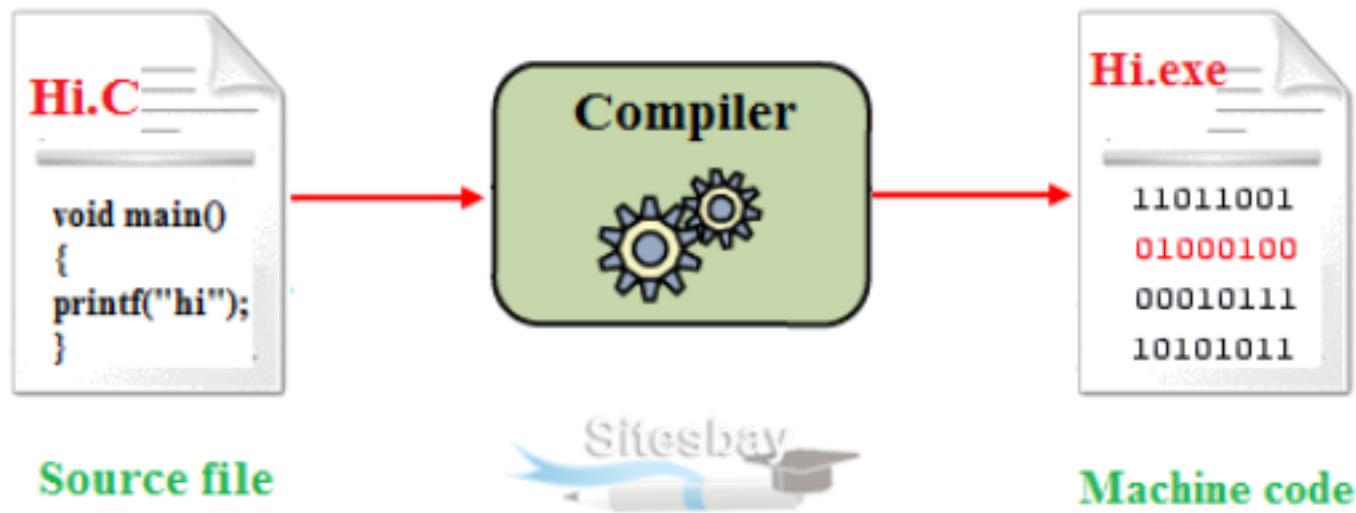
## Interpreters:

- An interpreter is a program that reads high-level code, such as Python or Ruby, and executes it directly.
- Interpreters do not produce executable files, but instead, they run code line-by-line, translating each line into machine code as they go.
- Interpreters are slower than compilers because they execute code one line at a time.
- Interpreters are useful for quick prototyping, debugging, and scripting.



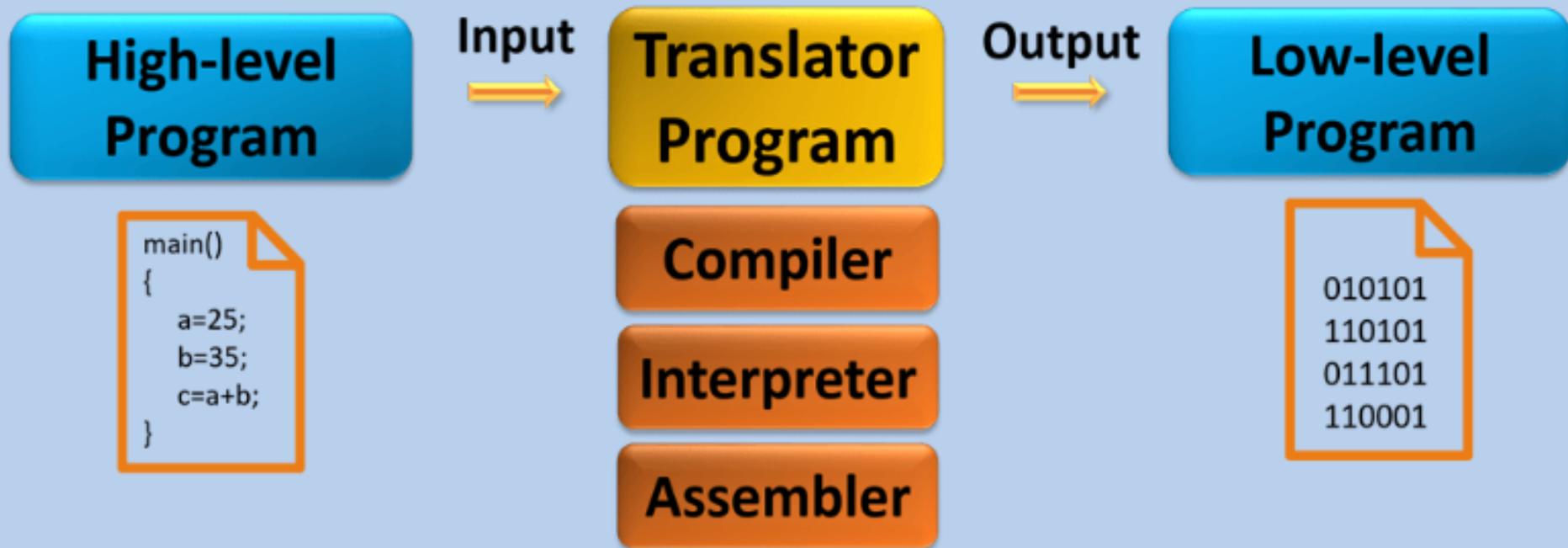
## Compilers:

- A compiler is a program that translates high-level code, such as C++ or Java, into machine code.
- Compilers produce executable files that can be run directly on the computer's hardware.
- Compilers analyze the entire code at once and generate optimized machine code that can run faster than interpreted code.
- Compilers are slower than interpreters during the compilation process, but the compiled code runs much faster.
- Compilers are used for developing large-scale applications and software libraries.



### Translators:

- A translator is a program that converts code from one high-level language to another.
- Translators can be used to port existing software to a new platform or to convert code from an outdated language to a modern one.
- Translators can be either source-to-source or binary-to-binary.
- Source-to-source translators translate code from one high-level language to another high-level language.
- Binary-to-binary translators translate executable files from one machine architecture to another.



A translator is a program that converts code from one high-level language to another. Is it a must that a translator translates high level languages only? Isn't an assembler a translator too? But it translates low level assembly?

You are correct that the term "translator" can refer to programs that convert code from one language to another, regardless of whether the source language is high-level or low-level. In the context of programming languages, however, the term "translator" is often used specifically to refer to programs that convert high-level code to another high-level code, rather than converting low-level code to high-level or vice versa.

An assembler is a program that converts assembly language, which is a low-level language, into

machine code, which is the lowest-level code that can be executed by a computer's processor. Assemblers are not typically referred to as translators, as they do not convert high-level code to another high-level code. Instead, they translate assembly code to machine code.

**So only compilers and interpreters are the only ones that are called translators. an assembler is not in that list?**

The terms "compilers" and "interpreters" are more commonly used to describe programs that translate high-level code, whereas "assemblers" are used to describe programs that translate assembly code to machine code.

However, technically speaking, all of these programs are translators, as they are converting code from one language to another.

So, while it is not as common to refer to an assembler as a translator, it can still be considered a type of translator in the broader sense of the term.

**Why do you suppose translated programs often execute more quickly than interpreted ones?**

Translated programs often execute more quickly than interpreted ones for several reasons:

- 1. Compiled code is translated to machine code**, which is a low-level language that can be executed directly by the computer's processor. **Interpreted code**, on the other hand, is executed by an interpreter, which must first interpret the code and then execute it. This additional step adds overhead and can slow down the execution of the code.



2. The **translation process for compiled code occurs only once**, during the compilation phase. Once the code is compiled, the resulting machine code can be executed repeatedly without the need for further translation. In contrast, interpreted code must be translated each time it is executed, which can slow down the overall performance.

# Translated at once

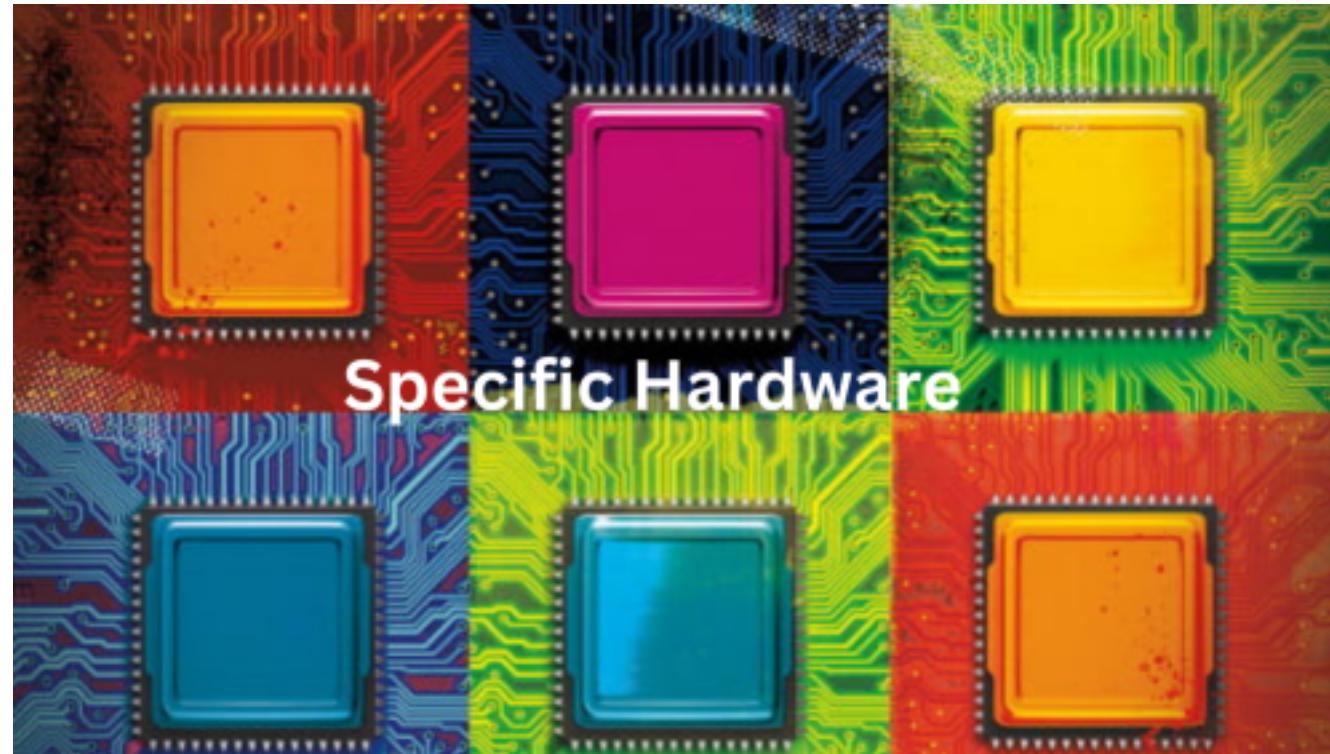


3. **Compiled code can often be optimized by the compiler** to improve its performance. For example, the compiler may be able to reorder instructions or eliminate redundant operations. Interpreted code, on the other hand, is typically not optimized in this way.

# Code Optimization



4. Compiled code can be optimized for the **specific hardware** it will be running on, whereas interpreted code must be interpreted on the fly and cannot be optimized for a specific platform. This can lead to better performance for compiled code on the target hardware.



Overall, the main advantage of compiled code over interpreted code is that it can be executed more efficiently, with less overhead and better optimization. However, interpreted code has the advantage of being more flexible and easier to debug and modify, as there is no need for a separate compilation step.

**(True/False): When an interpreted program written in language L1 runs, each of its instructions is decoded and executed by a program written in language L0?**

In interpretation, each instruction of the program written in language L1 is decoded and executed by a program written in language L0 as the L1 program is running.

The interpreter is a program written in L0 that reads each instruction of the L1 program, decodes it into L0 instructions, and executes those L0 instructions to perform the task specified by the L1

instruction.

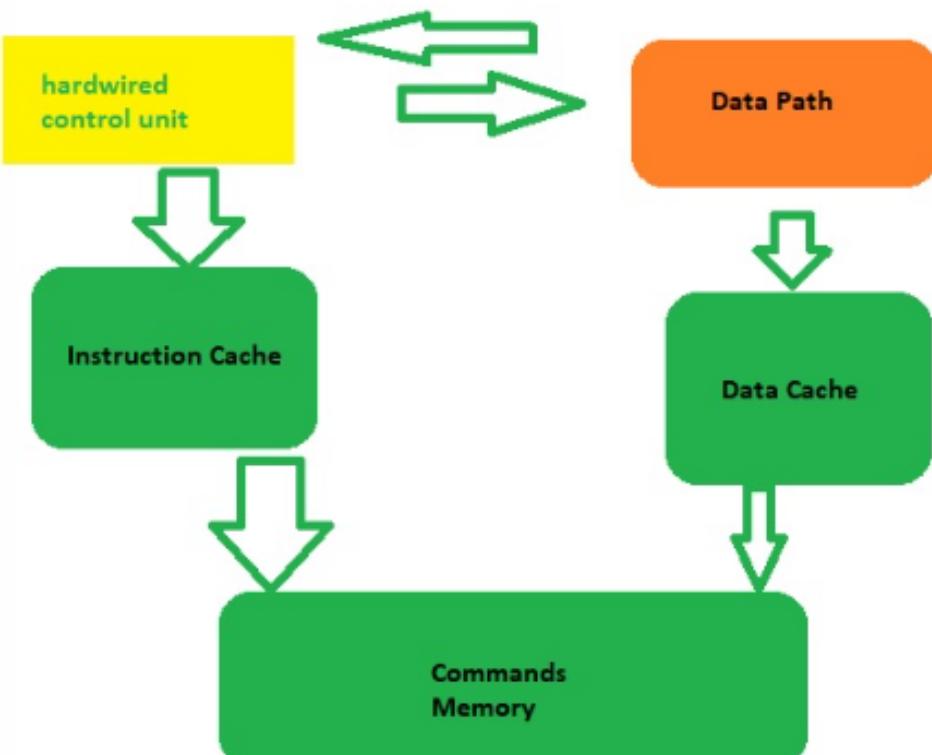
This means that the program written in L1 is not directly executed on the computer hardware; rather, it is executed indirectly through the interpreter program, which translates the L1 instructions into L0 instructions that can be executed by the computer hardware.

**Explain the importance of translation when dealing with languages at different virtual machine levels.**

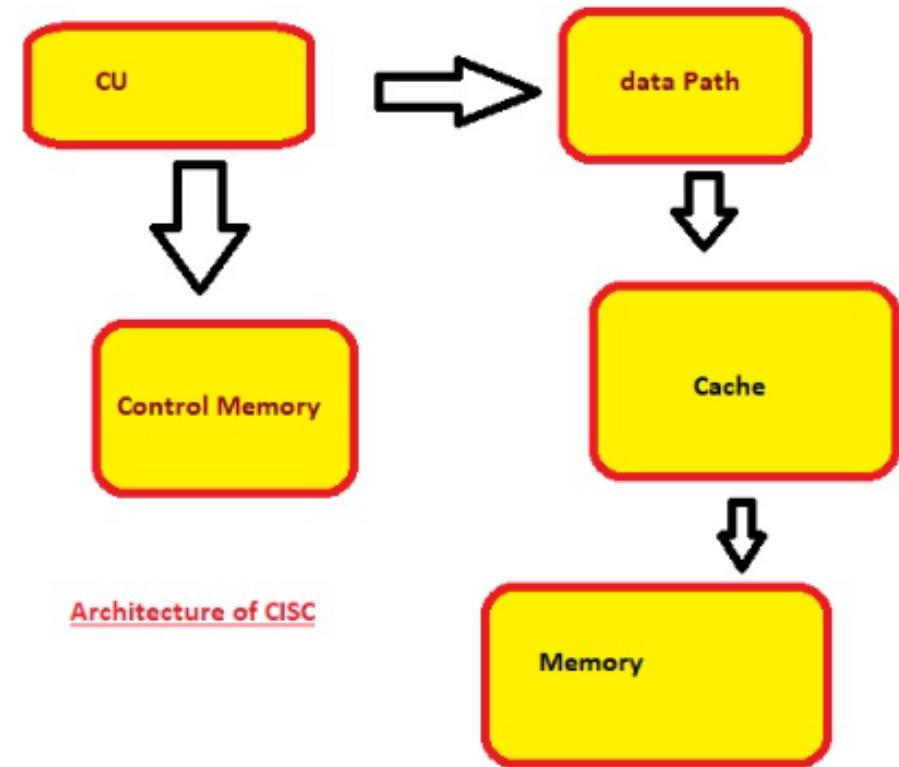
When dealing with languages at different virtual machine levels, translation plays an important role in making it possible for programs written in one language to be executed on a virtual machine designed for a different language. Here are some reasons why translation is important:

1. Different virtual machines have **different instruction sets and formats**, which means that a program written for one virtual machine cannot be directly executed on another virtual machine without modification. Translation allows a program written for one virtual machine to be converted into a format that can be executed on a different virtual machine.

## Difference between RISC and CISC



Architecture of RISC



Architecture of CISC

2. Programs written in a **high-level language** are easier to read, write, and understand than programs written in a low-level language. However, high-level languages are generally less efficient than low-level languages in terms of execution speed and memory usage. Translation allows programs written in a high-level language to be converted into a lower-level language, which can improve their performance.

HIGH LEVEL LANGAUGES ARE

EASY AS

1, 2, 3...

?

3. Translation allows software developers to **write programs in a language that is suitable for their needs**, even if that language is not natively supported by the virtual machine on which the program will be executed. For example, a developer may choose to write a program in Python, even if the target virtual machine only supports C++.



Overall, translation is important because it allows programs written in one language to be executed on a virtual machine designed for a different language. This makes it possible for software developers to use the languages and tools that are best suited for their needs, without being limited by the capabilities of the underlying virtual machine.

### **Does assembly ever appear in the JVM?**

Assembly language may appear after interpretation, depending on the implementation of the Java Virtual Machine (JVM) and the underlying hardware architecture.

When the JVM interprets Java bytecode, it converts each bytecode instruction into a sequence of

native machine instructions that can be executed directly by the hardware.

These machine instructions are typically represented in assembly language, which is a low-level programming language that is specific to a particular hardware architecture.

However, some JVM implementations may use just-in-time (JIT) compilation to dynamically generate optimized native machine code for frequently executed portions of the bytecode.

In this case, the resulting native machine code may not be represented in assembly language, as it is generated and executed at runtime rather than being precompiled.

Overall, whether assembly language appears after interpretation depends on the specific implementation of the JVM and the execution environment.

### **Why don't programmers write applications in machine language?**

Programmers do not typically write applications in machine language because it is a low-level language that is **very difficult to read and understand**, and it is **specific to a particular hardware architecture**. Writing applications in machine language requires a **deep understanding of the underlying hardware** and can be a **time-consuming** and **error-prone** process. Instead, programmers typically use higher-level programming languages, such as Java, Python, or C++, that provide more abstraction from the hardware and are easier to read and write.

### **Statements at the assembly language level of a virtual machine are translated into statements at which other level?**

Statements at the assembly language level of a virtual machine are typically translated into statements at the machine language level of the underlying hardware architecture. Assembly language is a low-level language that is specific to a particular hardware architecture, and it provides a more human-readable representation of machine language instructions. The translation process from

assembly language to machine language is typically performed by an assembler, which is a software tool that converts assembly language code into executable machine code that can be run on the hardware.

## **CHAPTER 1.2 : DATA REPRESENTATION**

Data Representation Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers.

Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used.

You must develop a certain fluency with number formats, so you can quickly translate numbers from one format to another.

Each numbering format, or system, has a base, or maximum number of symbols that can be assigned to a single digit.

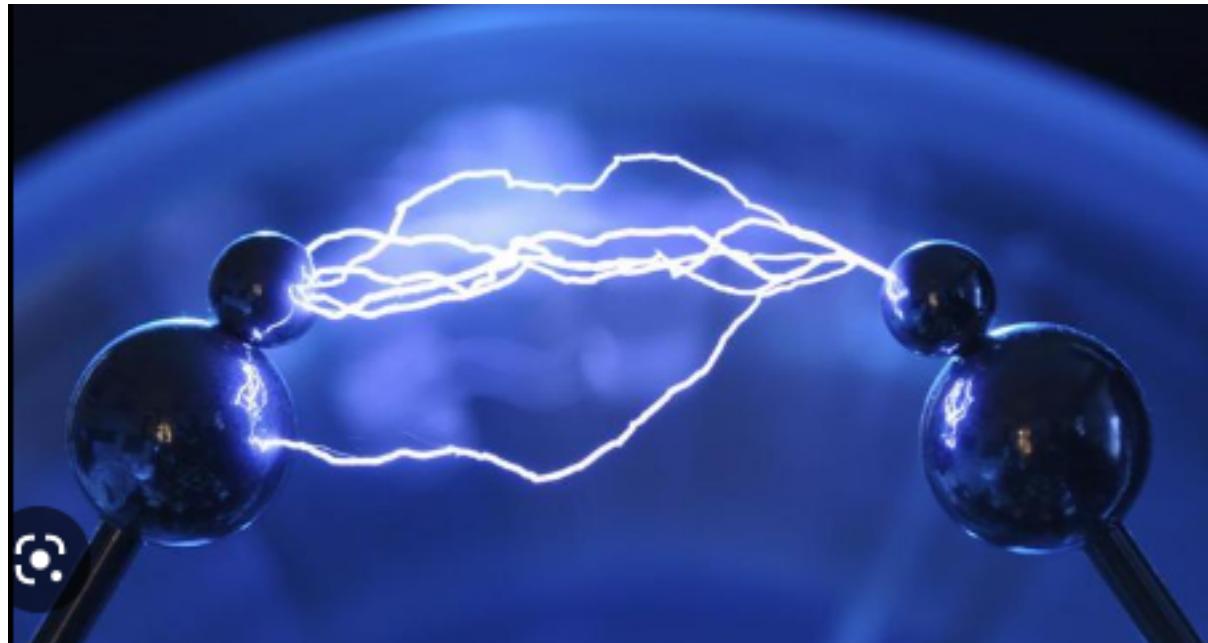
Table 1-2 shows the possible digits for the numbering systems used most commonly in hardware and software manuals. In the last row of the table, hexadecimal numbers use the digits 0 through 9 and continue with the letters A through F to represent decimal values 10 through 15.

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

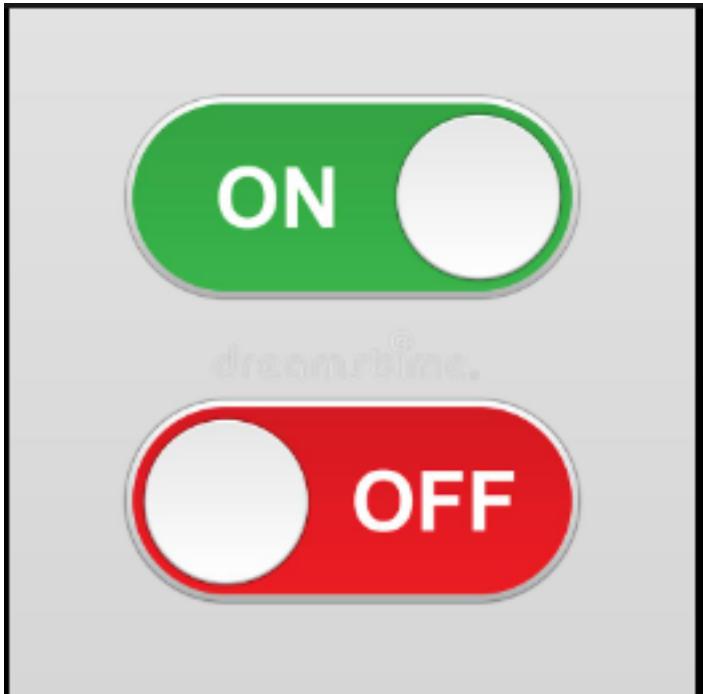
It is quite common to use hexadecimal numbers when showing the contents of computer memory and machine-level instructions.

## BINARY INTEGERS

A computer stores instructions and data in memory as collections of electronic charges.

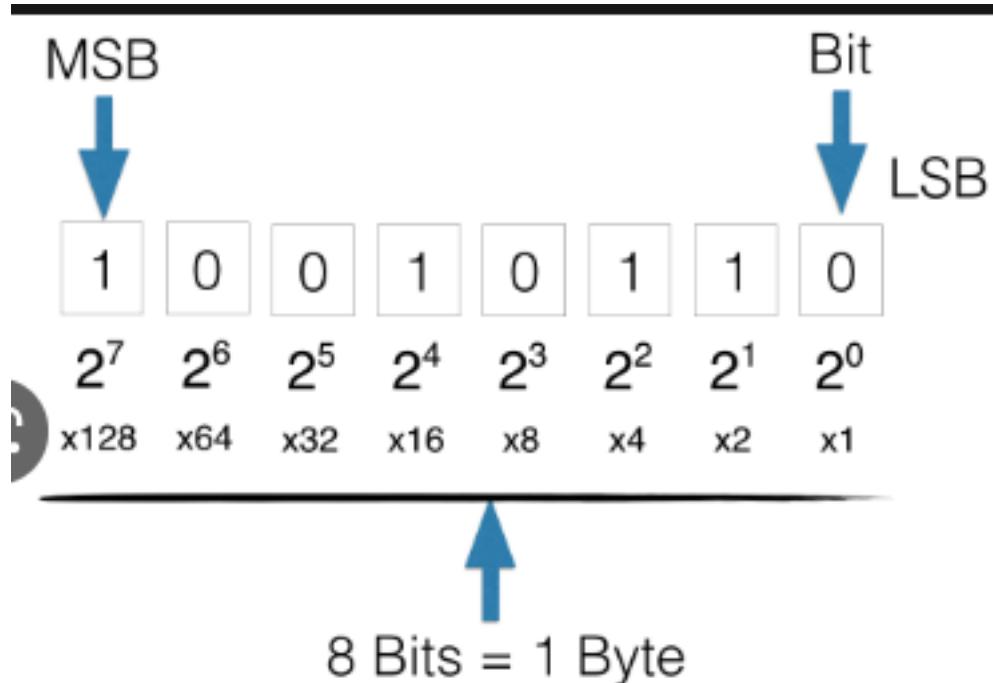


Representing these entities with numbers requires a system geared to the concepts of on and off or true and false.



Binary numbers are base 2 numbers, in which each binary digit (called a bit) is either 0 or 1.

S



Bits are numbered sequentially starting at zero on the right side and increasing toward the left. The bit on the left is called the most significant bit (MSB), and the bit on the right is the least significant bit (LSB). The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure:

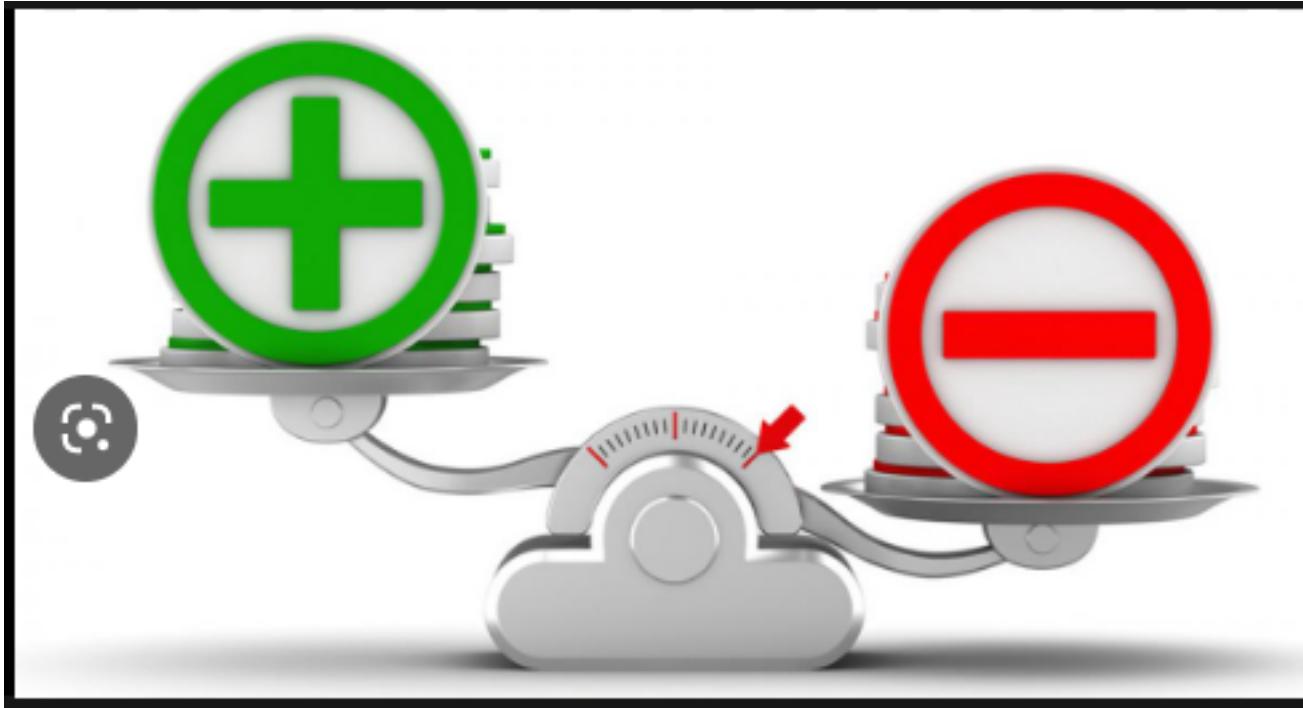


Table 1-2 Binary, Octal, Decimal, and Hexadecimal Digits.

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Binary integers can be signed or unsigned.

A **signed integer** is positive or negative.



An **unsigned integer** is by default positive. Zero is considered positive.



When writing down large binary numbers, many people like to insert a dot every 4 bits or 8 bits to make the numbers easier to read. Examples are 1101.1110.0011.1000.0000 and 11001010.10101100.

Unsigned Binary Integers Starting with the LSB, each bit in an unsigned binary integer represents an increasing power of 2.

The following figure contains an 8-bit binary number, showing how powers of two increase from right to left:

## QUICK QUESTION

We use 3 bits to represent a number like 8. ( $2^3$ )?

Using 3 bits, we can represent  $2^3=8$  different values.

The range of values we can represent with 3 bits is from 000 (0 in decimal) to 111 (7 in decimal).

Here's how we can represent the number 8 in binary using 3 bits:

```
8 (decimal) = 1000 (binary)
```

In binary, each digit can have only two possible values: 0 or 1.

Each digit in a binary number represents a power of 2, starting from the rightmost digit with  $2^0$ , then  $2^1$ ,  $2^2$ , and so on, doubling each time. In the example above, the leftmost digit represents  $2^3$ , or 8, which is why the value is 1 and the other digits are 0.

so, the answer is no. We use 4 bits?

Yes, that's correct. To represent the decimal number 8 in binary, we actually need 4 bits, not 3.

Using 4 bits, we can represent  $2^4=16$  different values.

The range of values we can represent with 4 bits is from 0000 (0 in decimal) to 1111 (15 in decimal).

Here's how we can represent the number 8 in binary using 4 bits:

8 (decimal) = 1000 (binary)

In this case, we have one digit representing  $2^3$ , or 8, and the other three digits are 0. What do I mean?

$$2^3 = 8$$

1	0	0	0
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$2^3$	$2^2$	$2^1$	$2^0$

## BINARY TO DECIMAL

BINARY TO DECIMAL

1 0 1 0 1 0<sub>(2)</sub>

1 0 1 0 1 0  
↓ ↓ ↓ ↓ ↓ ↓  
 $2^5 2^4 2^3 2^2 2^1 2^0$

$32 + 8 + 2 = 42$

## BINARY WITH DECIMAL POINTS

$$\begin{array}{r} 101010.01 \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^{-2} \end{array}$$

$$32 + 8 + 2 + \frac{1}{4} = \frac{42}{1} + \frac{1}{4} \\ = 42\frac{1}{4} / 42.25$$

## DECIMAL TO BINARY

## DECIMAL TO BINARY

$$100_{(10)} \rightarrow x_{(2)}$$

$$\begin{array}{r} 100 \div 2 = 50 \text{ R } 0 \\ 50 \div 2 = 25 \text{ R } 0 \\ 25 \div 2 = 12 \text{ R } 1 \\ 12 \div 2 = 6 \text{ R } 0 \\ 6 \div 2 = 3 \text{ R } 0 \\ 3 \div 2 = 1 \text{ R } 1 \\ 1 \div 2 = 0 \text{ R } 1 \end{array}$$

$= 1100100_{(2)}$

## DECIMAL WITH DECIMAL POINTS TO BINARY

- Convert the fractional part to binary, multiply fractional part with 2 and take the one bit which appears before the decimal.
- Follow the procedure repeatedly until result ends up with 1.0

$$0.25 \times 2 = 0.5 \text{ take } 0$$

$$0.5 \times 2 = 1.0 \text{ take } 1$$

$$0.25_{(2)} = (01)_2$$

$$100.25_{(10)} = 1100100.01_{(2)}$$

## BINARY TO HEX TABLE

## BINARY TO HEXADECIMAL

HexaDecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

HOW TO DRAW THE TABLE AT ONCE

Drawing the table at once:

8 zeros going down

8 ones going down



4 zeros going down

in column 2.

4 ones going down

repeats



2 zeros, 2 ones



Zero, one, zero, one, zero

## BINARY TO OCTAL TABLE

Octal	Binary
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

The table is drawn just like the hex table above.

## HEX TO DECIMAL

## HEX TO DECIMAL

7 C F<sub>(16)</sub>

You can convert to binary,  
using the table if you have  
no idea of method 2.

7 C F

$$\begin{array}{r} \downarrow \quad \downarrow \quad \downarrow \\ 16^2 \quad 16^1 \quad 16^0 \end{array} \rightarrow (7 \times 256) + (12 \times 16) + (15 \times 1) \\ \rightarrow 1999_{(10)}$$

## DECIMAL TO HEX

DECIMAL TO HEX

960<sub>(10)</sub>

$$\begin{array}{r} 960 \div 16 = 60 \text{ R } 0 \\ 60 \div 16 = 3 \text{ R } 12 \\ 3 \div 16 = 0 \text{ R } 3 \end{array}$$



300<sub>(16)</sub>

HEX TO OCTAL

HEX TO OCTAL

1BC<sub>(16)</sub>

Convert to binary  
using table.

000110110  
| B C

Subdivide into 3's  
from right to left

000 - 0      ~~110~~  
110 - 6      ~~111~~ right  
111 - 7      ~~100~~ to  
100 - 4      left

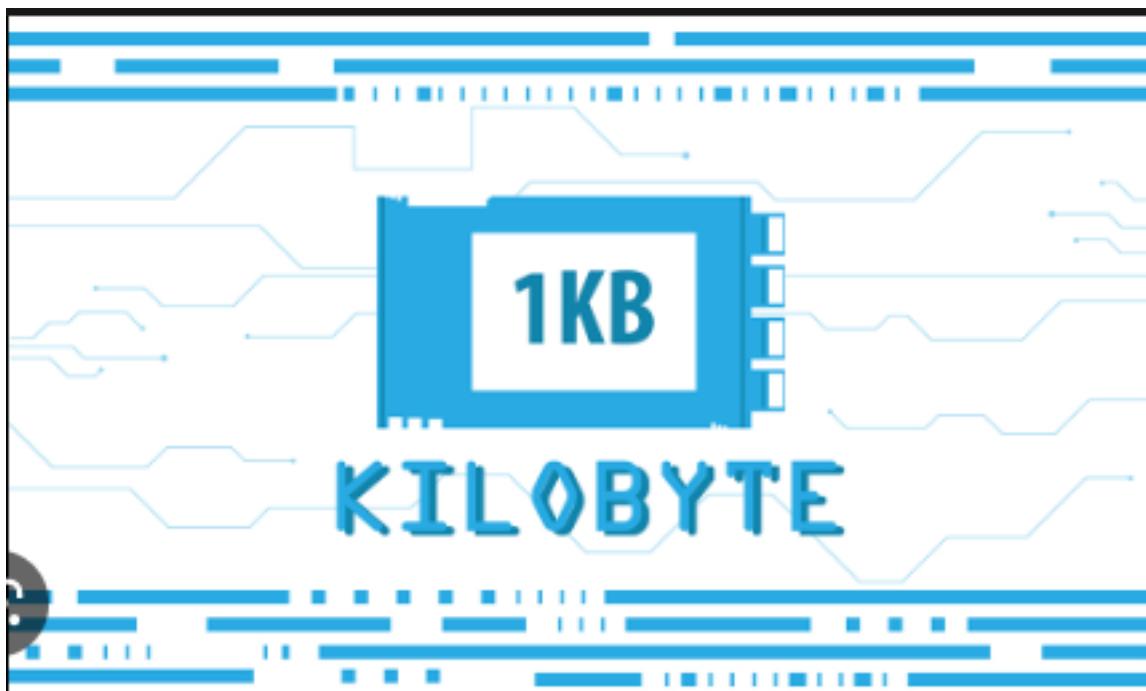
= 674<sub>(8)</sub>

Zero doesn't alter  
the value, so ignore

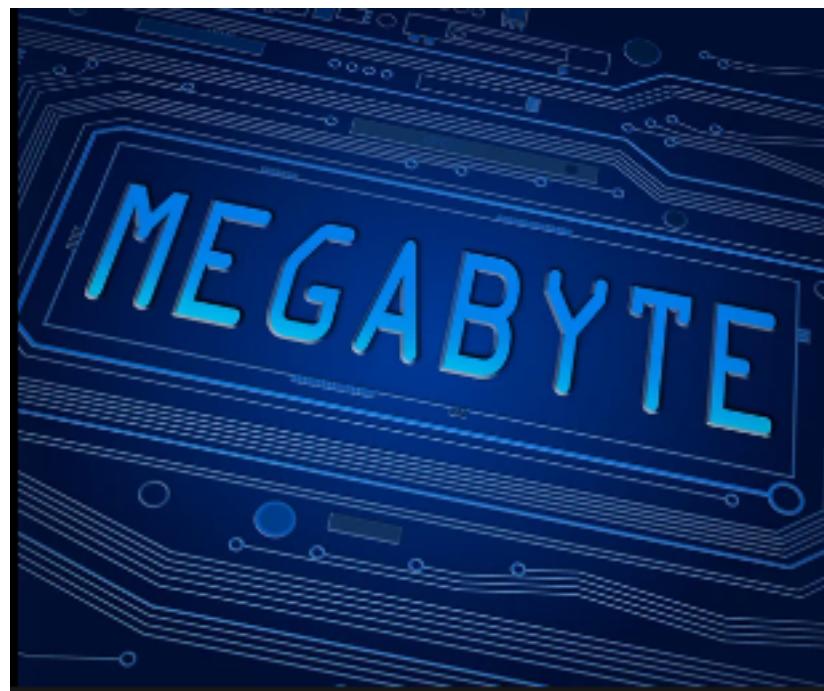
## MEASUREMENTS

Yes, there are several large measurements used when referring to both memory and disk space. Here are some of the most common ones:

Kilobyte (KB): 1,000 bytes



Megabyte (MB): 1,000 kilobytes or 1,000,000 bytes



Gigabyte (GB): 1,000 megabytes or 1,000,000,000 bytes



Terabyte (TB): 1,000 gigabytes or 1,000,000,000,000 bytes



Petabyte (PB): 1,000 terabytes or 1,000,000,000,000,000 bytes

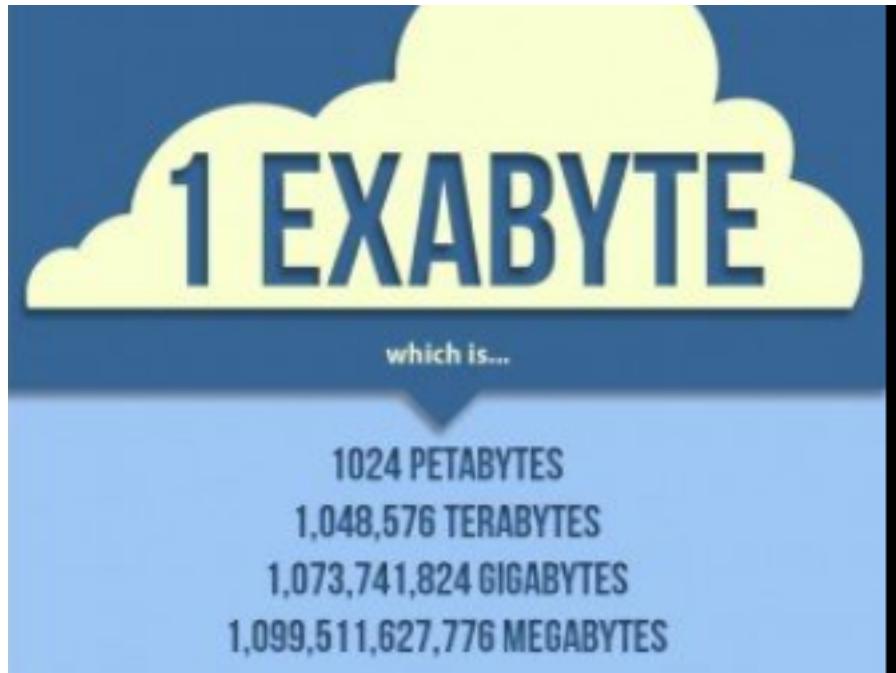
1.6 PETABYTES OF DATA IS

10 BILLION FACEBOOK PHOTOS

750 BILLION PAGES OF STANDARD TYPE TEXT

SEAGATE

Exabyte (EB): 1,000 petabytes or 1,000,000,000,000,000,000 bytes



Zettabyte (ZB): 1,000 exabytes or 1,000,000,000,000,000,000 bytes



Yottabyte (YB): 1,000 zettabytes or 1,000,000,000,000,000,000,000,000 bytes

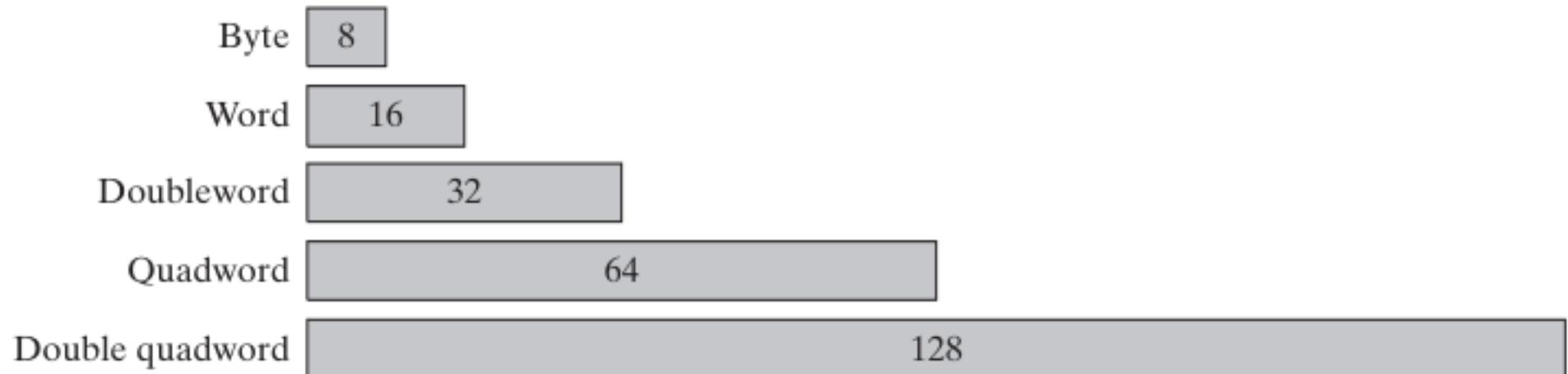


It's important to note that these measurements are based on the decimal system, which means that 1 kilobyte is 1,000 bytes, not 1,024 bytes as it would be in the binary system. Therefore, when discussing computer memory and storage, you may also encounter measurements based on the binary system, such as kibibyte (KiB), mebibyte (MiB), gibibyte (GiB), etc. These measurements are based on powers of 2 and are often used in technical contexts to avoid confusion.

## INTEGER STORAGE

The basic storage unit for all data in an x86 computer is a byte, containing 8 bits.

Other storage sizes are word (2 bytes), doubleword (4 bytes), and quadword (8 bytes).



Data Type	Size (bits)	Size (bytes)	Minimum Value	Maximum Value
Unsigned Byte	8	1	0	255
Unsigned Word	16	2	0	65,535
Unsigned Doubleword	32	4	0	4,294,967,295
Unsigned Quadword	64	8	0	18,446,744,073,709,551,615
Unsigned Double Quadword	128	16	0	340,282,366,920,938,463,463,374,607,431,768,211,455

Note that the minimum value for all of these data types is 0, since they are unsigned (i.e., they cannot represent negative values).

The maximum value increases as the size of the data type increases.

It's also important to note that the sizes and maximum values listed in this table are for unsigned integers in their raw binary form.

In programming languages, these values may be represented differently, depending on the data type and the language itself.

## SIGN AND ZERO EXTENSION

In computing, two's complement format integers have a fixed length, which can create issues when converting values between different lengths. For example, if you need to convert an 8-bit two's complement value to a 16-bit value, or vice versa.

To solve this problem, you can use sign extension or contraction operations. Sign extension replicates the most significant (i.e., leftmost) bit of the original value to fill the additional bits in the longer value.

This ensures that the value retains its original sign (positive or negative) when it is extended. On the other hand, zero extension adds additional 0 bits to the right of the original value to fill the additional bits in the longer value. This is used for unsigned values because there is no sign bit to replicate.

**Main point:** In the case of the 80x86 architecture, the system works with fixed-length values, even when processing unsigned binary numbers.

This means that:

1. zero extension is used to convert small unsigned values to larger unsigned values, while
2. sign extension is used to convert signed values, replicating the sign bit.

It's worth noting that some operations, such as multiplication and division, require conversion to

32-bit values, which may involve sign or zero extension depending on the original value's sign and whether it is being extended or contracted.

## TWO'S COMPLEMENT REPRESENTATION

Two's complement is a binary number representation that allows for the representation of both positive and negative integers using only 0's and 1's.

It is the most commonly used representation for signed integers in modern computing systems.

To represent a positive integer in two's complement, you simply represent it in binary form as usual. For example, to represent the number 5, you would write: 00000101.

To represent a negative integer in two's complement, you first take the binary representation of the absolute value of the number.

Then you invert all the bits (change 0's to 1's and 1's to 0's), and finally add 1 to the result.

For example, to represent the number -5, you would start with the binary representation of 5 (00000101), invert all the bits (11111010), and then add 1, resulting in the two's complement representation: 11111011.

## TWO'S COMPLEMENT

- 5 ( 00000101 )
  - Getting two's complement  
ie -5
  - 1's become zeros, 0's becomes 1's.

11111010

- Add 1 :

$$\begin{array}{r} 11111010 \\ + 00000001 \\ \hline 11111011 \end{array}$$

Remember that 1 in the high order bit means negative.

The key advantage of two's complement is that it simplifies the implementation of arithmetic operations, such as addition and subtraction, because the same hardware can be used for both positive and negative numbers.

In addition, it eliminates the need for a separate sign bit, which would require extra operations to check and manipulate.

## **NOTE:**

To find the negative number's binary eg negative (-6), you take its positive(+6), convert to binary(0110), invert the sign bit (1110) and that's it. Remember that 1 means negative.

To find the 1's complement you just invert the bits. To find the 2's complement, you invert, then add the binary of 1 to the result.

To subtract binary numbers, its straightforward, but you can also convert them to decimal, subtract, then go back to binary.

To subtract 39 and 25, just do it, then convert to binary.

## **CALCULATING FOR 39, -25**

1. Convert the absolute values of both numbers to binary using the 64, 32, 16, 8, 4, 2, 1 method.

39 in binary: 00100111

25 in binary: 00011001

2. Take the two's complement of -25 by inverting all the bits and adding 1:

25 in binary: 00011001

Inverted: 11100110

Add 1: 11100111

So, -25 in two's complement form is 11100111.

3. Make sure both binary numbers have the same number of bits (7 in this case).

**00100111** (39)

**11100111** (-25)

4. Add the two binary numbers:

**00100111**

**11100111**

The result in binary is **00001110**.

5. Convert the binary result to decimal:

**00001110** in decimal is 14.

So,  $39 - (-25) = 39 + 25 = 64$ , and the binary representation of 64 is **01000000**.

## TWO'S COMPLEMENT OF A HEXADECIMAL NUMBER

To create the two's complement of a hexadecimal integer in the hexadecimal number system, you would follow these steps:

1. Convert the hexadecimal integer to binary.
2. Reverse all the bits (change 0's to 1's and 1's to 0's).
3. Add 1 to the result.

An easy way to reverse the bits of a hexadecimal digit is to subtract the digit from F (15 in decimal). Example 1:

1. Convert 5A to binary:

0101 1010

2. Reverse all the bits:

1010 0101

3. Add 1 to the result:

$1010\ 0101 + 0000\ 0001 = 1010\ 0110$

So, the two's complement of 5A in hexadecimal is A6.

Here is another example to find the two's complement of the hexadecimal integer B3:

1. Convert B3 to binary:

1011 0011

2. Reverse all the bits:

0100 1100

3. Add 1 to the result:

$0100\ 1100 + 0000\ 0001 = 0100\ 1101$

So, the two's complement of B3 in hexadecimal is 4D.

## SIGNED BINARY TO DECIMAL

Starting value	11110000
Step 1: Reverse the bits	00001111
Step 2: Add 1 to the value from Step 1	$  \begin{array}{r}  00001111 \\  + \quad \quad 1 \\  \hline  \end{array}  $
Step 3: Create the two's complement	00010000
Step 4: Convert to decimal	16

## CONVERSION OF -43 SIGNED NUMBER TO BINARY

Here are the steps to convert -43 to binary:

1. Convert 43 to binary: 101011
2. Invert all the bits: 010100
3. Add 1 to the result: 010101

Therefore, the binary representation of -43 in 8-bit two's complement notation is: 10101011. Note that the leftmost bit (1) indicates that this is a negative number.

## MAXIMUM AND MINIMUM VALUES FOR SIGNED INTEGERS

Type	Range	Storage Size in Bits
Signed byte	$-2^7$ to $+2^7 - 1$	8
Signed word	$-2^{15}$ to $+2^{15} - 1$	16
Signed doubleword	$-2^{31}$ to $+2^{31} - 1$	32
Signed quadword	$-2^{63}$ to $+2^{63} - 1$	64
Signed double quadword	$-2^{127}$ to $+2^{127} - 1$	128

## CHARACTER SET - ASCII

Computers represent characters using character sets, which map each character to a unique integer. ASCII is a commonly used character set that assigns a unique 7-bit integer to each character.

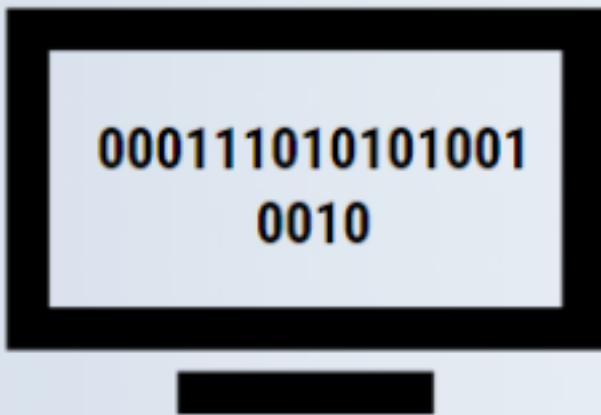
The first 128 characters of the 8-bit ANSI character set correspond to letters and symbols on a standard US keyboard, while the second 128 characters represent special characters from international alphabets, accents, currency symbols, and fractions.

Unicode is a universal standard for defining characters and symbols, which defines numeric codes for characters used in all major languages and scripts. Unicode has three transformation formats: UTF-8, UTF-16, and UTF-32.

Strings of characters are stored in memory as a sequence of bytes containing the corresponding character codes, and null-terminated strings are strings followed by a single byte containing zero.

## UTF-8, UTF-16, UTF-32

In computers, **encoding** is the process of putting a sequence of characters (letters, numbers, punctuation, and certain symbols) into a specialized format for efficient transmission or storage. **Decoding** is the opposite process - the conversion of an encoded format back into the original sequence of characters.



**Encoding**

**VS**



**Decoding**

UTF-8, UTF-16, and UTF-32 are three different encoding formats used to represent Unicode characters.

UTF-8                    UTF-16  
                          UCS-2

# UNICODE

ENCODINGS

UTF-32

- **UTF-8:** It is a variable-length encoding format that uses 8-bit code units to represent characters. In UTF-8, the first 128 characters (code points) are represented using a single byte. The remaining characters are represented using two, three, or four bytes, depending on their Unicode code point value. UTF-8 is the most widely used encoding format on the web and in email systems because it is backwards compatible with ASCII and can represent any Unicode character.
- **UTF-16:** It is a variable-length encoding format that uses 16-bit code units to represent characters. In UTF-16, the first 65,536 characters are represented using a single 16-bit code unit. The remaining characters are represented using two 16-bit code units (surrogate pairs). UTF-16 is used in some operating systems and programming languages, such as Microsoft Windows and Java.
- **UTF-32:** It is a fixed-length encoding format that uses 32-bit code units to represent characters. In UTF-32, each character is represented using a single 32-bit code unit. UTF-32 is less commonly

used than UTF-8 and UTF-16 because it requires more storage space than the other two formats. It is used in some programming languages, such as C and C++.

## ASCII STRINGS

ASCII strings are sequences of characters that are represented in memory as a succession of bytes containing ASCII codes. ASCII codes are numeric values that are assigned to each character in the ASCII character set. For example, the ASCII code for the capital letter "A" is 65, and the code for the digit "0" is 48.

To store a string of ASCII characters in memory, each character is represented by its corresponding ASCII code. For example, the string "ABC123" is stored in memory as the hexadecimal values 41h, 42h, 43h, 31h, 32h, and 33h, which correspond to the ASCII codes for the characters 'A', 'B', 'C', '1', '2', and '3', respectively.

A null-terminated string is a string of characters followed by a single byte containing zero. This zero byte is called a null terminator and is used to mark the end of the string. The C and C++ programming languages use null-terminated strings as the standard way to represent strings in memory. Many Windows operating system functions also require strings to be in this format.

When working with ASCII strings, it is important to keep in mind that each character requires one byte of storage, regardless of its actual size. This can be a limitation when working with multibyte character sets such as Unicode, where some characters require more than one byte of storage.

**Does 41h, 42h, represent the hex value of the memory location of that value 'a' and 'b' or it represents the hex equivalent of 'a' and 'b'??**

The hexadecimal values 41h, 42h, and 43h represent the ASCII codes for the characters 'A', 'B', and 'C', respectively. These values are not the memory locations of the characters in memory, but rather the numeric values that represent these characters in the ASCII character set.

the numeric values that represent these characters in the ASCII character set.

When a string of characters is stored in memory, each character is represented by its corresponding ASCII code. In the example given, the string "ABC123" is stored in memory as the hexadecimal values 41h, 42h, 43h, 31h, 32h, and 33h. These values represent the ASCII codes for the characters 'A', 'B', 'C', '1', '2', and '3', respectively.

To retrieve the characters from memory, the program must interpret the bytes as ASCII codes and convert them to their corresponding characters. In the case of the string "ABC123", the program would read the bytes from memory and interpret them as the characters 'A', 'B', 'C', '1', '2', and '3', in that order.

## USING ASCII TABLE

Using the ASCII Table A table on the inside back cover of this book lists ASCII codes used when running in Windows Console mode.

To find the hexadecimal ASCII code of a character, look along the top row of the table and find the column containing the character you want to translate.

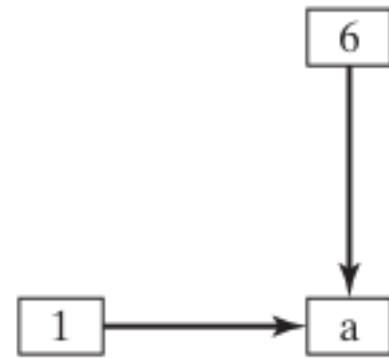
The most significant digit of the hexadecimal value is in the second row at the top of the table; the least significant digit is in the second column from the left.

For example, to find the ASCII code of the letter a, find the column containing the a and look in the second row.

The first hexadecimal digit is 6. Next, look to the left along the row containing a and note that the second column contains the digit 1.

Therefore, the ASCII code of a is 61 hexadecimal.

This is shown as follows in simplified form:



## ASCII CONTROL CHARACTERS

**ASCII control characters** are a set of **33 non-printable characters** with ASCII codes in the range 0 through 31. These characters are often referred to as "control codes" because they are used to control various functions of the computer, such as printing, cursor movement, and data transmission.

When a program writes these control characters to standard output (or any other output device), they will carry out predefined actions instead of being displayed as visible characters.

For example, the ASCII code 10h (hexadecimal) represents the **newline character**, which causes the cursor to move to the beginning of the next line.

Similarly, the ASCII code 08h (hexadecimal) represents the **backspace character**, which moves the cursor back one position and allows overwriting of the previous character.

In C++, you can output ASCII control characters to standard output using escape sequences, such as '\n' for newline and '\b' for backspace.

ASCII table at the start of this book.

## TERMINOLOGY IN DATA REPRESENTATION

It is important to use precise terminology when describing the way numbers and characters are represented in memory and on the display screen. Decimal 65, for example:

- Is stored in memory as a single binary byte as 01000001.
- A debugging program would probably display the byte as “41,” which is the number’s hexadecimal representation.
- If the byte were copied to video memory, the letter “A” would appear on the screen because 01000001 is the ASCII code for the letter A.

Because a number’s interpretation can depend on the context in which it appears, we assign a specific name to each type of data representation to clarify future discussions:

- A **binary integer** is an integer stored in memory in its raw format, ready to be used in a calculation. Binary integers are stored in multiples of 8 bits (such as 8, 16, 32, or 64).
- A **digit string** is a sequence of characters that represent a number. In the case of your example, "123" is a digit string representing the number one hundred twenty-three.

The table you mentioned is likely a reference to the various formats that a number can be represented in. For example, the decimal number 65 can also be represented in binary as "1000001" and in hexadecimal as "41". Here are the formats for the number 65:

- Binary digit string: "01000001".
- Decimal digit string: "65".
- Hexadecimal digit string: "41".
- Octal digit string: "101".

## SHIFTS AND ROTATES

Shifts and rotates are operations performed on binary values. These operations are commonly used in computer programming and digital circuit design to manipulate binary data.

A shift operation involves moving the bits of a binary value left or right by a certain number of positions. A left shift (`<<`) moves the bits to the left, and a right shift (`>>`) moves the bits to the right. For example, shifting the binary value 1010 (decimal 10) one position to the left results in the binary value 10100 (decimal 20).

A rotate operation is similar to a shift operation, but the bits that are shifted out of one end are moved to the other end. There are two types of rotate operations: a left rotate (ROL) and a right rotate (ROR). For example, rotating the binary value 1010 (decimal 10) one position to the left results in the binary value 0101 (decimal 5), with the bit that was shifted out at the left end being moved to the right end.

Shift and rotate operations can be useful for manipulating data in certain ways. For example, shifting a binary value to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Rotates can be useful for circular buffer operations or for

certain encryption algorithms.

More on this can be found online.

## ***CHAPTER 1.3: BOOLEANS***

**Boolean algebra** defines a set of operations on the values true and false.

Won't be covered here. It's a super wide topic that needs a book of its own.



## **EXAM FOR CHAPTER 1**

Use any high-level programming language you wish for the following programming exercises. Do not call built-in library functions that accomplish these tasks automatically. (Examples are `sprintf` and `sscanf` from the Standard C library.)

1. Write a function that receives a string containing a 16-bit binary integer. The function must return the string's integer value.
2. Write a function that receives a string containing a 32-bit hexadecimal integer. The function must return the string's integer value.
3. Write a function that receives an integer. The function must return a string containing the binary representation of the integer.
4. Write a function that receives an integer. The function must return a string containing the hexadecimal representation of the integer.

5. Write a function that adds two digit strings in base b, where  $2 \leq b \leq 10$ . Each string may contain as many as 1,000 digits. Return the sum in a string that uses the same number base.
6. Write a function that adds two hexadecimal strings, each as long as 1,000 digits. Return a hexadecimal string that represents the sum of the inputs.
7. Write a function that multiplies a single hexadecimal digit by a hexadecimal digit string as long as 1,000 digits. Return a hexadecimal string that represents the product.
8. Write a Java program that contains the calculation shown below. Then, use the javap -C command to disassemble your code. Add comments to each line that provide your best guess as to its purpose.

```
int Y;
int temp = Y + 4;
int X = temp * 3;
```

9. Devise a way of subtracting unsigned binary integers. Test your technique by subtracting binary 00000101 from binary 10001000, producing 10000011. Test your technique with at least two other sets of integers, in which a smaller value is always subtracted from a larger one.

## ***EXAM ANSWERS***

1.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 unsigned int binaryToDecimal(char* binary) {
5     int len = strlen(binary);
6     unsigned int result = 0;
7     int i;
8     for (i = 0; i < len; i++) {
9         if (binary[i] == '1') {
10             result = (result << 1) + 1;
11         } else if (binary[i] == '0') {
12             result = result << 1;
13         } else {
14             printf("Error: input string contains non-binary characters\n");
15             return 0;
16         }
17     }
18     return result;
19 }
20
```

```
21 int main() {
22     char binary[] = "1100110011001100";
23     unsigned int decimal = binaryToDecimal(binary);
24     printf("Binary: %s\nDecimal: %u\n", binary, decimal);
25     return 0;
26 }
```

The `binaryToDecimal` function takes a pointer to a string containing a binary integer and returns its decimal value as an unsigned integer. It works by iterating over the characters of the string from left to right, shifting the result to the left by one bit for each '0' encountered and adding 1 before shifting for each '1' encountered. If the string contains a character other than '0' or '1', the function prints an error message and returns 0. In the `main` function, we define a sample binary string, call `binaryToDecimal` to convert it to decimal, and print both values to the console.

2.

```
33 #include <stdio.h>
34 #include <string.h>
35 #include <ctype.h>
36
37 int hexToInt(char *hexStr) {
38     int result = 0;
39     int len = strlen(hexStr);
40     int power = 1;
41
42     // Iterate over the string in reverse order
43     for (int i = len - 1; i >= 0; i--) {
44         char c = toupper(hexStr[i]);
45         int digit = 0;
46
47         // Convert the hex digit to an integer
48         if (isdigit(c)) {
49             digit = c - '0';
50         } else if (c >= 'A' && c <= 'F') {
51             digit = c - 'A' + 10;
52         } else {
53             // Invalid character in the string
54             return -1;
55         }
56     }
57 }
```

```
57         // Add the digit to the result
58         result += digit * power;
59         power *= 16;
60     }
61
62     return result;
63 }
64 }
```

This function takes a string `hexStr` containing a 32-bit hexadecimal integer, and returns its integer value. It first calculates the length of the string, and then iterates over the string in reverse order. For each hex digit in the string, it converts it to an integer value (0-15) and adds it to the result, multiplying by the appropriate power of 16 based on the digit's position in the string. Note that this implementation assumes that the input string contains only valid hexadecimal digits (0-9, A-F, a-f). If the input string contains any other characters, the function returns -1 to indicate an error.

3.

```
75 #include <stdio.h>
76 #include <string.h>
77
78 // Returns the binary representation of a number as a string
79 char *int_to_bin(int num) {
80     static char bin[33]; // buffer to store binary string
81     int i;
82
83     // Convert number to binary string, bit by bit
84     for (i = 31; i >= 0; i--) {
85         bin[31-i] = ((num >> i) & 1) ? '1' : '0';
86     }
87
88     // Add null terminator to end of string
89     bin[32] = '\0';
90
91     return bin;
92 }
93
94 // Example usage
95 int main() {
96     int num = 42;
97     char *bin_str = int_to_bin(num);
98     printf("Binary representation of %d is %s\n", num, bin_str);
99     return 0;
100 }
```

In this implementation, the `int_to_bin` function takes an integer `num` and returns a pointer to a static buffer `bin` containing the binary representation of `num`. The binary string is constructed by iterating over the bits of the integer, starting with the most significant bit, and adding the appropriate character ('0' or '1') to the string based on the value of the bit. Finally, a null terminator is added to the end of the string. The static buffer is used to avoid having to dynamically allocate memory for the string, but it means that the function is not reentrant (i.e., can't be called safely from multiple threads at the same time).

4.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* intToHex(int num) {
    // calculate maximum length of the hexadecimal string
    int length = snprintf(NULL, 0, "%X", num);
    // allocate memory for the string
    char* hexStr = (char*) malloc(length + 1);
    if (hexStr == NULL) {
        printf("Memory allocation failed.");
        exit(1);
    }
    // convert integer to hexadecimal string
    snprintf(hexStr, length + 1, "%X", num);
    return hexStr;
}

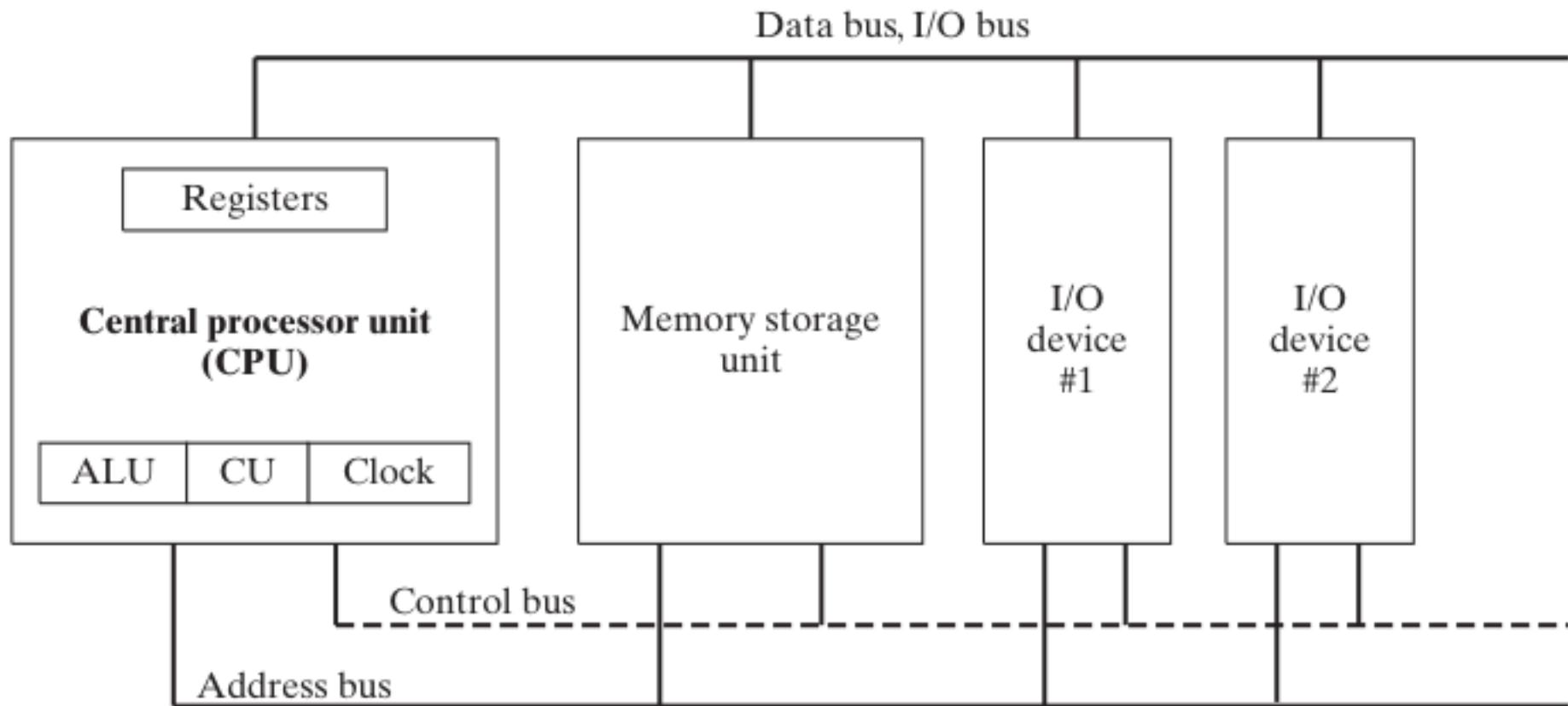
int main() {
    int num = 305441741; // example integer
    char* hexStr = intToHex(num);
    printf("%s\n", hexStr);
    free(hexStr); // free memory allocated for the string
    return 0;
}
```

This function first calculates the maximum length of the hexadecimal string using `snprintf()` and

then allocates memory for the string using `malloc()`. It then converts the integer to a hexadecimal string using `snprintf()` again and returns the string. Finally, the memory allocated for the string is freed using `free()`.

## ***x86 PROCESSOR***

Figure below shows the basic design of a hypothetical microcomputer.



The **central processor unit (CPU)**, where calculations and logical operations take place, contains a limited number of storage locations named registers, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The **clock** synchronizes the internal operations of the CPU with other system components.
- The control unit (CU) coordinates the sequencing of steps involved in executing machine instructions.
- The arithmetic logic unit (ALU) performs arithmetic operations such as addition and subtraction and logical operations such as AND, OR, and NOT.

The CPU is attached to the rest of the computer via pins attached to the CPU socket in the computer's motherboard. Most pins connect to the data bus, the control bus, and the address bus.

The **memory storage unit** is where instructions and data are held while a computer program is running.

The **storage unit** receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory.

All **processing of data** takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute.

Individual program instructions can be copied into the CPU one at a time, or groups of instructions can be copied together.

A **bus** is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address.

The **data bus** transfers instructions and data between the CPU and memory.

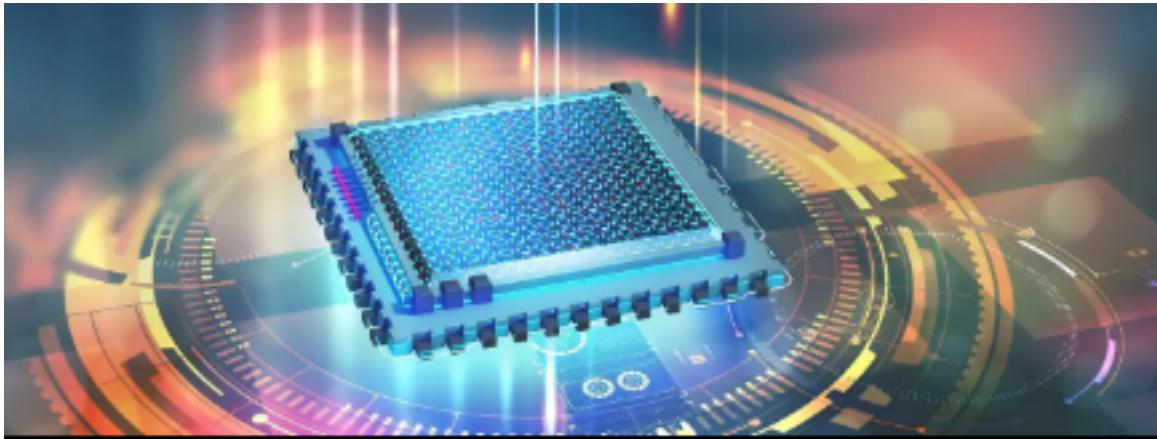
The **I/O bus** transfers data between the CPU and the system input/output devices.

The **control bus** uses binary signals to synchronize actions of all devices attached to the system bus.

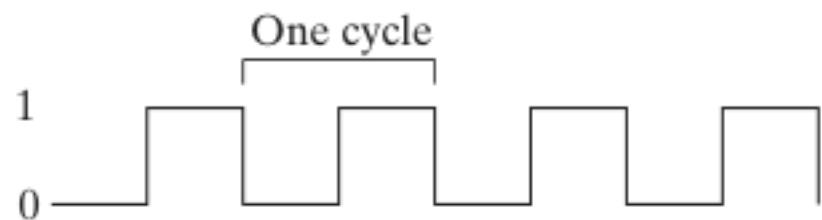
The **address bus** holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

## CLOCK AND CLOCK CYCLE

Sure, I can expand on this and re-write it in bullet points for easier reading:



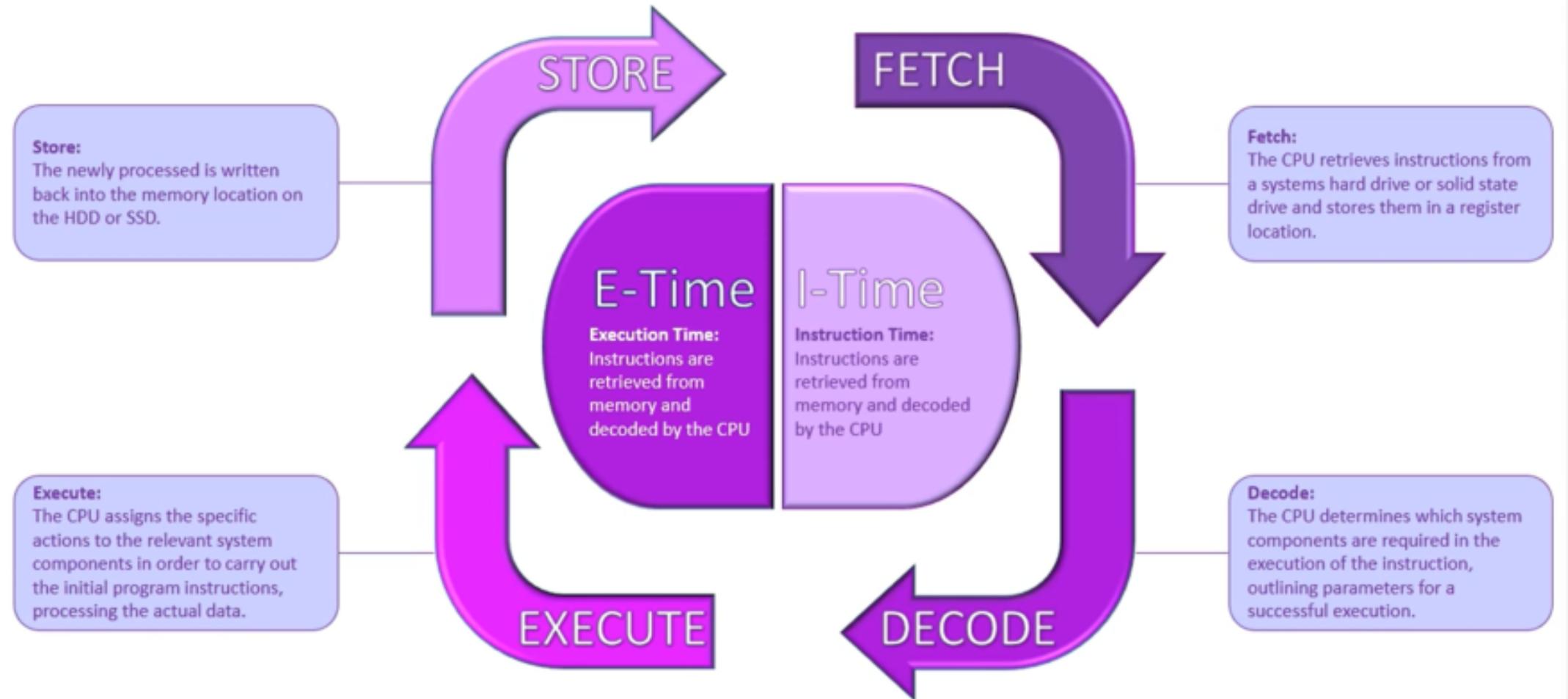
- The CPU and system bus operations are synchronized by an internal clock pulsing at a constant rate.
- The length of a clock cycle is the time required for one complete clock pulse, measured in oscillations per second.



- The duration of a clock cycle is calculated as the reciprocal of the clock's speed. For example, a

clock that oscillates 1 billion times per second (1 GHz) produces a clock cycle with a duration of one billionth of a second (1 nanosecond).

- A machine instruction requires at least one clock cycle to execute, and some instructions require over 50 clocks (such as the multiply instruction on the 8088 processor).
- Instructions that require memory access often have empty clock cycles called wait states due to differences in the speeds of the CPU, system bus, and memory circuits.
- The CPU goes through a predefined sequence of steps to execute a machine instruction, called the instruction execution cycle.
- Assuming the instruction pointer register holds the address of the instruction to execute, the steps are:
  1. **Fetch:** The CPU fetches the instruction from memory by sending the address from the instruction pointer register to the memory circuits over the system bus.
  2. **Decode:** The CPU decodes the instruction to determine what operation to perform and what operands to use.
  3. **Execute:** The CPU performs the operation on the operands according to the instruction.
  4. **Store:** The CPU stores the result of the operation back to memory or to a register.
- This sequence of steps is repeated for each instruction executed by the CPU.

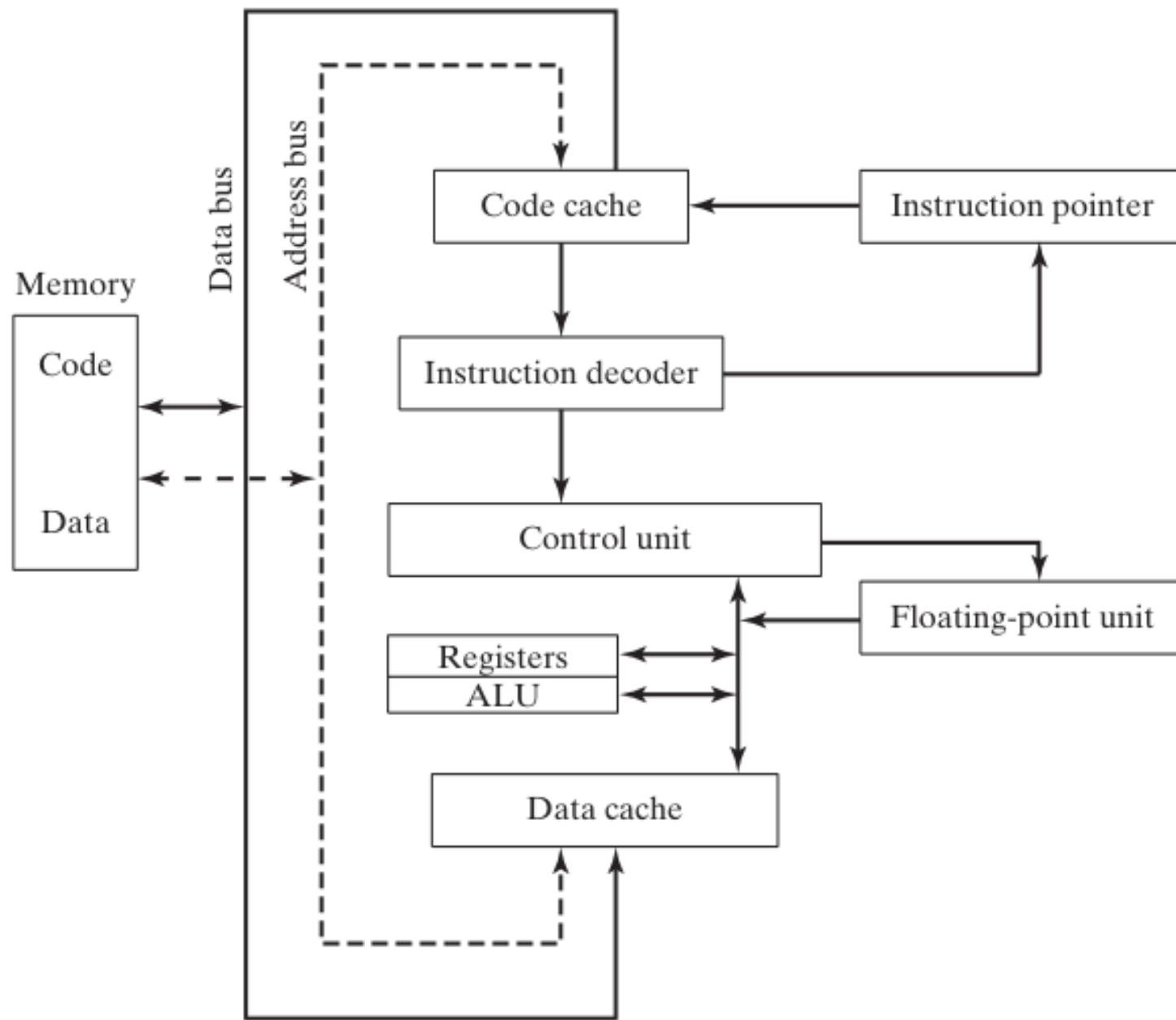


## FETCH, DECODE, EXECUTE, STORE

The CPU executes instructions by following a sequence of steps: fetching the instruction from memory and incrementing the instruction pointer, decoding the instruction to determine the operation and operands, fetching any necessary operands from registers or memory, executing the instruction and updating status flags, and storing the result if necessary. This process is often simplified to fetch, decode, and execute. The CPU interacts with other components, such as the memory controller, instruction decoder, control unit, and ALU, using digital signals and the system clock. Operand

values can be inputs or outputs to an operation, and a block diagram can be used to show data flow and relationships between components during the instruction execution cycle.

FIGURE 2–2 Simplified CPU block diagram.



## READING FROM MEMORY

**Reading from computer memory is slower than accessing internal registers** due to the four steps involved: placing the value you want to read on the address bus, asserting or change the value the processor's RD pin(ReadPin), waiting one clock cycle for the memory to respond, and copying the data from the data bus to the destination operand. To reduce the time spent reading and writing memory, CPU designers developed **cache memory**, which stores recently used instructions and data in high-speed memory. The most commonly used instructions and data are kept in cache memory, which is faster than conventional RAM as it is constructed from a special type of memory chip called **static RAM** that does not need constant refreshing. **Level-1 cache** is stored on the CPU, while **level-2 cache** is attached to the CPU by a high-speed data bus. When data is found in cache memory, it's called a **cache hit**, while a **cache miss** happens when the data is not found in cache.



### Why does memory access take more machine cycles than register access?

Memory access takes more machine cycles than register access because reading from memory involves multiple steps, including placing the memory address on the address bus, asserting the read signal, waiting for memory to respond, and copying the data from the data bus to the destination operand. Each of these steps typically takes one clock cycle, and the entire process can be slower than accessing registers, which can usually be done in a single cycle. Additionally, memory is generally slower than registers, so even if memory access only took one cycle, it would still be slower than

accessing registers. To address this speed difference, modern CPUs use cache memory to store frequently accessed data and instructions for quicker access.

## **LOADING AND EXECUTING A PROGRAM.**

Before a program can run, it must be loaded into memory by a program loader and the operating system must point the CPU to the program's entry point. The operating system searches for the program's filename and retrieves basic information about the program's file from the disk directory. The OS then loads the program file into memory, allocates a block of memory to the program, and enters information about the program's size and location into a descriptor table. The OS assigns the process an identification number (process ID), tracks its execution, and responds to requests for system resources. When the process ends, it is removed from memory.

JOHN R. LEVINE

# Linkers & Loaders



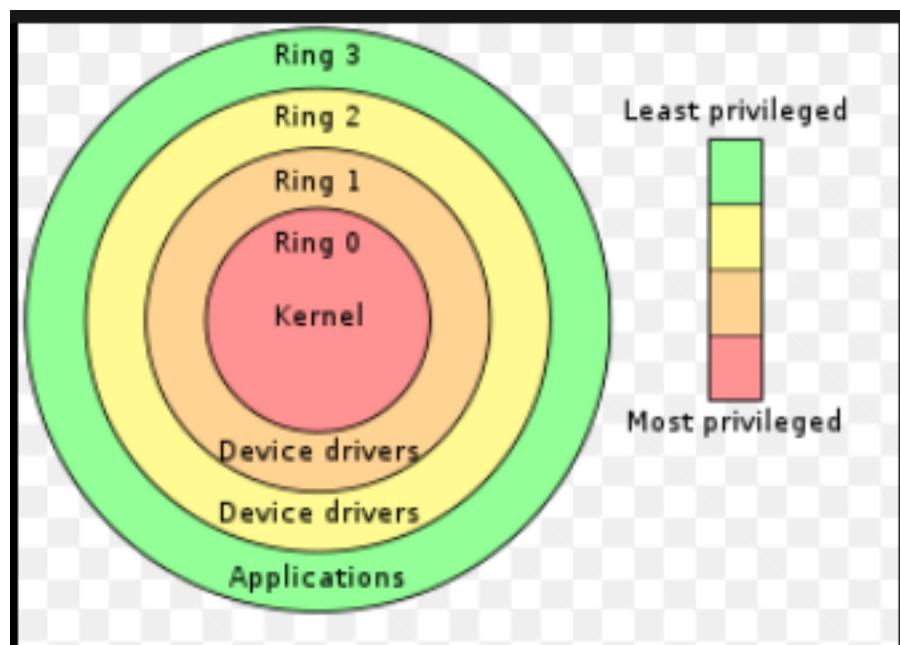
## MODES OF OPERATIONS IN X86 PROCESSORS

Here are some bullet points on modes of operation in x86 processors:

x86 processors have different modes of operation, which determine the level of privilege and the types of instructions that can be executed. The modes of operation in x86 processors include real

mode, protected mode, virtual-8086 mode, and long mode.

- **Protected mode** is a more advanced mode that provides memory protection and multitasking capabilities, allowing multiple programs to run simultaneously. Protected mode is the native state of the processor, in which all instructions and features are available. Programs are given separate memory areas named segments, and the processor prevents programs from referencing memory outside their assigned segments. Protected Mode, on the other hand, is a more advanced operating mode that was introduced in later x86 processors. It provides a more complex memory model with features such as virtual memory and memory protection. In Protected Mode, kernel ring protection is implemented through hardware support such as privilege levels and protection rings, which restrict access to memory and system resources based on the privilege level of the executing code.



- **Virtual-8086 mode** is a type of protected mode that allows legacy 16-bit applications to run in a protected environment without requiring modifications to the code. While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe environment. In other words, if a program crashes or attempts to write data into the system memory

area, it will not affect other programs running at the same time. A modern operating system can execute multiple separate virtual-8086 sessions at the same time.



- **Long mode** is the mode used by 64-bit x86 processors, which provides access to larger amounts of memory and increased processing power.

# long

- **Real-address mode** implements the programming environment of an early Intel processor with a few extra features, such as the ability to switch into other modes. This mode is useful if a program requires direct access to system memory and hardware devices. Real Mode is an operating mode of x86 processors that is compatible with the earliest IBM PC models. It provides a simple memory model and lacks the memory protection features of modern operating systems. In Real Mode, kernel ring protection can be implemented through software techniques such as segmentation, where code and data are separated into different segments and access to these segments is controlled through **segment descriptors**.

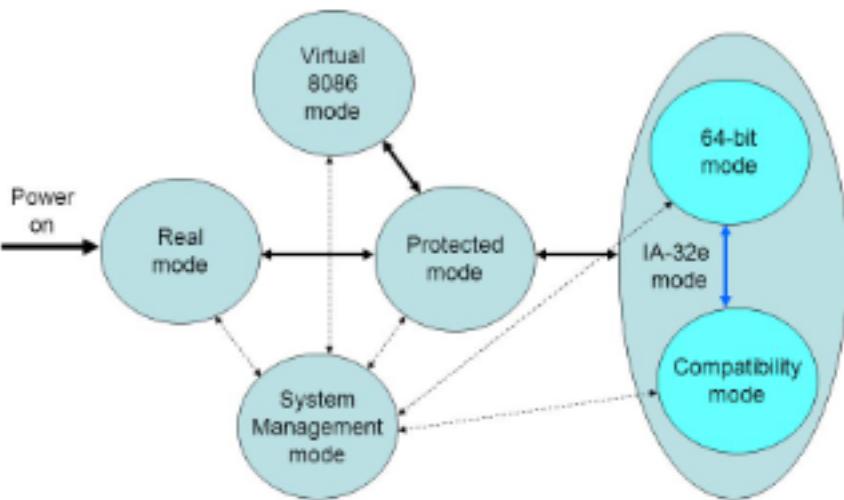


- **System management mode (SMM)** provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.



To switch between modes in x86 processors, the processor must execute a specific instruction or sequence of instructions, which may require special privileges or access to certain resources.

## The x86 operating modes



Each mode of operation in x86 processors has its own set of registers, memory management system, and interrupt handling mechanism.

The operating system typically determines the mode of operation in x86 processors, and applications must conform to the rules and limitations of the selected mode.

## ADDRESS SPACE

In 32-bit protected mode, a program or task can access up to 4 gigabytes of memory using linear addressing. This mode was introduced in later x86 processors and provides advanced features such as virtual memory and memory protection. With extended physical addressing, processors starting from the P6 model can address up to 64 gigabytes of physical memory.

On the other hand, real-address mode programs are limited to accessing only 1 megabyte of memory. This mode is compatible with the earliest IBM PC models and lacks the memory protection and virtual memory features of protected mode.

When running multiple programs in virtual-8086 mode within protected mode, each program is allocated its own 1 megabyte memory area. Virtual-8086 mode allows real-mode programs to run in a protected environment, providing them with access to the advanced features of protected mode while maintaining compatibility with legacy software.

In summary, protected mode provides a much larger addressable memory space, advanced features such as virtual memory and memory protection, and compatibility with real-mode programs through virtual-8086 mode. Real-address mode, on the other hand, is limited to accessing only 1 megabyte of memory and lacks these advanced features.

## ADDRESSING

**Addressing** is the process of referring to a specific memory location in a computer's memory system, such as RAM or a hard disk. Every memory location in a computer system has a unique address, which can be used to retrieve or store data from that location.

**Address space** refers to the range of memory addresses that a particular computer system can access. In other words, it is the total amount of memory that a processor can address or access. The size of the address space is determined by the number of bits used to represent memory addresses.

For example, in a 32-bit computer system, the processor can address a maximum of 4 gigabytes of memory, which is equivalent to  $2^{32}$  individual memory addresses. In contrast, a 64-bit computer system can address a maximum of 18 exabytes of memory, which is equivalent to  $2^{64}$  individual memory addresses.

The size of the address space is an important factor in determining the performance and capabilities

of a computer system. A **larger address space** allows for more memory to be accessed and can improve the performance of memory-intensive applications, while a smaller address space may limit the amount of memory that can be used and restrict the performance of the system.

## OPCODE AND OPERAND

In computer programming, an **operand** is a data value on which an operation is performed, while an **opcode** (operation code) is a code that represents the operation to be performed by the CPU.

The **opcode** specifies the type of operation that needs to be performed, such as addition, subtraction, multiplication, or division. The **operands**, on the other hand, are the values or variables that the operation is performed on. For example, in the expression " $x = y + z$ ", "+" is the opcode, while "y" and "z" are the operands.

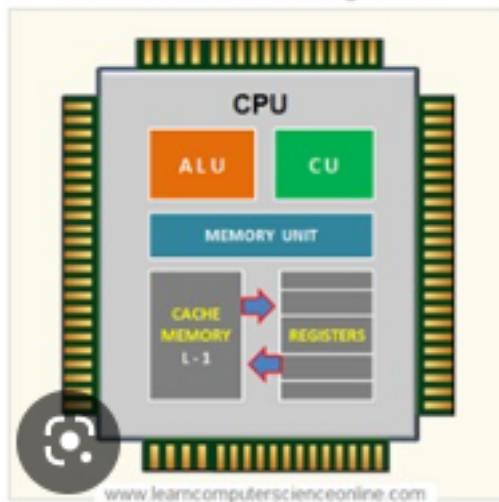
In a CPU's instruction set, an opcode typically appears in the form of a binary code that is associated with a particular operation. The CPU reads the opcode from memory and performs the corresponding operation on the operands that are specified in the instruction.

In summary, an opcode is a code that represents the operation to be performed, while an operand is a data value or variable that the operation is performed on. The **opcode and operands together form an instruction** that is executed by the CPU to perform a specific operation.

## GENERAL PURPOSE REGISTERS

**Registers** are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

## Central Processing Unit



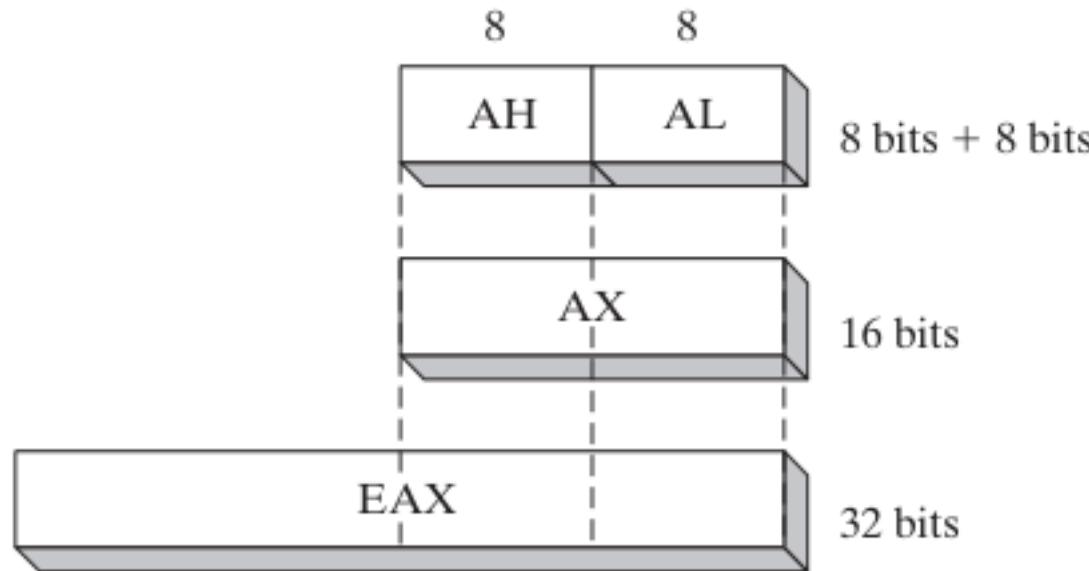
CPU Registers  
↓  
Special Purpose Registers      General Purpose Registers

The CPU Registers are **High Speed** memory area inside the processor chip  
And used by the processor during the program execution.

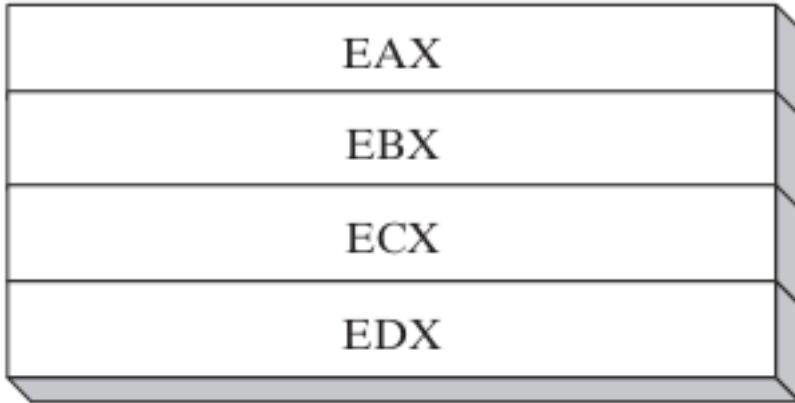
In 32-bit x86 architecture, there are eight general-purpose registers, each 32 bits in size. These registers can be used to hold data, addresses, or pointers, among other things. The eight 32-bit general purpose registers are:

“E” : Extended....

## General-purpose registers.



- **EAX (Accumulator Register):** used for arithmetic and logical operations and to store function return values.
- **EBX (Base Register):** used as a base pointer for memory access and to hold data that does not require frequent updates.
- **ECX (Counter Register):** used as a loop counter and for storing small values.
- **EDX (Data Register):** used for I/O operations and for holding high-order results of arithmetic operations.



Each register can hold a 32-bit value. The registers can be modified by arithmetic and logical operations, as well as by memory operations. The registers can be accessed by their full names (e.g., EAX, EBX) or their lower 16-bit or 8-bit parts (AX, AH, AL).

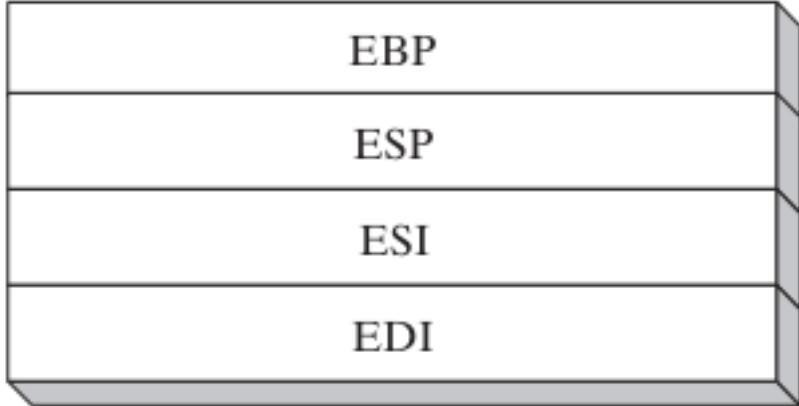
Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named AH and an 8-bit lower half named AL. The same overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

<b>32-Bit</b>	<b>16-Bit</b>	<b>8-Bit (High)</b>	<b>8-Bit (Low)</b>
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names.

<b>32-Bit</b>	<b>16-Bit</b>
ESI	SI
EDI	DI
EBP	BP
ESP	SP

- **EBP (Base Pointer Register)**: used as a base pointer for stack frames. EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the extended frame pointer register.
- **ESP (Stack Pointer Register)**: used to point to the top of the stack. ESP addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer. It is often called the extended stack pointer register. Some registers have specific uses, such as EBP and ESP for stack operations.
- **ESI (Source Index Register)**: used as a source pointer for string operations.
- **EDI (Destination Index Register)**: used as a destination pointer for string operations. ESI and EDI are used by high-speed memory transfer instructions.

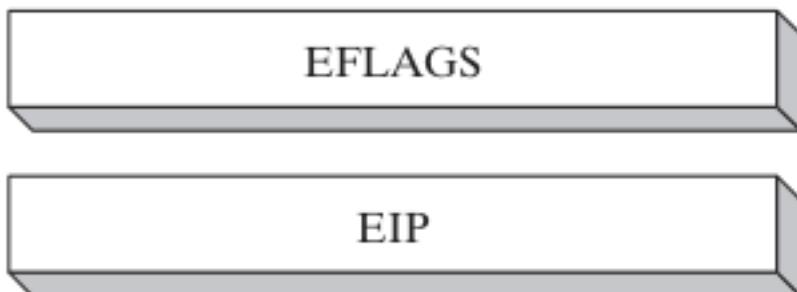


The values in the registers are lost when the program terminates, so they are not persistent.

## SPECIAL PURPOSE REGISTERS

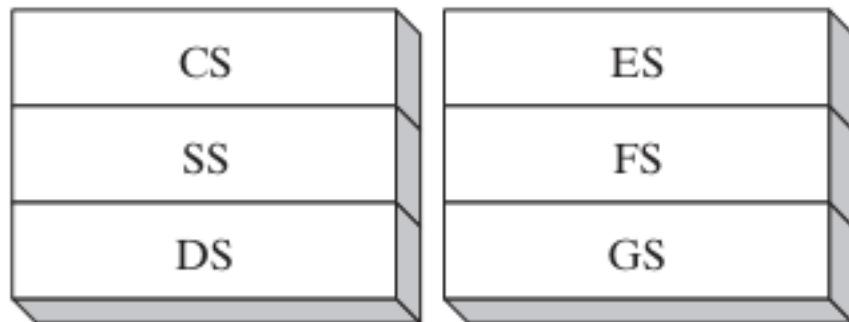
Here are the names of the special-purpose registers in x86 architecture:

1. **Instruction Pointer (IP)**: points to the memory address of the next instruction to be executed.
2. **Flags Register (FLAGS or EFLAGS)**: contains flags that indicate the state of the CPU after an arithmetic or logical operation.



3. **Stack Pointer (SP or ESP)**: points to the top of the stack.
4. **Base Pointer (BP or EBP)**: used as a base pointer for stack frames. This register is called the base pointer as its most common use is to point to the base of a stack frame during function calls. However, unlike the previously discussed registers, you may use any other register for this purpose if needed.
5. **Index Registers (SI, DI, ESI, and EDI)**: used for string and array operations.
6. **Segment Registers (CS, DS, SS, ES, FS, and GS)**: store the starting address of a memory segment.

### 16-Bit Segment Registers



7. **Control Registers (CR0, CR1, CR2, CR3, CR4, CR8)**: used to control various aspects of the processor and system, such as memory management and virtualization.
8. **Debug Registers (DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7)**: used for debugging purposes, such as setting breakpoints and monitoring memory access.
9. **Model-Specific Registers (MSR)**: used for processor-specific configuration and performance tuning.
10. **Task Register (TR)**: used for task switching in protected mode.
11. **Local Descriptor Table Register (LDTR)**: points to the local descriptor table in protected mode.

## **NOTE:**

ESP and EBP are both special purpose and general purpose?

Yes, both ESP and EBP are considered special-purpose registers because they have dedicated functions and are not used for general arithmetic or logical operations.

ESP, or the Stack Pointer, is used to point to the top of the stack, which is important for managing program flow and function calls.

EBP, or the Base Pointer, is used as a base pointer for stack frames, which helps with debugging and optimization.

However, ESP and EBP can also be used for general data storage and manipulation, so they can be considered both special-purpose and general-purpose registers. This is in contrast to registers like EAX or EBX, which are used exclusively for general arithmetic and logical operations and are therefore considered general-purpose registers.

**"E" in these registers stands for extended, right?**

Yes, in x86 architecture, the **"E" prefix** in registers such as EAX, EBX, ECX, and so on stands for "extended." These registers are 32-bit general-purpose registers and are used in **32-bit mode (also known as protected mode)** of the x86 architecture.

The "E" prefix is used to differentiate these 32-bit registers from their 16-bit counterparts, which were used in 16-bit mode (also known as **real mode**) of the x86 architecture. The 16-bit registers are named AX, BX, CX, and so on.

In **64-bit mode (also known as long mode)** of the x86 architecture, the **"R" prefix** is used for registers that are 64 bits wide, such as RAX, RBX, RCX, and so on.

# SEGMENT REGISTERS

Are part of special purpose registers.

Here are some key points about segment registers in x86 architecture:

**Segment registers** are registers that hold **segment selectors**, which are used to access different memory segments.

In real-address mode, segment registers are 16 bits wide and point to fixed memory areas called segments.

In protected mode, segment registers hold selectors that point to **segment descriptor tables**.

These tables contain **information about the memory segment**, including its base address, size, access rights, and other attributes.

In protected mode, there are six segment registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment), FS (F Segment), and GS (G Segment).

Each segment has a **base address** and a **limit**, which are **specified in the segment descriptor table**.

The **base address** is the starting address of the segment, and the **limit** is the size of the segment. The **limit** is used to prevent programs from accessing memory outside the segment.

- The **Code Segment (CS) register** points to the segment that contains the current code being

executed.

- The **Data Segment (DS) register** points to the segment that contains program data.
- The **Stack Segment (SS) register** points to the segment that contains the program stack.
- The **Extra Segment (ES) register** is a general-purpose segment register that can be used for various purposes, such as holding additional data.
- The **F Segment (FS) and G Segment (GS) registers** are additional segment registers that were added in later versions of the x86 architecture. They are typically used for thread-local storage, which allows each thread of a multi-threaded program to have its own copy of certain variables.

In real-address mode, the CPU uses 16-bit segment registers to access memory directly.

These registers include CS (code segment), DS (data segment), SS (stack segment), and ES (extra segment), and they **hold the base addresses** of preassigned memory areas called segments.

In protected mode, segment registers are still used, but they hold pointers to segment descriptor tables (SDTs) instead of directly accessing memory. The SDTs are stored in memory and provide information about the size, location, and access rights of the segments.

In protected mode, there are six segment registers instead of four in real-address mode. The additional registers are FS and GS, which can be used for storing additional data or addressing thread-local storage (TLS).

Each segment descriptor in the SDT contains a base address, a limit, and access rights information, such as **read-only**, **read-write**, **execute-only**, or **execute-read**.

The segment registers are used to calculate the physical memory address that corresponds to a logical address used by a program. This is done by adding the offset of the logical address to the base address of the corresponding segment register.

The segment registers can be **modified** using special instructions, such as MOV, PUSH, and POP. However, **in protected mode**, changing the segment register value will not directly change the physical memory address accessed by a program, since the address is calculated based on the segment descriptor in the SDT.

The values of segment registers can be changed using the "MOV" instruction, which allows a new selector value to be loaded into the register.

The selector is a 16-bit or 32-bit value that points to an entry in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT), which contains information about the segment.

The "MOV" instruction takes two operands: **the destination operand**, which is the segment register that will be modified, and **the source operand**, which is the new selector value to be loaded into the register.

For example, to load a new value into the DS (data segment) register, the following instruction can be used:

`MOV DS, new_selector.`

This instruction will load the value of "new\_selector" into the DS register, which will then be used to calculate the physical memory address accessed by the program.

It's important to note that changing the value of a segment register does not immediately change the memory segment accessed by the program.

Instead, the processor uses the new selector value to look up the corresponding entry in the GDT or LDT and retrieve the base address and size of the segment.

The memory address accessed by the program is then calculated by **adding the offset of the logical**

address to the base address of the segment.

## INSTRUCTION REGISTER

The EIP (Extended Instruction Pointer) is a 32-bit register in x86 architecture that holds the address of the next instruction to be executed by the CPU.

When the CPU fetches an instruction from memory, it reads the opcode from the address pointed to by EIP and then increments EIP to point to the next instruction.

Certain machine instructions, such as JMP (jump) and CALL (call subroutine), can manipulate the value of EIP to cause the program to branch to a new location in memory.

For example, a JMP instruction can be used to unconditionally jump to a different location in the code, while a CALL instruction can be used to call a subroutine and save the return address in the EIP register.

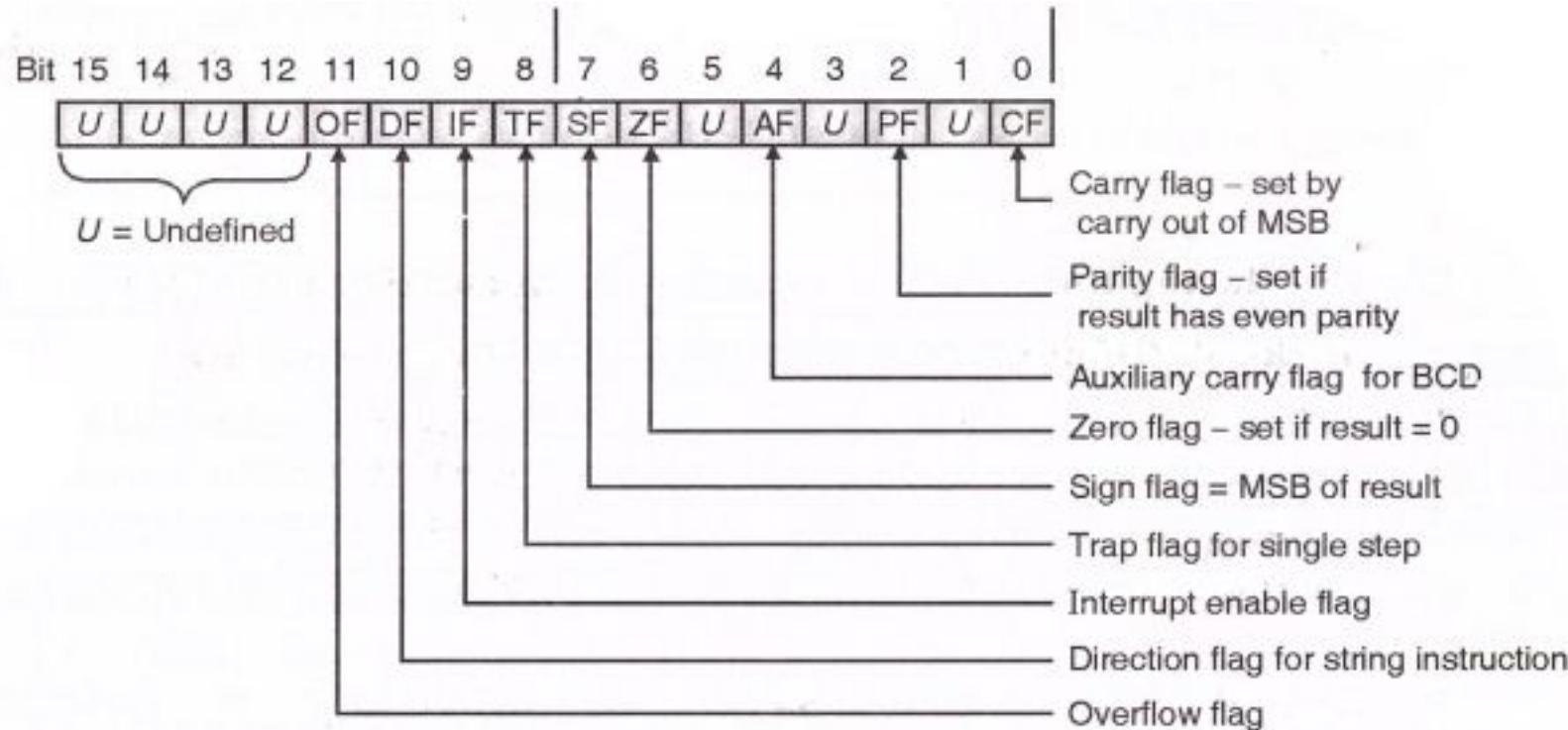
The EIP register is part of the larger EFLAGS (Extended Flags) register, which contains various CPU flags and control bits that are used to control the behavior of the CPU and indicate the results of arithmetic and logical operations.

The EIP register can be modified using certain machine instructions, such as JMP, CALL, RET (return from subroutine), and INT (interrupt), as well as by software interrupts or hardware interrupts that cause the CPU to jump to a different location in memory.

# FLAG REGISTERS

The **EFLAGS or FLAGS register** is a 32-bit register that contains various control bits and flags that are used to control the behavior of the CPU and indicate the results of arithmetic and logical operations. For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode. Programs can set individual bits in the EFLAGS register to control the CPU's operation. Examples are the **Direction** and **Interrupt** flags.

"Control Flag" is not a specific flag in the **x86 EFLAGS register**. However, the EFLAGS register as a whole can be thought of as a **control register**, since its various flags and control bits are used to control the behavior of the CPU and its operations. The EFLAGS register is sometimes referred to as the **"Flag Register"** or the **"Status Register"**, but these terms are not always used consistently across different sources or programming environments. In general, it's important to consult the specific documentation or reference material for a given software or hardware platform to understand the meaning and usage of specific flags and registers.



8086 flag register format

**Carry Flag (CF):** This flag is set when an arithmetic operation generates a carry or borrow out of the most significant bit of the result, and is used to support multi-word arithmetic operations and binary-coded decimal (BCD) arithmetic.

**Parity Flag (PF):** This flag is set if the least significant byte of an operation result contains an even number of set bits (i.e., the result has even parity), and is used for parity checking and error detection.

**Auxiliary Carry Flag (AF):** This flag is set when an arithmetic operation generates a carry or borrow out of the low nibble (bits 0-3) of the result, and is used to support BCD arithmetic.

**Zero Flag (ZF):** This flag is set when an operation generates a result of zero, and is used to test for null pointers and empty data structures.

**Sign Flag (SF):** This flag is set when the most significant bit of an operation result is set (i.e., the result is negative), and is used for signed arithmetic and conditional branching.

**Trap Flag (TF):** The trap flag (TF) is a specific flag in the x86 EFLAGS register that is used to enable single-step debugging. When the TF is set (i.e., 1), the CPU will automatically generate a trap interrupt after executing each instruction, allowing a debugger to inspect the state of the program and step through the code one instruction at a time. The trap flag is typically used in conjunction with hardware or software breakpoints, which temporarily modify the instruction stream of a program in order to halt its execution at a specific point. By setting the TF flag, a programmer or debugger can control the flow of a program and observe its behavior in detail, making it easier to diagnose and fix bugs and other issues. It's worth noting that the use of the TF flag can slow down the execution of a program considerably, as it adds additional overhead to each instruction cycle. For this reason, it is typically used only when debugging or profiling code, and is disabled during normal program execution.

**The Direction flag (DF)** controls the direction of string operations (such as REP MOVSB, REP STOSB, etc.) that copy or move blocks of data between memory locations. When the DF is set (i.e., 1), the string operations proceed in reverse order, from high memory addresses to low memory addresses. When the DF is clear (i.e., 0), the string operations proceed in forward order, from low memory addresses to high memory addresses.

**The Interrupt flag (IF)** controls the ability of the CPU to respond to external interrupts from devices or other sources. When the IF is set (i.e., 1), the CPU can respond to interrupts by executing the appropriate interrupt service routine (ISR) in response to an interrupt request. When the IF is clear (i.e., 0), the CPU disables external interrupts and will not respond to them.

**Overflow Flag (OF):** This flag is set when an arithmetic operation generates a result that is too large or too small to be represented in the available bit width of the operands, and is used to

detect signed arithmetic overflow and underflow.

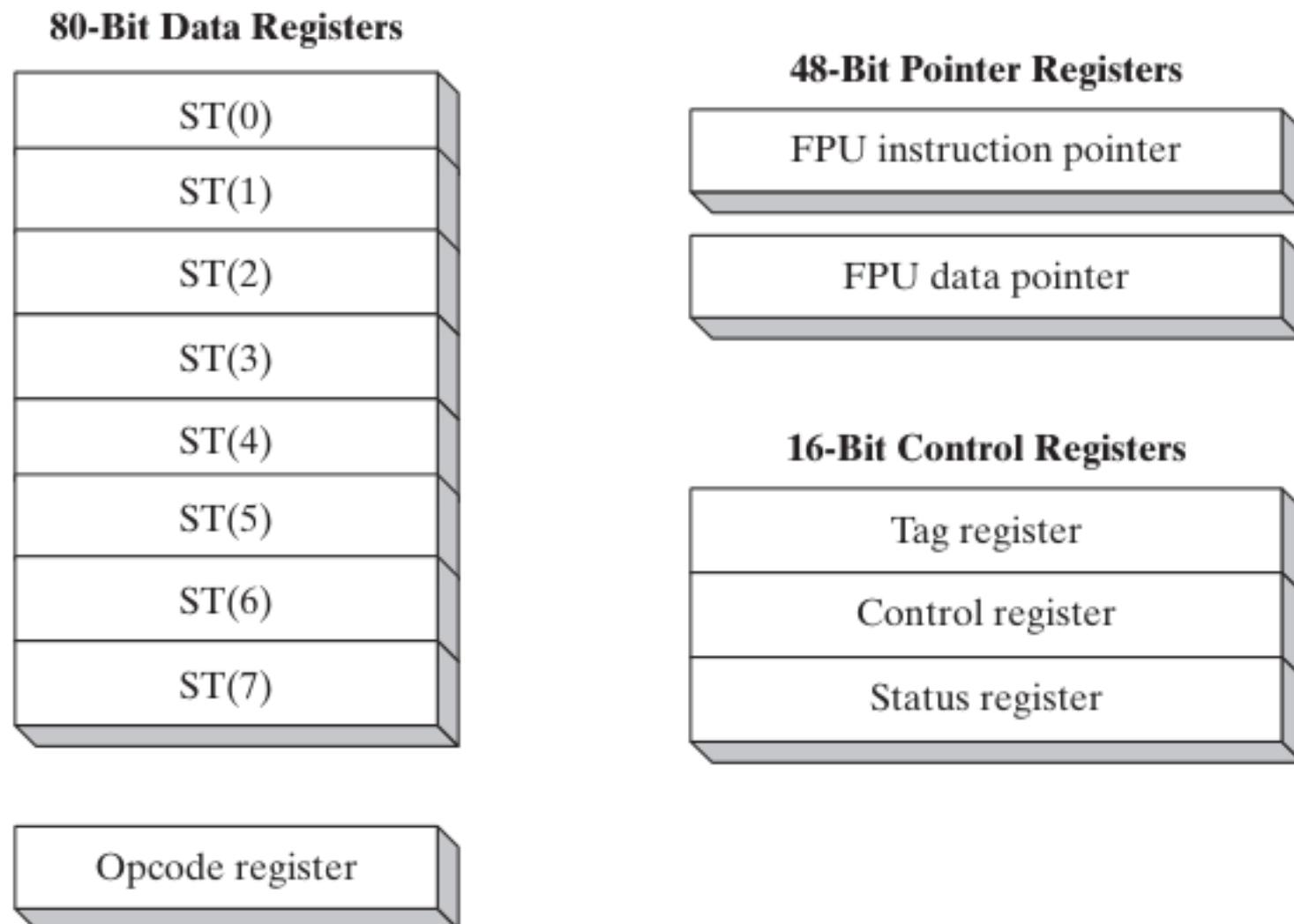
## MMX AND XMM REGISTERS

The **MMX technology** is a feature present in Intel processors that significantly improves their performance when implementing advanced multimedia and communication applications. These processors come equipped with eight 64-bit MMX registers that support **SIMD (Single-Instruction, Multiple-Data) instructions**. These instructions operate in parallel on data values contained in the MMX registers. While these registers appear to be separate, they are actually aliases for the same registers used by the floating-point unit.

The x86 architecture also includes eight 128-bit registers called **XMM registers**, which are used by streaming SIMD extensions to the instruction set. These registers are designed to enhance the performance of multimedia and signal processing applications. The MMX and XMM registers allow for the efficient handling of multimedia and other data-intensive tasks, providing developers with tools to optimize their software for Intel processors.

The **floating-point unit (FPU)** is responsible for high-speed floating-point arithmetic. In the past, a separate coprocessor chip was required to perform these calculations. However, from the Intel486 onwards, the FPU has been integrated into the main processor chip. There are eight floating-point data registers in the FPU, each named ST(0) through ST(7). These registers allow for the efficient processing of floating-point numbers and other **complex mathematical operations**.

FIGURE 2–5 Floating-point unit registers.



Overall, the MMX, XMM, and FPU registers provide Intel processors with the ability to perform complex mathematical calculations and multimedia processing at high speeds. They offer developers a range of tools to optimize their applications for Intel processors, ensuring maximum performance and efficiency.

<b>Register Name</b>	<b>Size</b>	<b>Description</b>
FPU Data Registers	80-bit	There are eight FPU data registers named ST(0)-ST(7). These registers are used to store floating-point numbers and other complex mathematical operations.
Opcode Register	16-bit	The opcode register contains the opcode for the currently executing instruction. This register is read-only and is updated by the processor during instruction execution.
48-bit Pointer Regs.	48-bit	The 48-bit pointer registers are used in protected mode to hold pointers to segment descriptor tables. These registers are used to calculate the physical address of memory locations in protected mode.
Control Registers	16-bit	There are four control registers named CR0-CR3. These registers contain system control flags that govern the behavior of the processor. They are used to set up memory management, protection, and other system-level features. Control registers are read-only in user mode and read/write in supervisor mode.

## **x86 MEMORY MANAGEMENT**

x86 processors manage memory according to the basic modes of operation discussed.

**Protected mode** is the most robust and powerful, but it does restrict application programs from directly accessing system hardware.

In **real-address mode**, only 1 MByte of memory can be addressed, from hexadecimal 00000 to FFFFF. The processor can run only one program at a time, but it can momentarily interrupt that program to process requests (**called interrupts**) from peripherals. Application programs are permitted to access any memory location, including addresses that are linked directly to system hardware. The MS-DOS operating system runs in real-address mode, and Windows 95 and 98 can be booted into this mode.

In **protected mode**, the processor can run multiple programs at the same time. It assigns each process (running program) a total of 4 GByte of memory. Each program can be assigned its own reserved memory area, and programs are prevented from accidentally accessing each other's code and data. MS-Windows and Linux run in protected mode.

In **virtual-8086 mode**, the computer runs in protected mode and creates a virtual-8086 machine with its own 1-MByte address space that simulates an 80x86 computer running in real-address mode. Windows NT and 2000, for example, create a virtual-8086 machine when you open a Command window. You can run

many such windows at the same time, and each is protected from the actions of the others. Some MS-DOS programs that make direct references to computer hardware will not run in this mode under Windows NT, 2000, and XP. Chapter 11 explains many more details of both real-address mode and protected mode.

## x86-64 PROCESSOR vs x86 PROCESSORS

The x86-64 instruction set is an extension of the x86 instruction set, but with several important features, such as 64-bit long addresses that allow for a virtual address space of size  $2^{64}$  bytes, 64-bit general-purpose registers, and **eight additional general-purpose registers** compared to the x86.

These processors use a 48-bit physical address space, which can support up to 256 terabytes of RAM. However, they do not support 16-bit real mode or virtual-8086 mode when running in native 64-bit mode. While x86-64 is technically an instruction set, it is commonly referred to as a processor type, and for the purpose of learning assembly language, it is not necessary to consider hardware implementation differences between processors that support x86-64.

**The x86-64 instruction set is a 64-bit extension of the x86 instruction set, and it is backward-compatible with the x86 instruction set.** In 64-bit mode, the processor runs applications that use the 64-bit linear address space, and it enables 64-bit instruction operands. There are two modes in the Intel 64 architecture, namely compatibility mode and 64-bit mode. Compatibility mode enables existing 16-bit and 32-bit applications to run without being recompiled, while 64-bit mode is the native mode for 64-bit Microsoft Windows.

In terms of registers, 64-bit processors have sixteen **64-bit general-purpose registers**, eight **80-bit floating-point registers**, a **64-bit status flags register** named **RFLAGS** (only the lower 32 bits are used), and a **64-bit instruction pointer** named **RIP**.

There are also specialized registers for multimedia processing, including eight **64-bit MMX registers** and **sixteen 128-bit XMM registers**. By **adding the REX prefix to each instruction**, the operands can be

extended to 64 bits, and a total of 16 general-purpose registers become available. The general-purpose registers can access 8-bit, 16-bit, 32-bit, or 64-bit operands, and in 64-bit mode, the default operand size is 32 bits.

Operand Size	Available Registers
8 bits	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L
16 bits	AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
32 bits	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
64 bits	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

Here are a few more details to remember: In 64-bit mode, a single instruction cannot access both a high-byte register, such as AH, BH, CH, and DH, and at the same time, the low byte of one of the new byte registers (such as DIL). The 32-bit EFLAGS register is replaced by a 64-bit RFLAGS register in 64-bit mode. The two registers share the same lower 32 bits, and the upper 32 bits of RFLAGS are not used. The status flags are the same in 32-bit mode and 64-bit mode.

## COMPONENTS OF x86 COMPUTER

The **motherboard** is the central component of a microcomputer, consisting of a flat circuit board that houses the CPU, supporting processors, main memory, input-output connectors, power supply connectors, and expansion slots.

These components are interconnected by a set of wires etched directly on the motherboard, known as

the bus.

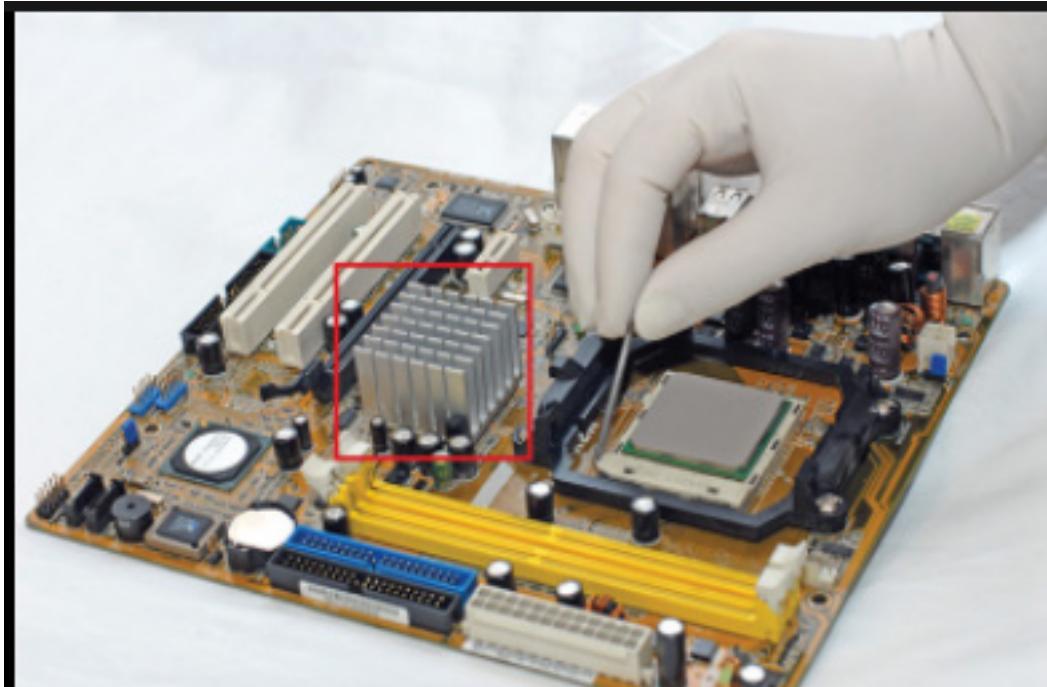
There are various types of motherboards available on the PC market, differing in terms of expansion capabilities, integrated components, and speed.

Some common components found on PC motherboards include a CPU socket (which comes in different shapes and sizes, depending on the type of processor they support), memory slots (which hold small plug-in memory boards), BIOS chips (which hold system software), CMOS RAM (which has a small circular battery to keep it powered), connectors for mass-storage devices like hard drives and CD-ROMs, USB connectors for external devices, and ports for keyboards and mice.

Assembling a computer requires careful consideration of the motherboard and the components that will be connected to it. Understanding the various components and their functions can help you make informed decisions when building or upgrading a system.

Additionally, programming at the assembly language level requires an understanding of how to interface with system hardware, firmware, and operating system functions to perform I/O operations at different levels of access.

A **motherboard chipset** is a collection of processor chips designed to work together on a specific type of motherboard with various features such as increased processing power, multimedia capabilities, or reduced power consumption.

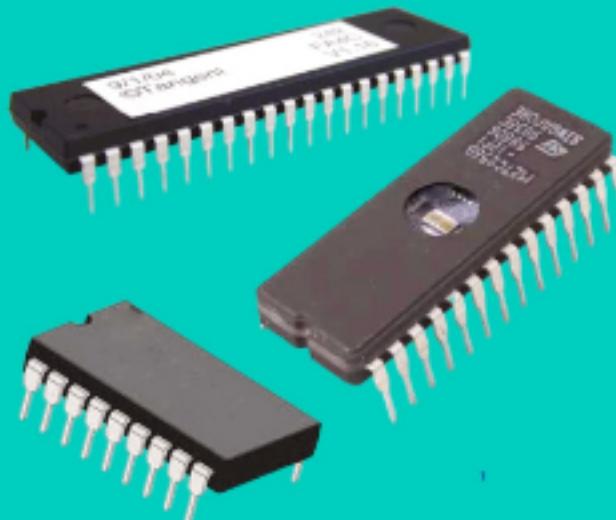


Intel Fast Memory Access uses an updated Memory Controller Hub (MCH). It can access dual-channel DDR2 memory, at an 800 MHz clock speed.

## RAM AND ROM

**Read-only memory (ROM)** is a type of memory that is permanently burned into a chip during the manufacturing process. ROM cannot be altered, and its contents cannot be erased or rewritten. ROM is often used to store the computer's BIOS (basic input-output system), which contains the firmware code that initializes the hardware during boot-up.

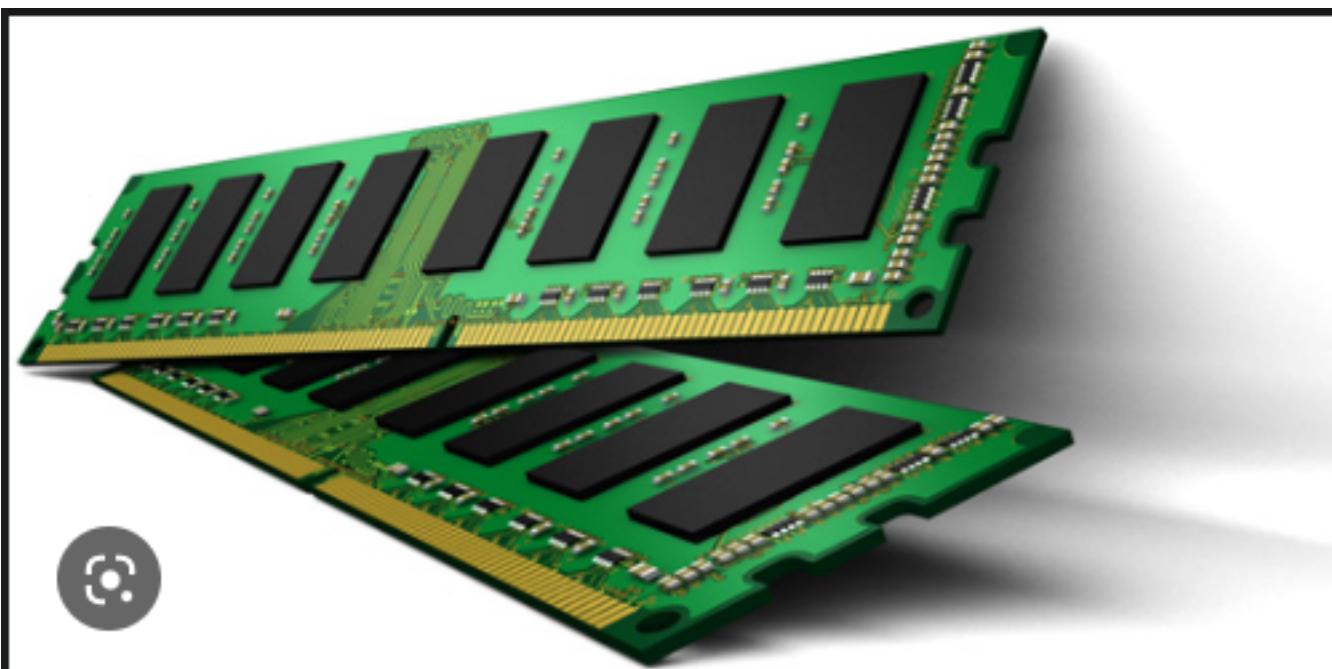
## *ROM (Read Only Memory)*



**Erasable programmable read-only memory (EPROM)** is a type of memory that can be erased and reprogrammed. To erase EPROM, ultraviolet light is used, which can take several minutes. EPROM is primarily used in embedded systems, where firmware needs to be updated frequently.



**Dynamic random-access memory (DRAM)** is the most commonly used type of memory in computers. DRAM is used as main memory where programs and data are stored when a program is running. DRAM is inexpensive, but it needs to be refreshed every millisecond to prevent data loss. Some systems use ECC (error checking and correcting) memory, which can detect and correct memory errors.



## Dynamic RAM, DRAM Memory Technology

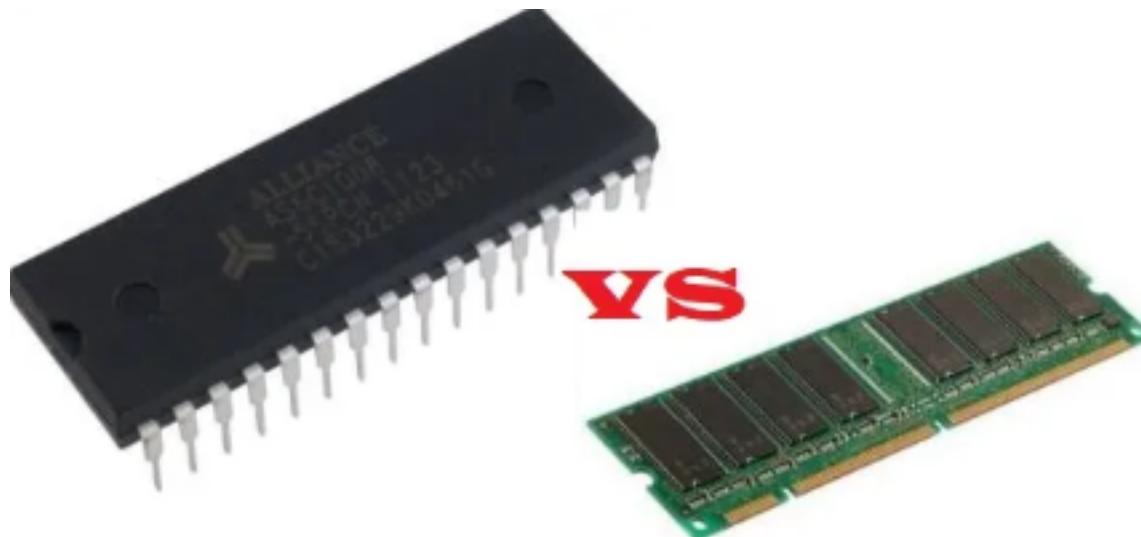
[Visit](#)

**Static RAM (SRAM)** is a type of memory that is used primarily for expensive, high-speed cache memory. SRAM does not need to be refreshed, and it is faster than DRAM. CPU cache memory is made up of SRAM.

NETWORK SETUP 101

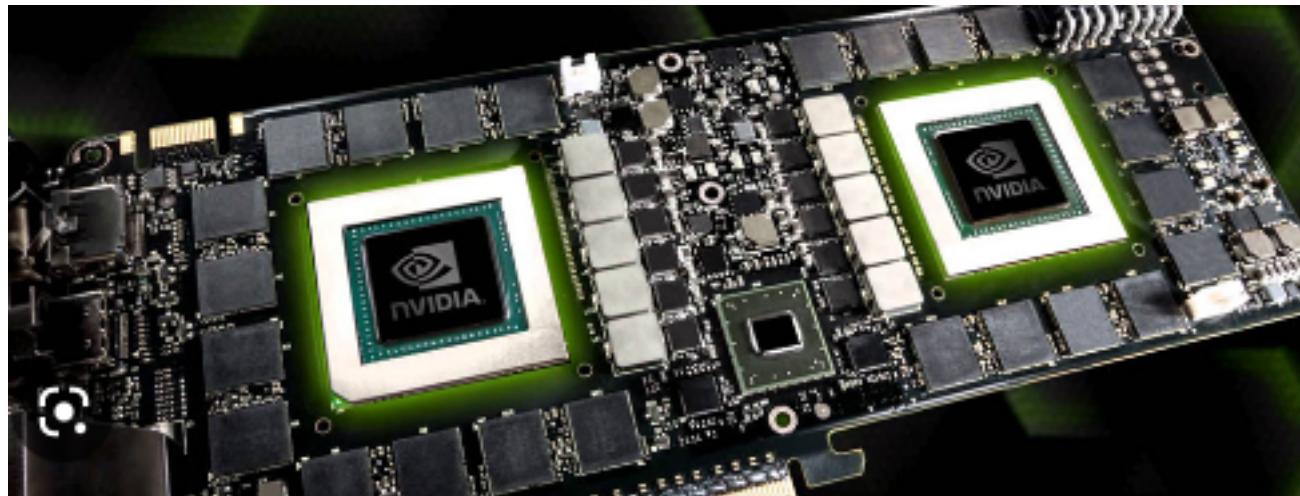
# SRAM VS. DRAM

*Which is Better?*

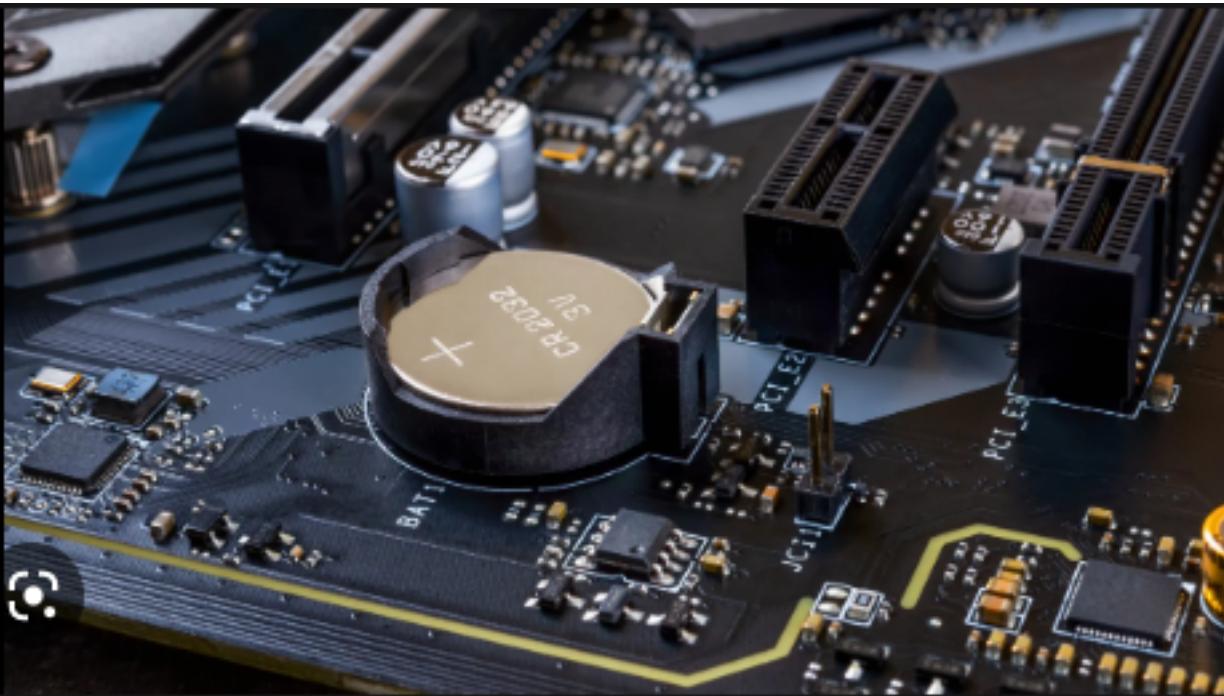


SRAM VS DRAM

**Video RAM (VRAM)** is a type of memory that is used to hold video data. VRAM is dual ported, which means that one port can continuously refresh the display while the other port writes data to the display. VRAM is designed to work with video controllers and graphics processing units (GPUs).



**Complimentary metal oxide semiconductor (CMOS) RAM** is a type of memory that is used to store system setup information on the motherboard. CMOS RAM is refreshed by a battery, which means that its contents are retained even when the computer is turned off. CMOS RAM is used to store information such as the date and time, as well as hardware configuration settings.

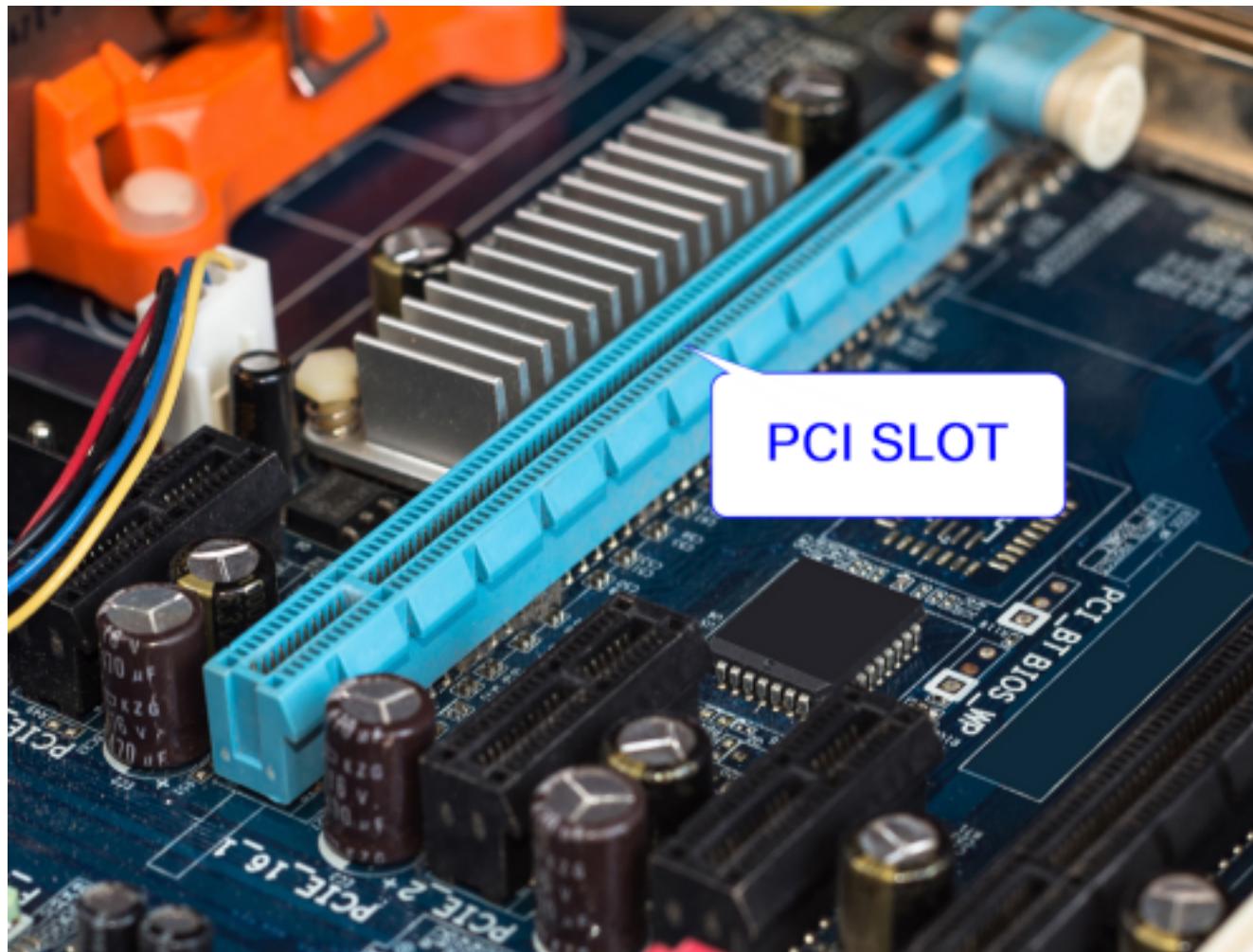


What Is CMOS?

[Visit](#)

## PCI AND PCI EXPRESS BUS ARCHITECTURE

The **Peripheral Component Interconnect (PCI)** bus is a standard for connecting various peripherals to the CPU in a computer system. It provides a high-speed data path between the CPU and devices such as hard drives, memory, video controllers, sound cards, and network controllers. The PCI bus provides a bridge between these devices and the CPU, allowing them to communicate with each other.



**PCI Express** is a newer and faster version of the PCI bus architecture. It provides two-way serial connections between devices, memory, and the processor, and carries data in packets, similar to networks. It uses separate "lanes" for data transfer, which allows it to transfer data at much higher speeds than the older PCI bus architecture. PCI Express is widely supported by graphics controllers, and is commonly used for high-speed data transfer in modern computer systems.



PCI Express x1



PCI Express x4



PCI Express x8

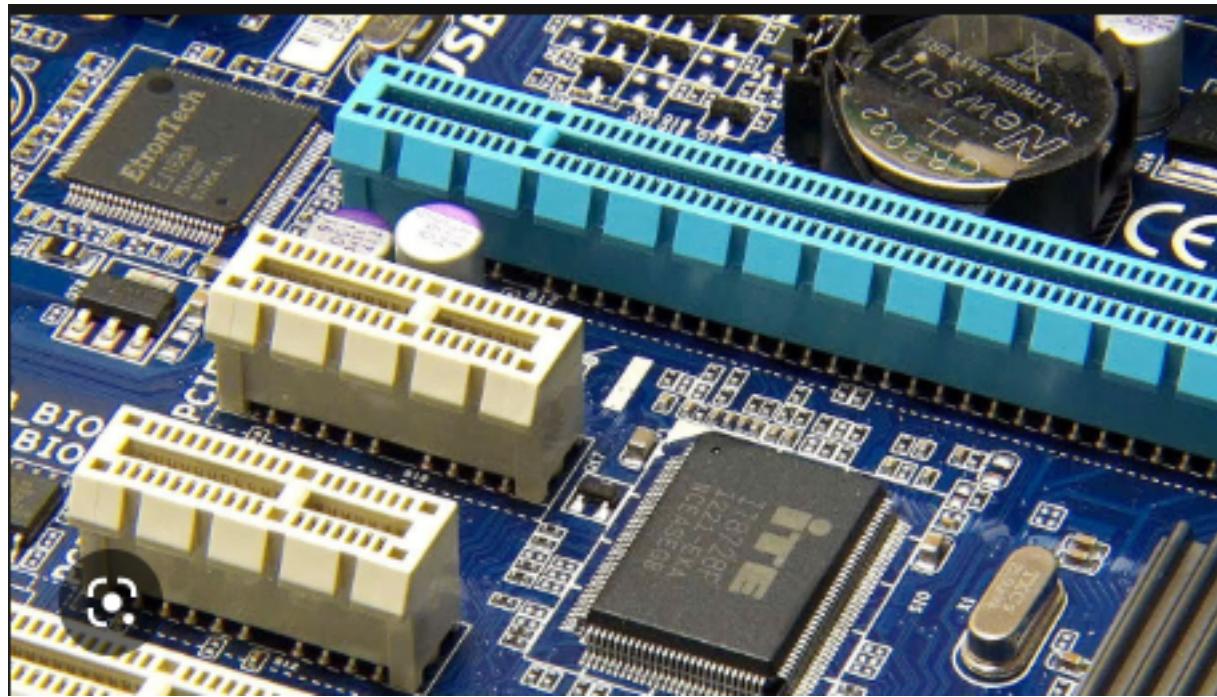


PCI Express x16



Lifewire

EG.



## LEVELS OF I/O ACCESS

I/O can be performed by calling functions provided by the operating system, instead of directly accessing hardware.

There are three primary levels of I/O access: **high-level language functions**, **operating system functions**, and **BIOS low-level subroutines** that communicate directly with hardware devices.

**High-level language functions:** A high-level programming language such as C++ or Java contains functions to perform input-output. These functions are portable because they work on a variety of different computer systems and are not dependent on any one operating system.

**Operating system:** Programmers can call operating system functions from a library known as the API (application programming interface). The operating system provides high-level operations such as writing strings to files, reading strings from the keyboard, and allocating blocks of memory.

**BIOS:** The basic input-output system is a collection of low-level subroutines that communicate directly with hardware devices. The BIOS is installed by the computer's manufacturer and is tailored to fit the computer's hardware. Operating systems typically communicate with the BIOS.

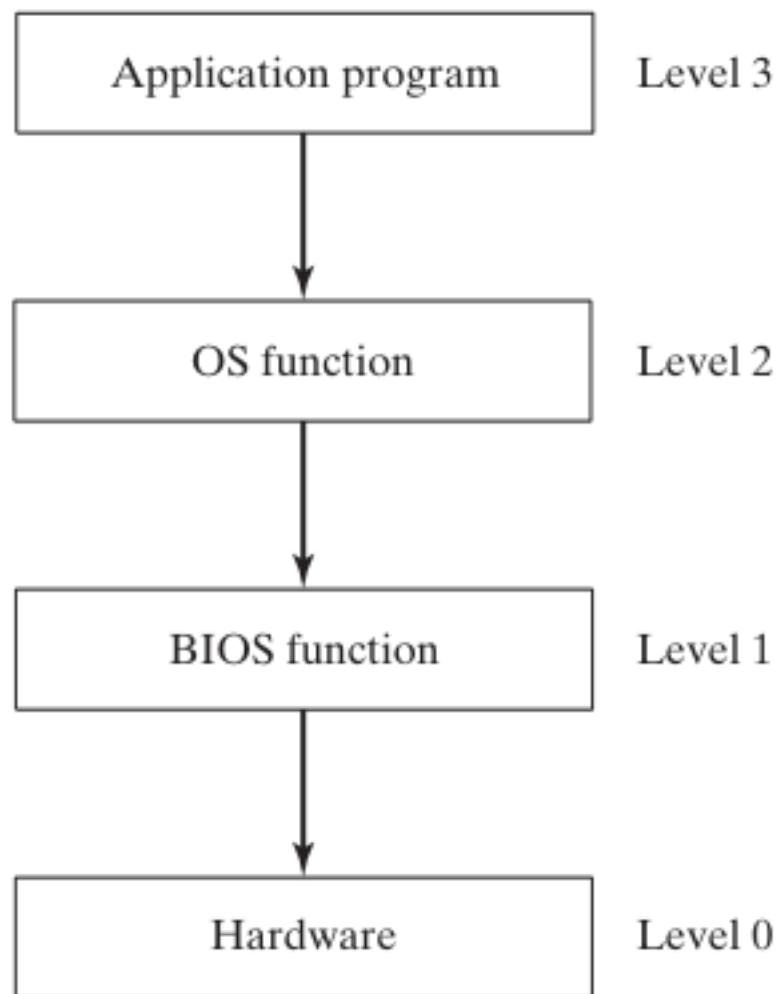
## DEVICE DRIVERS

**Device drivers** are programs that permit the operating system to communicate directly with hardware devices and the system BIOS. For example, a device driver might receive a request from the OS to read some data; the device driver satisfies the request by executing code in the device firmware that reads data in a way that is unique to the device.

Device drivers are usually installed in one of two ways: (1) before a specific hardware device is attached to a computer, or (2) after a device has been attached and identified. In the latter case, the OS recognizes the device name and signature; it then locates and installs the device driver software onto the computer.

We can put the I/O hierarchy into perspective by showing what happens when an application program displays a string of characters on the screen.

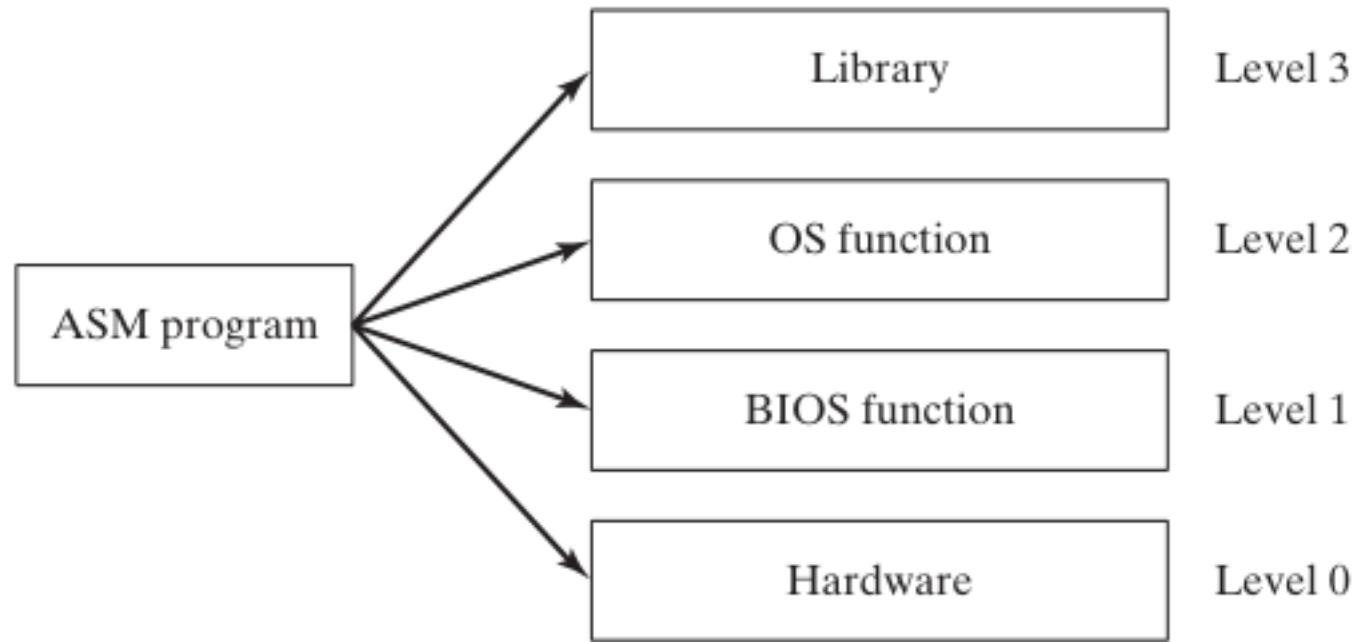
FIGURE 2–7 Access levels for input–output operations.



The following steps are involved: A statement in the application program calls an HLL library function that writes the string to standard output. The library function (Level 3) calls an operating system function, passing a string pointer. The operating system function (Level 2) uses a loop to call a BIOS subroutine, passing it the ASCII code and color of each character. The operating system calls another BIOS subroutine to advance the cursor to the next position on the screen. The BIOS subroutine (Level 1) receives a character, maps it to a particular system font, and sends the

character to a hardware port attached to the video controller card. The video controller card (Level 0) generates timed hardware signals to the video display that control the raster scanning and displaying of pixels.

FIGURE 2–8 Assembly language access levels.



## Programming at Multiple Levels

Assembly language programs have power and flexibility in the area of input-output programming. They can choose from the following access levels (Figure 2-8).

**Level 3:** Call library functions to perform generic text I/O and file-based I/O. We supply such a library with this book, for instance.

**Level 2:** Call operating system functions to perform generic text I/O and file-based I/O. If the OS uses a graphical user interface, it has functions to display graphics in a device-independent way.

**Level 1:** Call BIOS functions to control device-specific features such as color, graphics, sound, keyboard input, and low-level disk I/O.

**Level 0:** Send and receive data from hardware ports, having absolute control over specific devices.

This approach cannot be used with a wide variety of hardware devices, so we say that it is not portable. Different devices often use different hardware ports, so the program code must be customized for each specific type of device. What are the tradeoffs? Control versus portability is the primary one.

**Level 2 (OS)** works on any computer running the same operating system. If an I/O device lacks certain capabilities, the OS will do its best to approximate the intended result. Level 2 is not particularly fast because each I/O call must go through several layers before it executes.

**Level 1 (BIOS)** works on all systems having a standard BIOS, but will not produce the same result on all systems. For example, two computers might have video displays with different resolution capabilities. A programmer at Level 1 would have to write code to detect the user's hardware setup and adjust the output format to match.

Level 1 runs faster than Level 2 because it is only one level above the hardware. Level 0 (hardware) works with generic devices such as serial ports and with specific I/O devices produced by known manufacturers. Programs using this level must extend their coding logic to handle variations in I/O devices.

Real-mode game programs are prime examples because they usually take control of the computer. Programs at this level execute as quickly as the hardware will permit.

Suppose, for example, you wanted to play a WAV file using an audio controller device. At the OS level, you would not have to know what type of device was installed, and you would not be concerned with nonstandard features the card might have.

At the BIOS level, you would query the sound card (using its installed device driver software) and find out whether it belonged to a certain class of sound cards having known features.

At the hardware level, you would fine tune the program for certain models of audio cards, taking advantage of each card's special features.

**General-purpose operating systems** rarely permit application programs to directly access system hardware, because to do so would make it nearly impossible for multiple programs to run simultaneously.

Instead, hardware is accessed only by device drivers, in a carefully controlled manner. On the other hand, smaller operating systems for specialized devices often do connect directly to hardware.

They do this in order to reduce the amount of memory taken up by operating system code, and they almost always run just a single program at one time.

The last Microsoft operating system to allow programs to directly access hardware was MS-DOS, and it was only able to run one program at a time.

**What characteristics distinguish BIOS-level input/output?**

## **1ST ASSEMBLY PROGRAM**

To see how this works, let's look at a simple assembly language program that adds two numbers and saves the result in a register.

```
01 main PROC  
02     mov eax, 5          ;Move 5 into eax register  
03     add eax, 6          ;Add 6 to the contents of eax  
04  
05     INVOKE ExitProcess, 0    ;end the program  
06 main ENDP
```

Also, don't try to type in and run this program just yet—it's missing some important declarations that we will include later on in this chapter.

Line 1 starts the `main` procedure, the entry point for the program.

Line 2 places the integer 5 in the `eax` register.

Line 3 adds 6 to the value in `EAX`, giving it a new value of 11.

Line 5 calls a Windows service (also known as a function) named `ExitProcess` that halts the program and returns control to the operating system.

Line 6 is the ending marker of the `main` procedure.

You probably noticed that we included comments, which always begin with a semicolon character.

We've left out a few declarations at the top of the program that we can show later, but essentially this is a working program.

```
01 .data          ;This is the data area create
02 sum DWORD 0   ;Create a variable called sum
03
04 .code          ;This is the code area/section
05 main PROC
06     mov eax, 5 ;Move 5 into eax register
07     add eax, 6 ;Add 6 to the contents of eax
08
09     INVOKE ExitProcess, 0 ;end the program
10 main ENDP
```

The `sum` variable is declared on Line 2, where we give it a size of 32 bits, using the `DWORD` keyword.

There are a number of these **size keywords**, which work more or less like data types.

But they are not as specific as types you might be familiar with, such as `int`, `double`, `float`, and so on.

They **only specify a size**, but there's no checking into what actually gets put inside the variable.

Remember, **you are in total control**.

By the way, those code and data areas we mentioned, which were marked by the `.code` and `.data` **directives**, are called segments.

So you have the **code segment** and the **data segment**.

Later on, we will see a third segment named **stack**.

## INTEGER LITERALS

An integer literal (also known as an integer constant) is made up of an **optional leading sign**, **one or more digits**, and an **optional radix character** that indicates the number's base:

[ {+ | - } ] digits [ radix ]

So, for example, 26 is a valid integer literal.

It doesn't have a radix, so we assume it's in decimal format.

If we wanted it to be 26 hexadecimal, we would have to write it as 26h.

Similarly, the number 1101 would be considered a decimal value until we added a "b" at the end to make it 1101b (binary).

Here are the possible radix values:

Name	Base	Notation
Decimal	10	d
Hexadecimal	16	h
Octal	8	q/o
Encoded Real	N/A	r
Binary	2	b
Binary (alternate)	2	y

Note that "Encoded Real" does not have a specific base value since it is a format used to represent floating-point numbers in binary.

And here are some integer literals declared with various radices. Each line contains a comment:

```

15 26          ;decimal
16 26d         ;decimal
17 11011011b   ;binary
18 42q         ;octal
19 42o         ;octal
20 1Ah         ;hexadecimal
21 0A3h        ;hexadecimal

```

## HEXADECIMAL BEGINNING WITH A LETTER

A hexadecimal literal beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.

Here's an example of a hexadecimal literal beginning with a letter that requires a leading zero to prevent it from being interpreted as an identifier:

```
mov ax, A123h ; This will cause an "Undefined symbol" error
```

In the above example, the assembler will interpret A123h as an identifier because it begins with the letter "A". To prevent this, we need to add a leading zero to indicate that it is a hexadecimal literal:

```
mov ax, 0A123h ; This will be correctly interpreted as a hexadecimal literal
```

In the corrected example, the **leading zero** tells the assembler that **A123h** should be treated as a hexadecimal number, not an identifier.

## CONSTANT INTEGER EXPRESSION

A constant integer expression is a mathematical expression involving integer literals and arithmetic operators.

Each expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh).

The arithmetic operators are listed in Table 3-1 according to their precedence order, from highest (1) to lowest (4).

Table 3-1 Arithmetic Operators.

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

The important thing to realize about constant integer expressions is that they can only be evaluated at assembly time.

From now on, we will just call them integer expressions.

In assembly language programming, **unary plus** and **minus** are operators that perform arithmetic operations on a single operand.

The **unary plus operator** simply returns the value of its operand, while the **unary minus operator** returns the negative value of its operand.

The reason that the **precedence of unary plus and minus is higher than that of division and multiplication** is due to the rules of operator precedence in most programming languages, including assembly language.

According to these rules, unary plus and minus have the highest precedence, followed by multiplication and division, and then addition and subtraction.

This means that when a mathematical expression contains both unary operators and multiplication or division operators, the unary operators are evaluated first, before the multiplication or division is performed.

For example, consider the following expression:

```
mov eax, -2 * 3
```

In this expression, the unary minus operator has a higher precedence than the multiplication operator.

This means that the expression **-2** is evaluated first, resulting in the value **-2**.

The multiplication is then performed, resulting in the final value of **-6**.

If the multiplication operator had a higher precedence than the unary minus operator, the expression would be evaluated as  $(-2) * 3$ , resulting in a value of -6.

It's worth noting that operator precedence can be overridden using parentheses to group operations in the desired order.

**Operator precedence** refers to the implied order of operations when an expression contains two or more operators.

`4 + 5 * 2`

Multiply, add

`12 -1 MOD 5`

Modulus, subtract

`-5 + 2`

Unary minus, add

`(4 + 2) * 6`

Add, multiply

Use parentheses in expressions to clarify the order of operations so you don't have to remember precedence rules.

In assembly language programming, the **modulus operator** is used to **calculate the remainder of a division operation** between two operands. It is denoted by the percent sign (%), and is also sometimes referred to as the "remainder" operator.

# REAL NUMBER LITERALS

Real number literals (also known as floating-point literals) are represented as either decimal reals or encoded (hexadecimal) reals.

A decimal real contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

[sign]integer.[integer][exponent]

These are the formats for the sign and exponent:

sign {+,-} exponent E[{+,-}] integer

Following are examples of valid decimal reals:

```
24 ;The following are real number literals/floating point literals
25
26 ; [sign]integer.[integer][exponent]
27 ;Exponent- component of the float representing power of the base
28 2.
29 +3.0
30 -44.2E+05
31 26.E5
```

Both 26.E5 and 44.2E05 are valid ways to represent a floating-point number in scientific notation.

The notation [sign]integer.[integer][exponent] is just one convention for representing floating-point numbers in scientific notation, but it's not the only convention.

In the notation 26.E5, the integer component is assumed to be 0. This is because in scientific notation, the significant digits of a number are represented by the digits to the left of the decimal point, while the exponent represents the power of 10 that the number is being multiplied by. So in this case, the significant digits are 26, and the exponent is 5, meaning that the number is equal to  $26 \times 10^5$ , or 2,600,000.

In the notation 44.2E05, the integer component is 44, and the decimal component is 2. The exponent is 5, meaning that the number is equal to  $44.2 \times 10^5$ , or 4,420,000.

So while the notation [sign]integer.[integer][exponent] is a common convention for representing floating-point numbers in scientific notation, it's not necessary to include an integer component before the decimal point if the significant digits of the number don't require it.

### EXAMPLE 1:

An **encoded real** represents a real number in hexadecimal, using the IEEE floating-point format for short reals.

0011 1111 1000 0000 0000 0000 0000 0000

The same value would be encoded as a short real in assembly language as:

3F800000r

We will not be using real-number constants for a while, because most of the x86 instruction set is geared toward integer processing.

However, Chapter 12 will show how to do arithmetic with real numbers, also known as floating-point numbers. It's very interesting, and very technical.

### EXAMPLE 2:

The binary representation of decimal +1.0 in IEEE floating-point format for short reals is:

```
0 01111111 000000000000000000000000
```

In this representation, the first bit is the sign bit (0 for positive), the next 8 bits represent the biased exponent (in this case, 01111111 is the biased exponent for +1.0), and the last 23 bits represent the mantissa (which is 1.0 for this number).

To convert this to hexadecimal, we can group the bits into groups of 4 and convert each group to its hexadecimal equivalent:

```
0011 1111 1100 0000 0000 0000 0000 0000  
3     F     C     0     0     0     0     0
```

So the hexadecimal representation of decimal +1.0 in IEEE floating-point format for short reals is 3FC00000.

## CHARACTER LITERALS

A **character literal** is a single character enclosed in single or double quotes. The assembler stores the value in memory as the character's binary ASCII code. Examples are:

```
35 'A'  
36 "d"
```

Recall that Chapter 1 showed that character literals are stored internally as integers, using the ASCII encoding sequence.

So, when you write the character constant “A,” it’s stored in memory as the number 65 (or 41 hex).

We have a complete table of ASCII codes on the inside front cover of this book, so be sure to look over them from time to time.

## STRING LITERALS

A **string literal** is a sequence of characters (including spaces) enclosed in single or double quotes:

```
38 ;String literals include spaces found inside the quotes
39 'abc'
40 'ABC'
41 "GoodNight, Gracie"
42 '4096'
```

Just as character constants are stored as integers, we can say that string literals are stored in memory as sequences of integer byte values.

So, for example, the string literal “ABCD” contains the four bytes 41h, 42h, 43h, and 44h.

#### **WHY CHARACTERS, STRING LITERALS ETC ARE STORED AS INTEGERS IN A COMPUTER MEMORY?**

Characters and string literals in assembly are stored in memory as integers because all data in a computer's memory is ultimately represented as a sequence of binary digits (bits). In other words, **everything in a computer's memory is represented as a number**.

In order to represent characters and strings in memory, **a standard encoding scheme is used, such as ASCII or Unicode**. These encoding schemes define a mapping between characters and their corresponding numerical codes, which are typically represented as integers.

For example, in ASCII, the letter "A" is represented by the integer value 65, the letter "B" is represented by 66, and so on. String literals in assembly are typically stored as a sequence of these integer values, with a null terminator (a value of 0) at the end to indicate the end of the string.

So while characters and string literals are conceptually different from integers, they are **stored in memory as integers because that is the fundamental representation of all data in a computer's**

memory.

## RESERVED WORDS

Reserved words have special meaning and can only be used in their correct context. Reserved words, by default, are not case-sensitive. For example, MOV is the same as mov and Mov. There are different types of reserved words:

- Instruction mnemonics, such as MOV, ADD, and MUL
- Register names.
- Directives, which tell the assembler how to assemble programs.
- Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD.
- Operators, used in constant expressions.
- Predefined symbols, such as @data, which return constant integer values at assembly time.

## IDENTIFIERS

An **identifier** is a name that a programmer chooses to give to a variable, constant, procedure, or code label in their program. Identifiers can be used to make the code more readable and to help the programmer understand what the different parts of the program do.

Here are the rules for forming identifiers in assembly:

1. **Length:** An identifier can be between 1 and 247 characters long. This means that identifiers can be very long if needed, but it's usually best to keep them relatively short and descriptive.
2. **Case sensitivity:** In assembly, identifiers are not case sensitive. This means that the same

identifier can be written using upper or lowercase letters (or a combination of both), and the assembler will treat them as the same identifier. For example, "myVariable" and "MYVARIABLE" are treated as the same identifier.

**3. First character:** The first character of an identifier must be a letter (A..Z, a..z), underscore (\_), @ , ?, or \$. This means that an identifier cannot start with a number or any other character. For example, "\_myVariable" and "\$myVariable" are both valid identifiers.

**4. Subsequent characters:** After the first character, subsequent characters in an identifier may also be digits (0..9). This means that an identifier can include numbers, but it must start with a letter, underscore, @ , ?, or \$.

**5. An identifier cannot be the same as an assembler reserved word.** This means that certain words are reserved by the assembler for specific purposes, and cannot be used as identifiers. Example: mov, add, sub, and jmp.

Generally, you should avoid the @ symbol and underscore as leading characters, since they are used both by the assembler and by high-level language compilers.

Although assembly language instructions are short and cryptic, there's no reason to make your identifiers hard to understand also! Here are some examples of well-formed names:

```
045 lineCount  
046 firstValue  
047 index  
048 line_count  
049 myFile  
050 xCoord  
051 main  
052 x_Coord
```

```
055 ;The following names are legal, but not as desirable  
056  
057 _lineCount  
058 $first  
059 @myFile
```

## DIRECTIVES

**Directives** in assembly are special instructions that are used to provide information to the assembler, rather than to generate machine code instructions. Directives are typically used to specify data values, allocate memory for variables or arrays, and to control the layout and organization of the generated code.

A directive is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime, but they let you define variables, macros, and procedures.

They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. Directives are not, by default, case sensitive.

For example, `.data`, `.DATA`, and `.Data` are equivalent.

The following example helps to show the difference between **directives** and **instructions**.

The **DWORD directive** tells the assembler to reserve space in the program for a doubleword variable.

The **MOV instruction**, on the other hand, executes at runtime, copying the contents of `myVar` to the `EAX` register:

```
061 myVar DWORD 26
062 mov eax,myVar
```

Although all assemblers for Intel processors share the same instruction set, they usually have different sets of directives. The **Microsoft assembler's REPT directive**, for example, is not recognized by some other assemblers.

**Defining Segments** One important function of assembler directives is to define program sections, or segments. Segments are sections of a program that have different purposes.

For example, one segment can be used to define variables, and is identified by the `.DATA` directive:

Here are some common directives used in assembly programming:

## .data:

One segment can be used to **define variables**, and is identified by the .DATA directive.

This directive is used to specify that the following lines of code will **define data values that need to be stored in memory**.

The .data directive is typically followed by one or more lines that specify the names and initial values of variables. Example:

```
.data  
myVar1 dw 123  
myVar2 db 'hello'
```

## .bss:

This directive is used to specify that the following lines of code will **define uninitialized data** that needs to be stored in memory. The .bss directive is typically followed by one or more lines that specify the names and sizes of variables. Example:

```
.bss  
myArray resb 100
```

## .text:

The .CODE or .TEXT directive identifies the area of a program **containing executable instructions**. This directive is used to specify that the following lines of code will define executable instructions that need to be loaded into memory. The .text directive is typically followed by one or more lines that specify the assembly language instructions to be executed. Example:

```
.text  
main:  
    mov  eax, 1  
    mov  ebx, 0  
    int  0x80
```

.equ:



**equ**

In assembly language, a **symbol** is a label that is used to represent a memory location or a constant value. It can be used to represent a variety of things such as data, instructions, addresses, and offsets.

Symbols are often used to **make code more readable** and easier to understand. They allow programmers to use meaningful names instead of memory addresses or raw numbers.

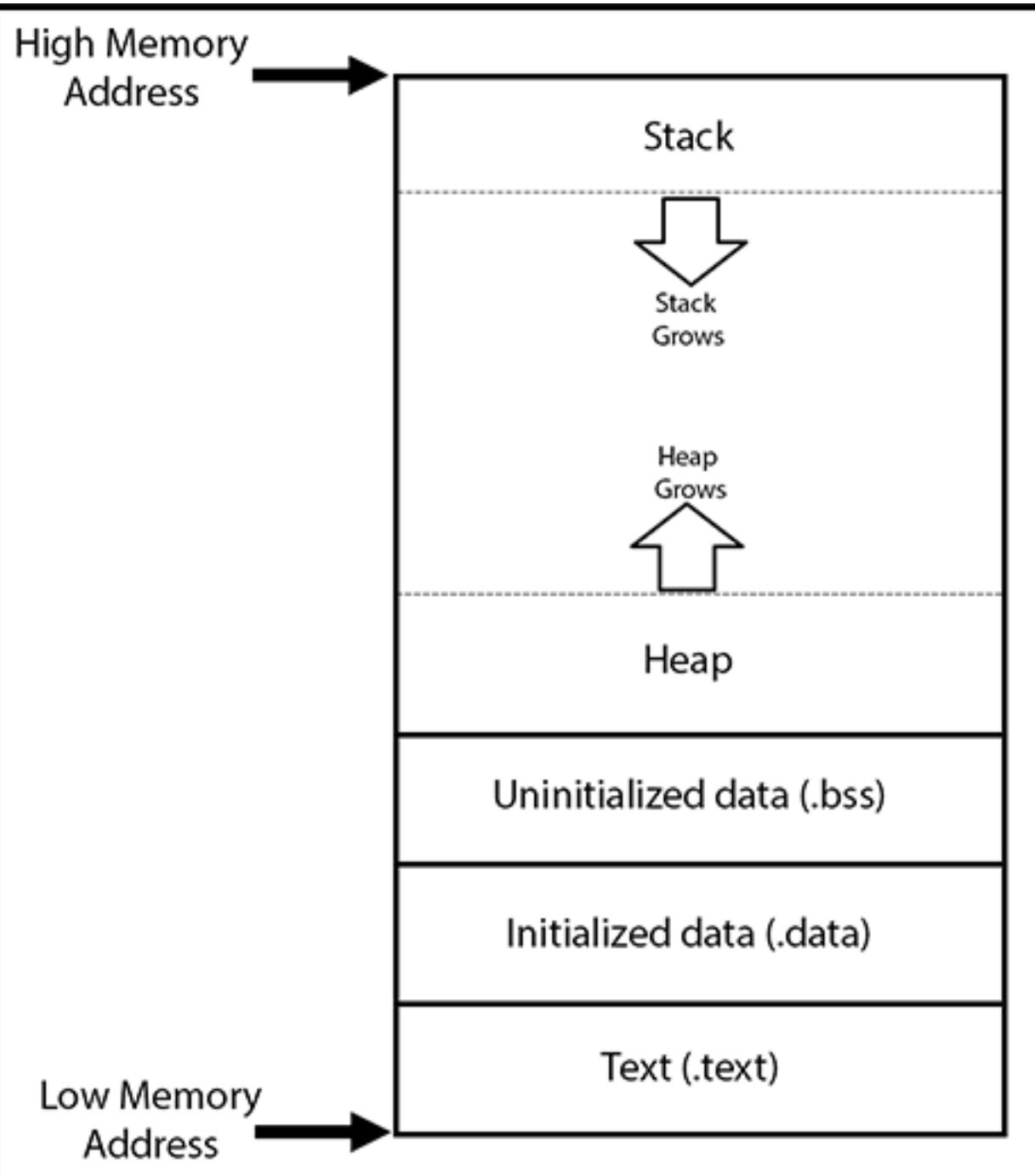
Additionally, symbols can be used to make code more flexible and portable, as they can be easily modified without changing the underlying code.

The **.equ directive** in assembly language is used to define a symbol with a constant value. Once defined, the symbol can be used throughout the code to represent that value. Example:

```
.equ MAX_VALUE, 100  
myVar db MAX
```

## **.stack:**

The .STACK directive identifies the area of a program holding the runtime stack, setting its size.



The runtime stack is a data structure used by a program to manage subroutine calls and local variables.

It is a contiguous area of memory that is dynamically allocated and grows downwards, with the top of the stack being the most recently pushed data.

The .STACK directive in assembly language identifies the starting address and size of the runtime stack.

The syntax of the .STACK directive varies depending on the assembler, but generally it takes one argument, which is the size of the stack in bytes. For example, in NASM (Netwide Assembler), the syntax is as follows:

```
.STACK 100
```

This directive sets the size of the runtime stack to 100 bytes. The assembler will reserve this amount of memory for the stack at runtime, starting from the address specified by the operating system.

It is important to set the correct size of the stack to prevent stack overflow, which occurs when the stack grows beyond its allocated size and overwrites other memory. A stack overflow can cause a program to crash or behave unpredictably.

Here is an example of using the .STACK directive in a simple assembly program:

```
01 ; Set the stack size to 100 bytes
02 .STACK 100
03
04 section .data
05 message db 'Hello, world!', 0
06
07 section .text
08     global _start
09
10 _start:
11
12     push message          ; Push the address of the message string onto the stack
13     call puts              ; Call the puts function to print the message
14     add esp, 4              ; Restore the stack pointer
15     mov eax, 1              ; Exit the program
16     xor ebx, ebx
17     int 0x80
```

## INSTRUCTIONS

**Instruction** is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. Parts of an instruction:

1. **Label (optional)**: A label is an optional identifier that is used to mark a specific memory location or instruction in a program. It is usually followed by a colon (:) and must be unique within the program. Labels are used to provide reference points for jump and branch instructions.

Here is an example of a label in assembly language:

```
loop:  
    ; instructions go here
```

**2. Instruction mnemonic (required):** The instruction mnemonic is the operation code (opcode) that tells the computer what operation to perform. It is a short abbreviation for the operation, such as ADD for addition or MOV for move. Here is an example of an instruction mnemonic in assembly language:

```
mov eax, 0
```

**3. Operand(s) (usually required):** An operand is a value or memory address that is used as an input or output for an instruction. Most instructions require one or more operands. The operands can be constants, registers, memory locations, or labels. Here is an example of an instruction with operands in assembly language:

```
add eax, ebx
```

**4. Comment (optional):** A comment is a note or explanation that is added to the program to provide information to the programmer or to document the code. Comments are ignored by the assembler and do not affect the program execution. Here is an example of a comment in assembly language:

`push message`

; Push the address of the message string onto the stack

## LABEL

**Labels** in assembly language are identifiers that are used to mark specific locations in the program code. They are used to refer to specific memory addresses or instructions within the program. Labels are typically used to **mark the beginning of a subroutine or function**, to **indicate the location of a loop or conditional branch**, or to **mark a specific data location in memory**. A label in the code area of a program (where instructions are located) must end with a colon (:) character.

**MAIN POINT:** In assembly language, labels are typically defined using a colon (:) at the end of the label name, followed by the instruction or data that follows. For example, the label "loop\_start:" might be used to mark the beginning of a loop in the code.

A label is an identifier used to mark a specific location in the program code, so there would need to be an additional line of code that defines a label. Labels are essential in assembly language programming, as they allow programmers to write code that is easy to read and understand, and to create more complex and powerful programs.

A **label** is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

There are two types of labels: **Data labels** and **Code labels**. A **data label identifies the location of a variable**, providing a convenient way to reference the variable in code. The following, for

example, defines a variable named count:

```
myDataSection:  
    count DWORD 100  
    ; other data values here
```

Here's an example of using a label to represent the variable myCount declared as a word (2 bytes) with an initial value of 100 in assembly language.

```
myCount dw 100    ; Declare myCount as a word with initial value of 100  
  
count_label:  
    dw myCount    ; Define a label count_label to represent myCount
```

In this example, the **myCount variable** is declared using the dw directive with an initial value of 100. Then, a label **count\_label** is defined to represent the memory address where myCount is stored.

The **dw directive**

in the **count\_label** assigns the value of myCount to the label count\_label. The label **count\_label** can now be used in place of the variable myCount in any instruction that requires its memory address.

For example, to load the value of myCount into the eax register, you can use the following instruction:

```
mov ax, [count_label] ; Load the value of myCount into ax register
```

Note that the square brackets around `count_label` indicate that the `value` at the memory address represented by `count_label` should be loaded, not the memory address itself.

In assembly language programming, the assembler assigns a numeric address to each label. This address represents the memory location where the label is defined. It is also possible to define multiple data items following a label. The following example illustrates this:

```
array:  
dw 1024, 2048, 3072, 4096 ; Define an array of four 2-byte integers
```

In this example, **the label array defines the memory location of the first number in the array**, which is 1024. The other numbers (2048, 3072, and 4096) immediately follow in memory, each taking up 2 bytes.

Using the label array, you can easily reference any of the numbers in the array by adding an offset to the label's address. For example, to load the second number (2048) into the eax register, you can use the following instruction:

```
mov ax, [array+2] ; Load the second number in the array into ax register
```

In this example, the +2 offset is added to the address of the array label to reference the memory location of the second number in the array.

**Code labels are used as targets of jumping and looping instructions.** For example, the following JMP (jump) instruction transfers control to the location marked by the label named target, creating a loop:

```
01 target:  
02     mov ax,bx  
03     ...  
04     jmp target
```

A **code label** can share the same line with an instruction, or it can be on a line by itself:

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

Label names follow the same rules we described for identifiers. You can use the same code label more than once in a program as long as each label is unique within its enclosing procedure.

## INSTRUCTION MNEMONIC

Instruction mnemonics are a **shorthand notation** used in assembly language programming **to represent machine instructions** in a more readable and human-friendly way.

Each instruction mnemonic corresponds to a specific operation that the processor can perform, such as adding two numbers together or branching to a different part of the program.

Similarly, assembly language instruction mnemonics such as `mov`, `add`, and `sub` provide hints about the type of operation they perform.

Here are some examples of common instruction mnemonics in assembly language:

- **MOV** - move data from one location to another
- **ADD** - add two values together
- **SUB** - subtract one value from another
- **CMP** - compare two values
- **JMP** - jump to a different part of the program
- **CALL** - call a subroutine
- **RET** - return from a subroutine
- **PUSH** - push a value onto the stack
- **POP** - pop a value off the stack

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

In addition to the instruction mnemonic, each instruction in assembly language also includes one or more operands that specify the data to be operated on or the location of the data in memory.

For example, the MOV instruction might have two operands that specify the source and destination of the data being moved.

It's important to note that different processors and architectures may have different sets of instruction mnemonics and operand formats.

As such, it's necessary to consult the documentation for your particular processor or assembly language to determine the correct syntax and usage of each instruction.

## OPERANDS

The **opcode** is a short code that identifies the specific operation that the processor should perform, such as add, subtract, or move data. The **operand(s)** specify the data on which the operation should be performed, such as registers, memory addresses, or immediate values.

```
mov ax, bx
```

The **opcode** is **mov**, which instructs the processor to move data, and the **operands** are **ax** and **bx**, which specify the registers containing the data that should be moved. Here's another example:

```
add eax, [ebx+8]
```

In this instruction, the **opcode** is **add**, which instructs the processor to add two values. The **operands** are **eax** and **[ebx+8]**. **eax** specifies the register that will hold the result of the addition operation, while **[ebx+8]** specifies the memory location where the second value to be added is stored. The **+8 offset** specifies that the value is stored 8 bytes beyond the memory address contained in the **ebx** register.

In summary, the **opcode specifies the operation to be performed, while the operand(s) provide the data to be used in that operation. Together, they form the complete instruction that is executed by the processor.**

Assembly language instructions can have **between zero and three operands**, each of which can be a register, memory operand, integer expression, or input-output port. An **operand** is a value that an assembly language instruction uses as input or output.

To create memory operands, you can use variable names or registers surrounded by brackets. When you

use a variable name as an operand, it tells the computer to access the contents of memory at the specified address.

EG

```
mov eax, [my_var]
```

In this instruction, `my_var` is a **variable name used as an operand**. The **brackets surrounding `my_var` indicate that** the contents of memory at the address specified by `my_var` should be loaded into the `eax` register. The assembler will replace `my_var` with the appropriate memory address during the assembly process. Similarly, you can use **registers surrounded by brackets** to create memory operands. For example, consider the following instruction:

```
mov [ebx], ecx
```

In this instruction, `[ebx]` is a memory operand. The brackets indicate that the contents of memory at the address stored in the `ebx` register should be updated with the value in the `ecx` register. Using memory operands allows programmers to read from and write to specific memory locations, enabling the manipulation of data stored in memory.

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

Let's look at examples of assembly language instructions having varying numbers of operands. The STC instruction, for example, has no operands:

```

01 stc          ;set carry flag
02 inc eax      ;add 1 to eax
03 mov count, eax ;move ebx to count
04

```

There is a natural ordering of operands. When instructions have multiple operands, the first one is typically called the **destination operand**. The second operand is usually called the **source operand**.

In general, the contents of the destination operand are modified by the instruction. In a **MOV instruction**, for example, data is copied from the **source to the destination**.

The IMUL instruction has three operands, in which the first operand is the destination, and the following two operands are source operands, which are multiplied together:

```
01 stc           ;set carry flag  
02 inc eax       ;add 1 to eax  
03 mov count, eax ;move ebx to count  
04 imul eax,ebx,5
```

In this case, EBX is multiplied by 5, and the product is stored in the EAX register.

## COMMENTS

Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code.

The following information is typically included at the top of a program listing:

- Description of the program's purpose
  - Names of persons who created and/or revised the program
  - Program creation and revision dates
  - Technical notes about the program's implementation
- Comments can be specified in two ways:
- Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.
  - Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. Here is an example:

For block comments, just use semi-colon(;).

## NOP OPERATION(NO OPERATION)

In assembly language programming, the NOP (no operation) instruction is an instruction that performs no operation when executed. It is a placeholder instruction that is often used for padding or alignment purposes, especially in cases where it is necessary to align code to specific memory boundaries.

The NOP instruction takes up one byte of program storage and doesn't perform any work when executed. The instruction is useful in situations where it is necessary to maintain the length of the instruction stream, but no actual operation needs to be performed. This can happen, for example, when aligning code to specific memory boundaries that require instruction alignment.

In the following example, the first MOV instruction generates three machine code bytes. The NOP instruction aligns the address of the third instruction to a doubleword boundary (even multiple of 4):

```
01 00000000 66 8B C3 mov ax, bx
02 00000003 90 nop ;align next instruction
03 00000004 8B D1    mov edx, ecx
```

OR

```
mov eax, [ebx]
mov ebx, [ecx]
nop
```

In this example, the first two instructions move data between registers. The third instruction is a NOP instruction that does nothing, but it ensures that the address of the next instruction is aligned to a doubleword boundary. This alignment can improve the performance of the code by allowing the processor to access memory more efficiently.

In summary, the NOP instruction is a placeholder instruction that is often used for padding or alignment purposes. It takes up one byte of program storage and doesn't perform any work when executed. It is sometimes used by compilers and assemblers to align code to efficient memory boundaries, which can improve performance.

## x86 PROCESSORS AND SPEED

x86 processors are designed to load code and data more quickly from even doubleword (4-byte) addresses. This is because the x86 architecture uses a memory bus that transfers data in units of 4 bytes, and loading data from even doubleword addresses ensures that the data can be transferred in a single bus cycle.

When data is loaded from an address that is not aligned to a doubleword boundary, the processor must perform two memory accesses to fetch the entire 4-byte value. This can slow down memory access and reduce the performance of the program.

To avoid this performance penalty, programmers can use techniques such as code padding or NOP instructions to ensure that code and data are aligned to doubleword boundaries. This can improve the performance of the program by allowing the processor to load and process data more efficiently.

In summary, x86 processors are designed to load code and data more quickly from even doubleword addresses, and programmers can improve performance by ensuring that code and data are aligned to these boundaries.

## QUESTIONS AND ANSWERS

Write the real number  $-6.2 \times 10^4$  as a real number literal using MASM syntax.

**-6.2e4**

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

Must string literals be enclosed in single quotes?

In MASM syntax, string literals must be enclosed in double quotes (" "). Single quotes (' ') are used to denote character literals.

For example, to define a string literal "Hello, World!" in MASM syntax, we would write:

arduino

 Copy code

```
myString DB "Hello, World!",0
```

Here, "myString" is the label for the string, "DB" stands for "define byte", and the string is enclosed in double quotes.

If we were to enclose the string in single quotes, like this:

arduino

 Copy code

```
myString DB 'Hello, World!',0
```

This would cause a syntax error in MASM, because single quotes are not used to define strings in MASM syntax.

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

Reserved words can be instruction mnemonics, attributes, operators, predefined symbols and \_\_\_\_\_

Reserved words can also be assembler directives, which are special instructions to the assembler that tell it how to process the code. Assembler directives are not executed as part of the program, but rather provide information to the assembler about how to assemble the program.

Examples of assembler directives include:

- .DATA: Defines a section of memory for storing initialized data
- .CODE: Defines a section of memory for storing program code
- .MODEL: Specifies the memory model to use for the program
- .STACK: Specifies the size of the program stack
- .INCLUDE: Includes the contents of another file in the current assembly

As with other reserved words, using an assembler directive as a label or variable name can cause errors or unexpected behavior during assembly.

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

What is the maximum length of an identifier?

In MASM syntax, the maximum length of an identifier is 247 characters. However, it is recommended to keep identifier names shorter for readability and maintainability.

It is also important to note that the first character of an identifier must be a letter, underscore, or special character (\$ or @), and subsequent characters can be letters, digits, underscores, or special characters. Additionally, identifiers are case-insensitive, meaning that "myVariable" and "MYVARIABLE" are treated as the same identifier.

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

Is A5h a valid hexadecimal literal? NO, Use 0A5h

Yes, A5h is a valid hexadecimal literal. In MASM syntax, hexadecimal literals are represented with the prefix "0x" or "0X" followed by one or more hexadecimal digits (0-9, A-F). However, in some other assemblers or programming languages, the "h" suffix is used to indicate a hexadecimal literal. Therefore, A5h can be interpreted as a hexadecimal literal in such contexts.

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

Using the value -35, write it as an integer literal in decimal, hexadecimal, octal, and binary formats that are consistent with MASM syntax.

Here are the integer literals in decimal, hexadecimal, octal, and binary formats that are consistent with MASM syntax for the value -35:

- Decimal: -35
- Hexadecimal: 0xFFFFFFFDD
- Octal: 3777777643
- Binary: 0b11111111111111111111111011101

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

**Explain why and what is hex with an example 05h, and why we also use ASCII?**

IN ASSEMBLY LANGUAGE, I FIND IT HARD TO UNDERSTAND WHAT SOMETHING LIKE mov eax, 05H means. that hex part is really confusing. Like the same to ascii, eg ) represented by a certain hex. I know ascii is a mapping of keyboard characters to respective numbers coz computers can only read numbers.

I can understand how hexadecimal notation can be confusing at first, especially if you're not used to working with it. Let me try to explain it in a way that might help.

In assembly language, `mov eax, 05h` means "move the value 5 (in hexadecimal notation) into the EAX register". The "h" at the end indicates that the number is written in hexadecimal notation. Hexadecimal notation is a base-16 number system, which means that it uses 16 digits instead of the 10 digits used in the decimal (base-10) number system. The digits in hexadecimal notation are 0-9 and A-F, where A represents the decimal value of 10, B represents 11, and so on up to F which represents the decimal value of 15.

So in the example `mov eax, 05h`, the value 5 in hexadecimal notation is equivalent to the decimal value 5. Similarly, the value FFh in hexadecimal notation is equivalent to the decimal value 255. In assembly language, hexadecimal notation is often used because it is more compact and easier to read than binary notation.

Regarding **ASCII**, you're correct that it is a mapping of keyboard characters to respective numbers. Each character in the ASCII table is assigned a unique number, or ASCII code, which is represented in hexadecimal notation. For example, the ASCII code for the character ")" is 29h in hexadecimal notation. When you want to display this character on a computer screen, the computer looks up the ASCII code for ")" and displays the corresponding character.

\*\*\*\*\*

NEXT QUESTION

\*\*\*\*\*

## ADDING AND SUBTRACTING

```
1 ;addTwo.asm - adds two 32-bit integers
2
3 .386           ;identifies this is a 32-bit program
4 .model flat, stdcall ;selects the program's memory model (flat)/calling convention for procedures
5 .stack 4096      ;sets aside 4096 bytes of storage for the runtime stack,every program must have.
6 ExitProcess PROTO, dwExitCode:DWORD
7
8 .code
9 main PROC
10    mov eax,5          ;move 5 to the EAX register
11    add eax,6          ;add 6 to the EAX register
12
13    INVOKE ExitProcess, 0
14 main ENDP
15 END main
```

Line 4 contains the **.386 directive**, which identifies this as a 32-bit program that can access 32-bit registers and addresses.

Line 5 **selects the program's memory model (flat)**, and identifies the calling convention (named stdcall) for procedures. The "flat" memory model is a memory model where all memory is treated as a single, contiguous address space. In a flat memory model, the program sees all memory locations as a single, linear address space, regardless of the physical organization of the memory. This means that the program can access any memory location directly, without the need for special instructions or addressing modes. We use this because 32-bit Windows services require the stdcall convention to be used. (Chapter 8 explains how stdcall works.)

Line 6 **sets aside 4096 bytes of storage** for the runtime stack, which every program must have.

bash

 Copy code

```
ml /c /coff yourfile.asm
```

This command will compile your assembly code and generate an object file with the ".obj" extension.

4. Link your object file: Once you have compiled your assembly code, you need to link it to create an executable file. You can use the following command to link your object file:

bash

 Copy code

```
link /subsystem:console yourfile.obj
```

In line 7 of the code, a **prototype** for the `ExitProcess` function is declared. This function is a standard Windows service that is used to exit a program. A **prototype** is a declaration of the function's name and input parameters, which allows the assembler to generate correct machine code for the function call.

The input parameter for `ExitProcess` is called `dwExitCode`, which is like a return value that is

passed back to the operating system. A return value of zero indicates that the program was successful, while any other integer value generally indicates an error code number. This helps the operating system to understand the outcome of the program's execution.

You can think of your assembly programs as subroutines or processes that are called by the operating system. When your program is ready to finish, it calls the `ExitProcess` function and returns an integer that tells the operating system about the program's success or failure. This way, the operating system can manage the execution of multiple programs and handle any errors that might occur.

In the context of the previous explanation, system administrators often create script files that execute a series of programs in sequence. At each point in the script, they need to know if the most recently executed program has failed or succeeded, so they can determine the next step to take.

To achieve this, the operating system provides a mechanism for programs to indicate whether they completed successfully or not. This is typically done by returning an integer value from the program, where a value of 0 indicates success, and any non-zero value indicates an error code.

In a script file, system administrators can use the `ErrorLevel` variable to check the return code of the previous command or program. For example, if a program returns a non-zero value indicating an error, the `ErrorLevel` variable will be set to 1, and the script can be set up to exit or take appropriate action based on this value.

In summary, by using the return value mechanism provided by the operating system, programs can indicate whether they completed successfully or not, and system administrators can use this information to automate and manage the execution of a sequence of programs in a script file.

## CODE EXPLAINED

`.MODEL directive` tells the assembler which memory model to use. In 32-bit programs, we always use

the flat memory model, which is associated with the processor's protected mode. The **stdcall keyword** tells the assembler how to manage the runtime stack when procedures are called. That's a complicated subject that we will address in Chapter 8.

In Assembly language, the **ExitProcess symbol** is usually defined using the **PROTO directive**, which specifies the function prototype. The ExitProcess function is a Win32 API function that terminates the calling process and returns the specified exit code to the operating system.

The **PROTO directive** is used to declare the function prototype and its parameter list. In this case, ExitProcess PROTO, dwExitCode:DWORD indicates that ExitProcess is a function that takes one parameter, dwExitCode, which is a 32-bit unsigned integer.

Next, the **.STACK directive** tells the assembler how many bytes of memory to reserve for the program's runtime stack. The value 4096 is probably more than we will ever use, but it happens to correspond to the size of a memory page in the processor's system for managing memory.

All modern programs use a **stack** when calling subroutines—first, to **hold passed parameters**, and second, to **hold the address of the code that called the function**.

\*\*\*\*\*

Let's see the stack example in C:

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
```

```
    return n * factorial(n - 1);
}

int main() {
    int result = factorial(5);
    printf("The factorial of 5 is %d\n", result);
    return 0;
}
```

When the `main()` function calls the `factorial()` function, the following steps occur:

1. The stack pointer is decremented to make room for the return address.
2. The return address is pushed onto the stack.
3. The parameters to the `factorial()` function are pushed onto the stack.
4. The program counter is set to the address of the first instruction in the `factorial()` function.
5. The `factorial()` function executes.
6. When the `factorial()` function returns, the following steps occur:
  - The parameters to the `factorial()` function are popped off the stack.
  - The return address is popped off the stack and used to set the program counter.
  - The stack pointer is incremented to restore the original stack frame.

The following diagram shows the stack before and after the `main()` function calls the `factorial()` function:

Stack before calling `factorial()`:

| Return address |

```
| Parameter 1      |
| Parameter 2      |
| .....           |
```

Stack after calling factorial():

```
| Return address for factorial()   |
| Parameter 1 for factorial()       |
| Parameter 2 for factorial()       |
| Return address for main()        |
| Parameter 1 for main()           |
| Parameter 2 for main()           |
| .....                           |
```

In addition to holding passed parameters and the return address, the stack can also be used to store local variables. Local variables are variables that are declared inside a function.

When a function is called, a new stack frame is created to store the local variables for that function call. When the function returns, the stack frame is popped off the stack and the local variables are destroyed.

Here is an example of how to use the stack to store local variables:

```
#include <stdio.h>

int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
```

```
    result *= i;
}
return result;
}

int main() {
    int result = factorial(5);
    printf("The factorial of 5 is %d\n", result);
    return 0;
}
```

In this example, the `factorial()` function declares a local variable called `result`. This variable is stored on the stack while the `factorial()` function is executing.

When the `factorial()` function returns, the variable `result` is destroyed.

Stack usage is an important part of C programming. By understanding how the stack works, you can write more efficient and reliable code.

\*\*\*\*\*

The CPU uses this address to return when the function call finishes, back to the spot where the function was called. In addition, the runtime stack can hold local variables, that is, variables declared inside a function.

The `.CODE` directive marks the beginning of the code area of a program, the area that contains executable instructions. Usually the next line after `.CODE` is the declaration of the program's entry point, and by convention, it is usually a procedure named `main`. The entry point of a program is the location of the very first instruction the program will execute. We used the following lines to convey this information:

The **ENDP directive** marks the end of a procedure. Our program had a procedure named `main`, so the `endp` must use the same name.

Finally, the **END directive** marks the end of the program, and references the program entry point:

```
.code  
main PROC  
    ; Your code goes here  
main ENDP  
END main
```

The "**END**" directive in assembly language is a marker that indicates where the program ends, and the "**\_start**" or "**main**" label specifies the entry point where the program execution begins.

The name used with the "**END**" directive can be any valid label or symbol in your assembly code. It does not have to be "`main`" or "`_start`". You can use any label or symbol that refers to the location of the last instruction in your code.

```
section .text
global AddTwoFunction

AddTwoFunction:
    ; program code goes here
    ret

end AddTwoFunction
```

The END directive, on the other hand, marks the end of the entire program, not just the "main" procedure. It typically appears at the end of the source file and **references the program entry point**, which is typically the "main" function in a C or C++ program.

So, in summary, the "main ENDP" directive marks the end of the "main" procedure, while the "END main" directive marks the end of the entire program and references the program entry point, which is typically the "main" function. While the names are the same, they refer to different entities in the program.

**NB:**

If you add any more lines to a program after the END directive, they will be ignored by the assembler. You can put anything there—program comments, copies of your code, etc.—it doesn't matter.

## RUNNING AND DEBUGGING

It ran successfully:

The screenshot shows the Microsoft Visual Studio interface with a dark theme. The main window displays an assembly code file named 'AddTwo.asm' containing the following code:

```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    mov eax, 5 ;move 5 to the eax register
    add eax, 6 ;add 6 to the eax register

    INVOKE ExitProcess, 0
main ENDP
END main
```

To the right of the code editor is the Solution Explorer, which shows the project structure:

- AddTwo1 (C:\VS-Assembly\AddTwo1)
- Debug
- x64
- AddTwo.asm
- AddTwo1.sln
- AddTwo1.vcxproj
- AddTwo1.vcxproj.filters
- AddTwo1.vcxproj.user

Below the code editor is the Microsoft Visual Studio Debug Console, which outputs:

```
C:\VS-Assembly\AddTwo1\Debug\AddTwo1.exe (process 16168) exited with code 0.
Press any key to close this window . . .
```

The bottom left corner shows the Output window with the message "No issues found".

Just make sure the project is linked with masm and the files you run are in C drive together with masm.

Another way to start a debugging session is to **set a breakpoint** on a program statement by clicking the mouse in the vertical gray bar just to the left of the code window. A large red dot will mark the breakpoint location. Then you can run the program by selecting Start Debugging from the Debug menu.

**Tip:** If you try to set a breakpoint on a non-executable line, Visual Studio will just move the breakpoint forward to the next executable line when you run the program.

Memory window, registers etc have been removed in VS Community 2022.

```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    mov eax, 5 ;move 5 to theeax register
    add eax, 6 ;add 6 to theeax register

    INVOKE ExitProcess, 0
main ENDP
END main
```

146 % C:\VS-Assembly\AddTwo1\Debug\AddTwo1.exe

Autos

Search (Ctrl+E)

Figure 3-3 shows the program at the start of a debugging session.

A breakpoint was set on Line 11, the first MOV instruction, and the debugger has paused on that

line.

The line has not executed yet.

When the debugger is active, the bottom status line of the Visual Studio window turns orange.

When you stop the debugger and return to edit mode, the status line turns blue.

The visual cue is helpful because you cannot edit or save a program while the debugger is running.

Figure 3-4 shows the debugger after the user has stepped through lines 11 and 12, and is paused on line 14.

By hovering the mouse over the EAX register name, we can see its current contents (11).

The screenshot shows the Microsoft Visual Studio interface during assembly language debugging. The main window displays the assembly code for 'AddTwo.asm' under '(Global Scope)'. The code is as follows:

```
8 .code
9
10 main proc
11     mov eax,5
12     add eax,6
13     ; eax = 11
14     invoke ExitProcess,0
15 main endp
16 end main
```

The instruction at line 13, 'add eax,6', has a tooltip 'eax = 11' displayed above it. Below the code editor, the 'Call Stack' window is open, showing the current call stack frames:

Name	Language
Project.exe!main() Line 14	Unknown
ntdll.dll!76ff377b()	Unknown
[Frames below may be incorrect]	
ntdll.dll!76ff374e()	Unknown

At the bottom of the interface, the status bar shows 'Ln 14' and 'Col 1'. The toolbar contains various icons for file operations, search, and debugging.

We can then finish the program execution by clicking the Continue button on the toolbar, or by clicking the red Stop Debugging button (on the right side of the toolbar).

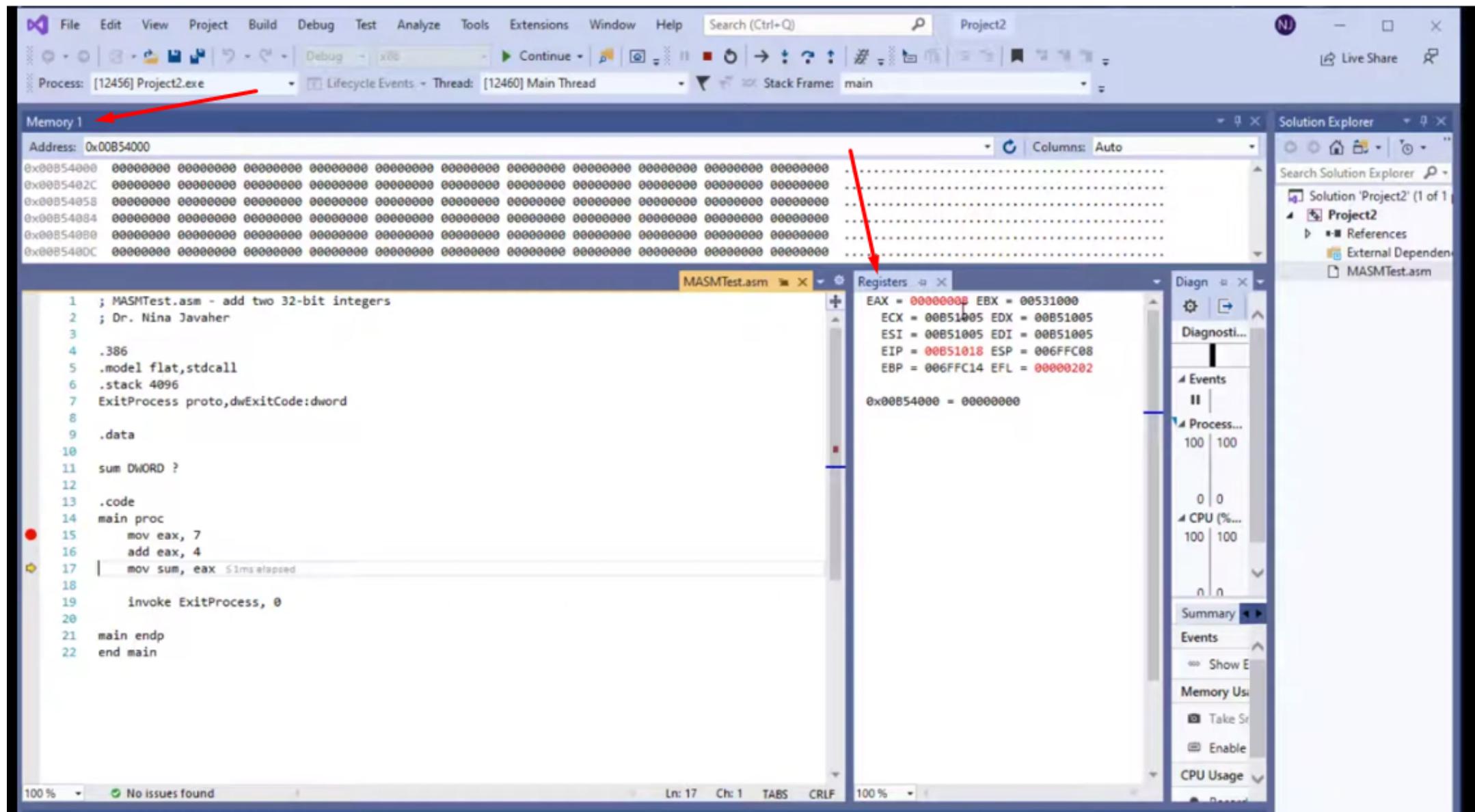
clicking the red Stop Debugging button (on the right side of the toolbar).

## Customizing the Debugging

You can customize the debugging interface while it is running. For example, you might want to display the CPU registers; to do this, select Windows from the Debug menu, and then select Registers.

Figure 3-5 shows the same debugging session we used just now, with the Registers window visible. We also closed some other nonessential windows. The value shown in EAX, 0000000B, is the hexadecimal representation of 11 decimal.

Works in old VS Community old versions only.



In the Registers window, the EFL register contains all the status flag settings (Zero, Carry, Overflow, etc.). If you right-click the Registers window and select Flags from the popup menu, the

window will display the individual flag values. Figure 3-6 shows an example, where the flag values from left to right are: OV (overflow flag), UP (direction flag), EI (interrupt flag), PL (sign flag), ZR (zero flag), AC (auxiliary carry), PE (parity flag), and CY (carry flag).

The precise meaning of these flags will be explained in Chapter 4. One of the great things about the Registers window is that as you step through a program, any register whose value is changed by the current instruction will turn red. Although we cannot show it on the printed page (which is black and white), the red highlighting really jumps out at you, to let you know how your program is affecting the registers.

Project (Debugging) - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW

HELP

Process: [9740] Project.exe Suspend Thread: [4712] Main Thread

AddTwo.asm

(Global Scope)

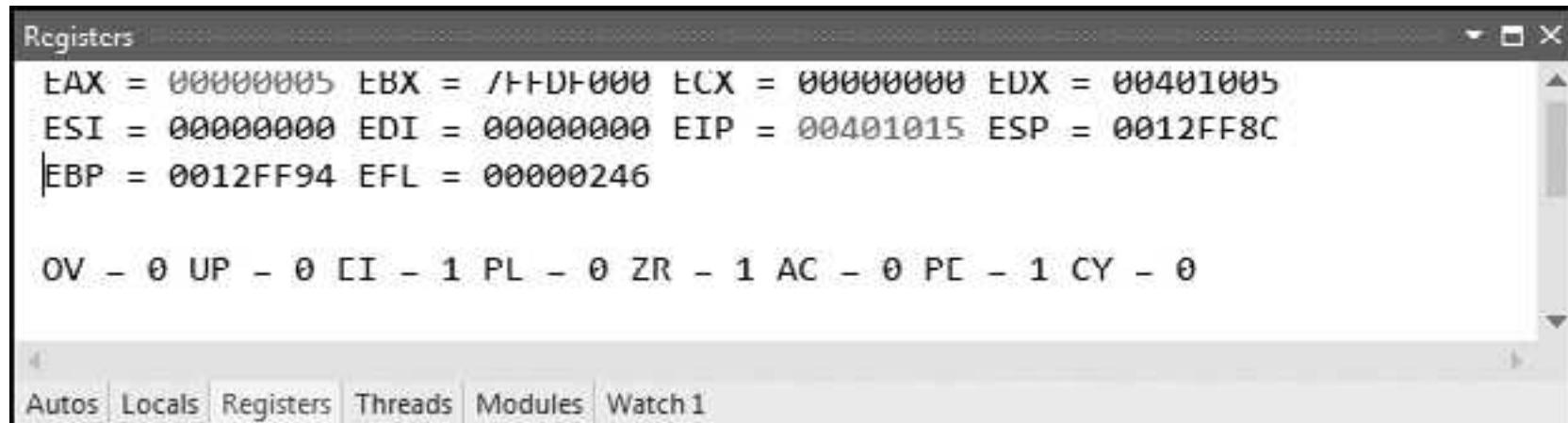
```
8
9 .code
10 main proc
11     mov eax,5
12     add eax,6
13
14     invoke ExitProcess,0
15 main endp
16 end main
```

Registers

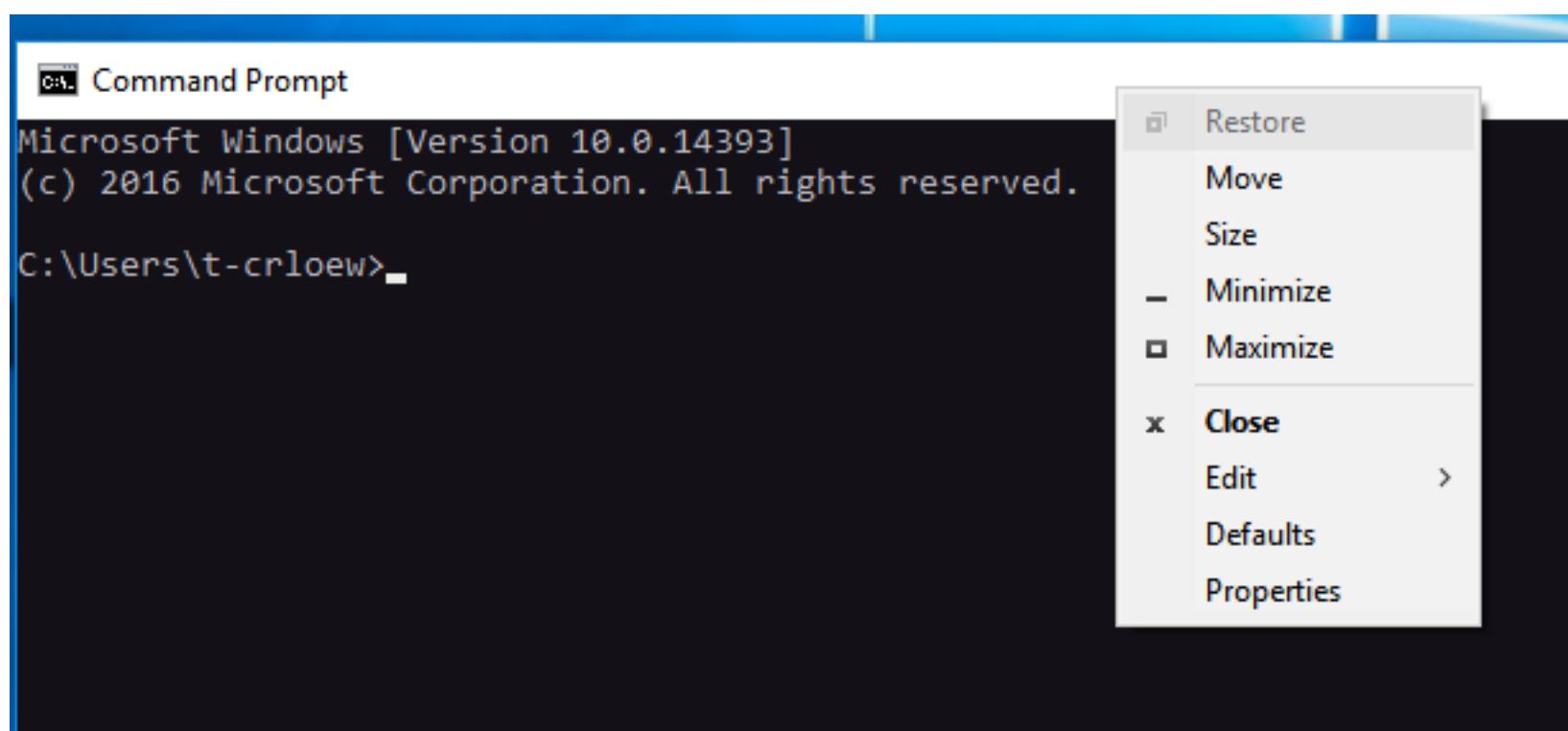
EAX = 0000000B EBX = 7FFD8000 ECX = 00000000 EDX = 00401005  
ESI = 00000000 EDI = 00000000 EIP = 00101018 ESP = 0012FF8C  
EBP = 0012FF94 EFL = 00000202

Autos Locals Registers Threads Modules Watch1

Ready Ln14 Col1 Ch1 INS



When you run an assembly language program inside Visual Studio, it launches inside a console window.



This is the same window you see when you run the program named cmd.exe from the Windows Start menu. Alternatively, you could open up a command prompt in the project's Debug\Bin folder and run the application directly from the command line.

If you did this, you would only see the program's output, which consists of text written to the console window. Look for an executable filename having the same name as your Visual Studio project.

### 3.2.3 Program Template Assembly language programs have a simple structure, with small variations.

When you begin a new program, it helps to start with an empty shell program with all basic elements in place. You can avoid redundant typing by filling in the missing parts and saving the file under a new name.



The following program (Template.asm) can easily be customized. Note that comments have been inserted, marking the points where your own code should be added. Capitalization of keywords is

optional:

```
01 ;Program template for almost all assembly programs
02 ;doesn't work in emu8086 because of the .386(32-bit)
03
04 .386
05 .model flat, stdcall
06 .stack 4096
07 ExitProcess PROTO, dwExitCode:DWORD
08
09 .data
10     ;declare initialised data/variables here
11
12 .code
13     main PROC
14         ;write your code here
15
16         INVOKE ExitProcess, 0
17
18 main ENDP
19 END main
```

**Use Comments** - It's a very good idea to include a program description, the name of the program's author, creation date, and information about subsequent modifications. Documentation of this kind is

useful to anyone who reads the program listing (including you, months or years from now).



**Many programmers have discovered, years after writing a program, that they must become reacquainted with their own code before they can modify it. If you're taking a programming course, your instructor may insist on additional information.**

In assembly language, the **INCLUDE directive** is used to include the contents of an external file into the source code being assembled.

# INCLUDE

```
INCLUDE filename
```

Here, `filename` is the name of the file to be included. The file may have an extension specific to the assembler being used, such as `.inc` or `.asm`. For example, suppose you have a file named `my_macros.asm` that contains some commonly used macros. You can include this file in your main assembly file using the `INCLUDE` directive like this:

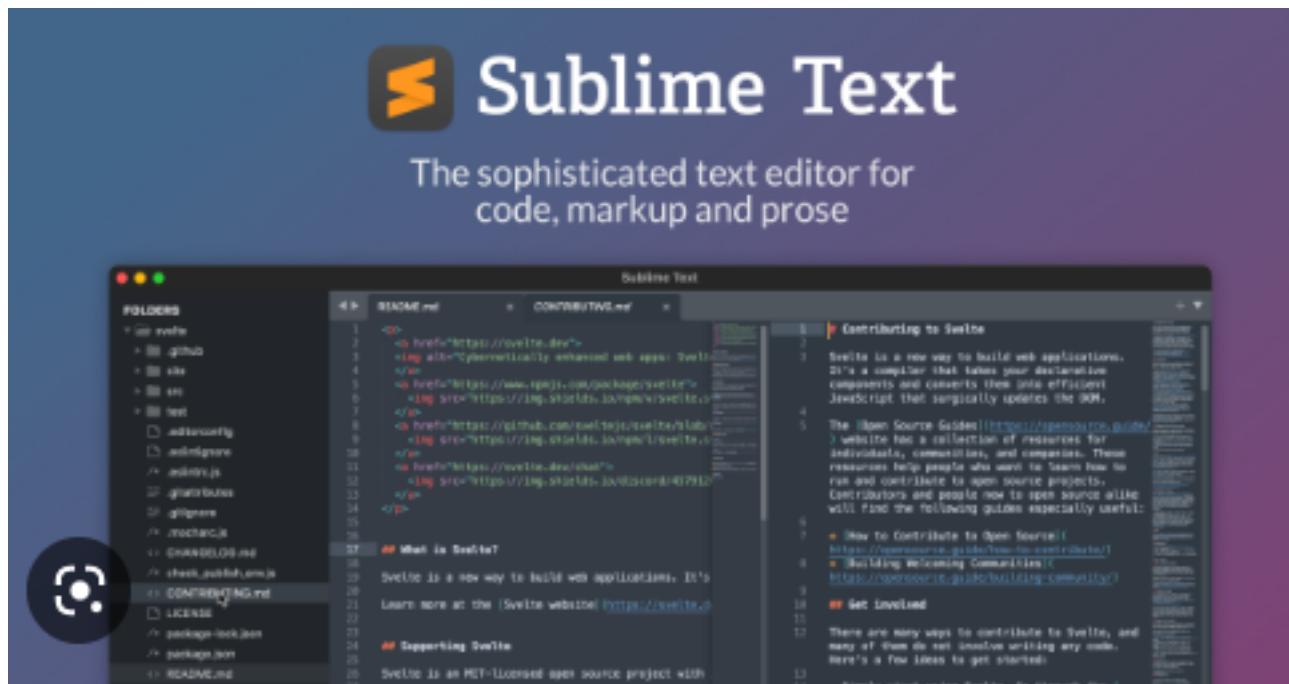
```
INCLUDE my_macros.asm
```

## ASSEMBLE, LINKING AND RUNNING PROGRAMS

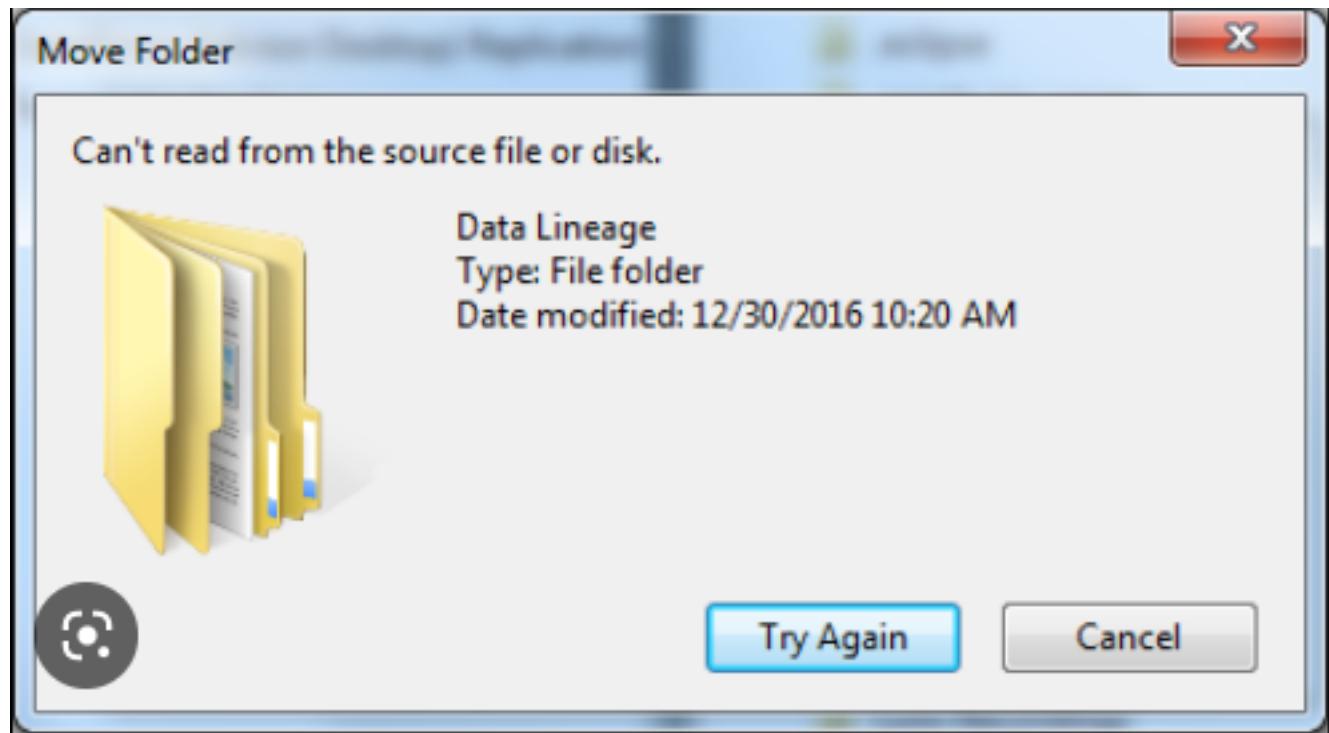
- A source program written in assembly language cannot be executed directly on its target computer.
- It must be translated or assembled into executable code using an assembler.

- The assembler produces an **object file**, which contains machine language.
- The object file must be passed to a linker which produces an executable file that is ready to execute from the operating system's command prompt.
- The process of editing, assembling, linking, and executing assembly language programs involves the following steps:

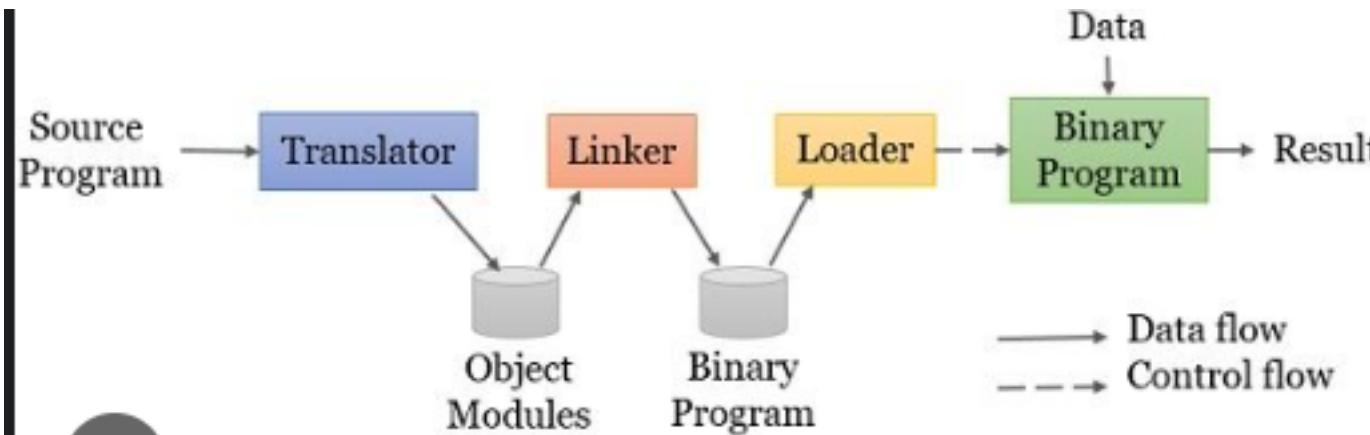
**Step 1:** The programmer uses a text editor to create an ASCII text file named the source file.



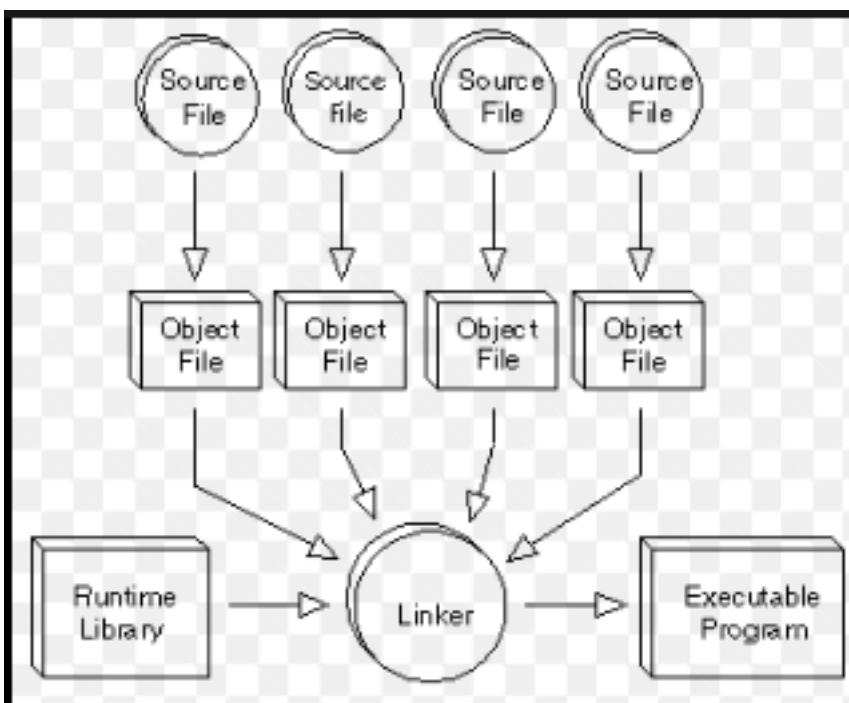
**Step 2:** The assembler reads the source file and produces an object file, optionally producing a listing file. If any errors occur, the programmer must return to Step 1 and fix the program.



**Step 3:** The linker reads the object file, checks for calls to procedures in a link library, copies required procedures from the library, combines them with the object file, and produces the executable file.

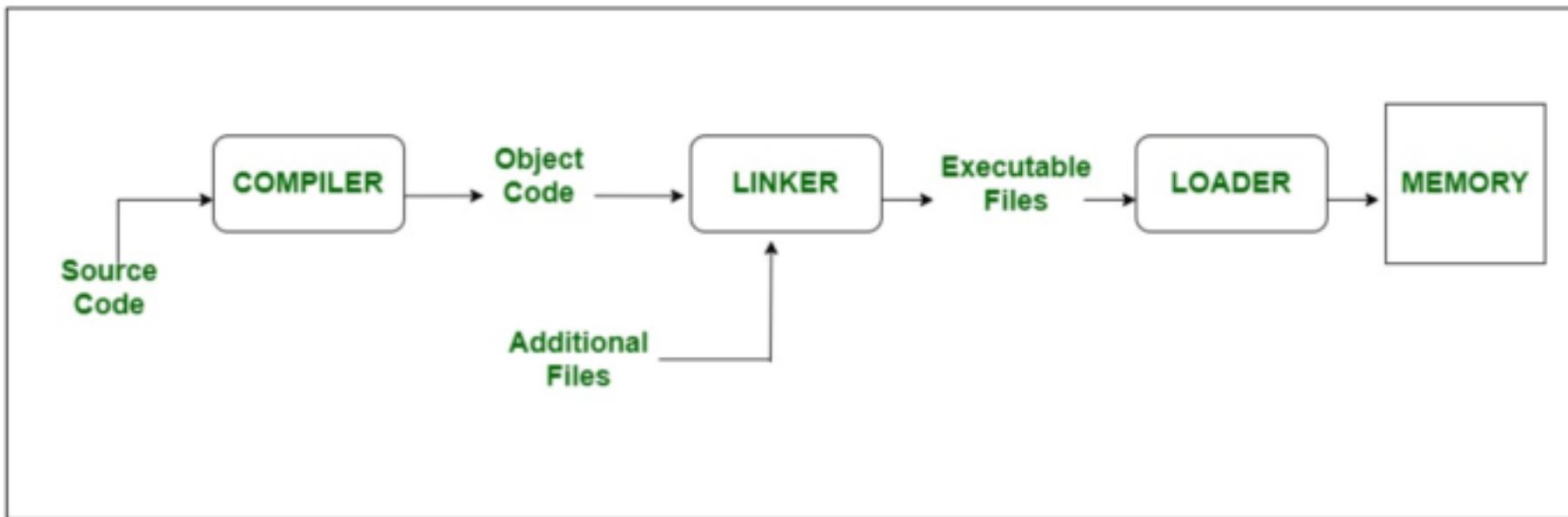


### Schematic Execution of the Program



**Step 4:** The operating system loader utility reads the executable file into memory, branches the CPU

to the program's starting address, and the program begins to execute.



## LISTING FILES AND SYMBOL TABLES

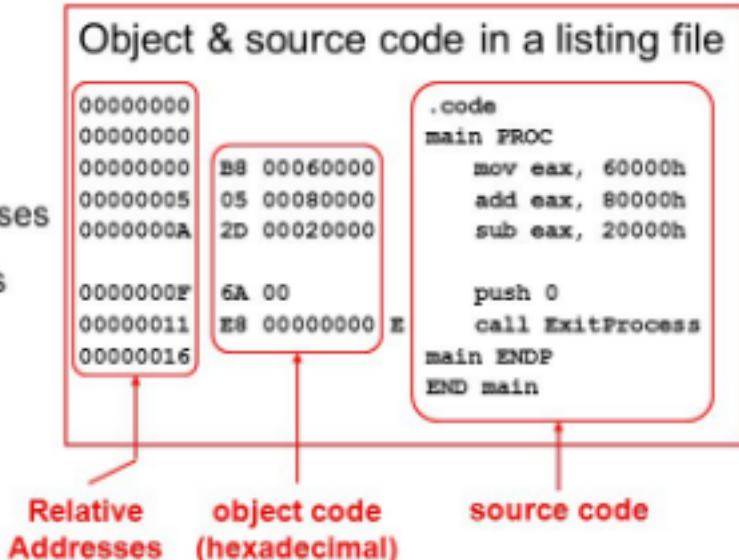
A **listing file** contains a copy of the program's source code, with line numbers, the numeric address of each instruction, the machine code bytes of each instruction (in hexadecimal), and a symbol table.

# List File

- ❖ Use it to see how your program is assembled

- ❖ Contains

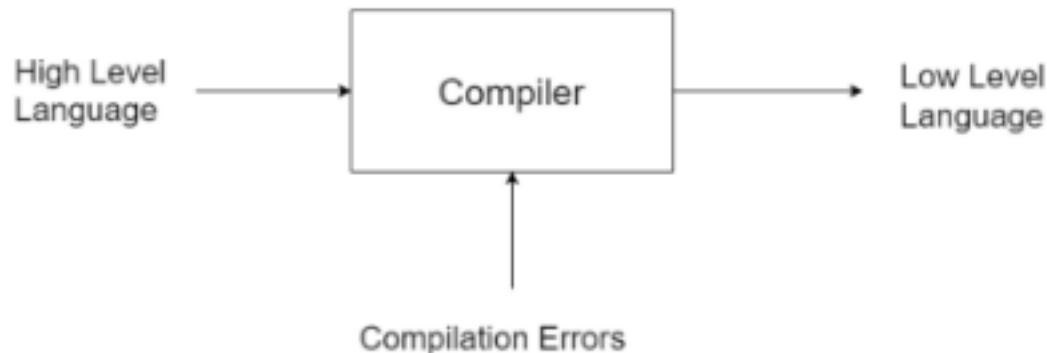
- ❖ Source code
- ❖ Object code
- ❖ Relative addresses
- ❖ Segment names
- ❖ Symbols
  - Variables
  - Procedures
  - Constants



The **symbol table** contains the names of all program identifiers, segments, and related information. Advanced programmers sometimes use the listing file to get detailed information about the program.



# Symbol Tables in Compiler Design



**MOV AL,0001H** means load the AL register with the contents of memory location 0001H. That is, if you stored the **pay rate in memory location 0001H**, then the next time you make use of the pay rate it should be fetched from memory location 0001H, but if you had used memory location 0002H instead of 0001H then that would be fine also - as long as you always used memory location 0002H when you wanted to use the pay rate.

For most tasks the exact memory location that you use to store some data is irrelevant - as long as it is always used consistently. In the early days programmers would have to start their programs by performing manual memory allocation. that is they first assigned uses to particular memory locations - 001 will be the total, 002 the running count, 003 a temporary result and so on.

Then as they programmed the action of the program they used the addresses that had been assigned in instructions. Clearly trying to remember where you stored everything isn't fun and it's very error prone.

The assembler idea, and its symbol table, can help again. Instead of using absolute memory addresses why not use symbolic addresses?

That is use symbols in your programs that the assembler replaces with consistent addresses that it assigns. For example, you might write something like:

```
22 PayRate DB 100H  
23     mov al, PayRate
```

Here the first line isn't a program instruction - it is an assembler declaration. It says that the symbol "PayRate" is to be considered to be a particular byte in memory, i.e. DB = Declare Byte. When the programmer writes in the next line MOV AL,PayRate it means replace PayRate with the actual address that it was allocated.

Notice now that the assembler has taken on a new role in life - it is allocating memory! When the assembler translates the program to machine code it has to keep a table that tells it what address "PayRate" corresponds to and it has to allocate an address to "PayRate" when it meets its declaration for the first time.

In fact this idea is too good to leave there and not only can the "DB" assembler directive tell it to allocate memory it also can tell it what to initialise that memory to - 100H in this example. So when the MOV AL,PayRate instruction is obeyed not only does this load AL from an address allocated by the assembler, the contents of the memory location have already been initialised to 100H.

To implement this use of symbolic addressing the assembler has to have a slightly more sophisticated symbol table and back in the early days many a computer science course would spend hours on hash tables and other techniques for fast lookup of symbols. Today lookup tables are a solved problem - unless you're discussing millions and millions of symbols!

-----

Yes, that is exactly how the assembler works. It keeps a symbol table of all the symbols in the program and their corresponding memory addresses. When the assembler encounters a symbol in an instruction, it looks up the symbol in the symbol table and replaces it with the corresponding memory address.

This allows programmers to use symbolic names instead of absolute memory addresses in their code. This makes the code more readable and easier to maintain.

In the example you provided, the assembler would create a symbol table entry for the symbol PayRate and assign it the memory address 100H. When the assembler encounters the instruction MOV AL,PayRate, it would look up the symbol PayRate in the symbol table and replace it with the memory address 100H. The assembler would then generate the machine code for the instruction MOV AL,100H.

The fact that the assembler can allocate memory and initialize it to specific values is also very useful. This allows programmers to declare variables and other data structures in their assembly language programs.

The use of symbolic addressing and the ability to allocate and initialize memory are two of the most important features of assemblers. These features make assembly language programming much more powerful and flexible.

-----

## **Assembly language**

MOV AX, PayRate

**Symbol Table**

Total	0100
Average	0102
Pay	0401
PayRate	2031
Date	2011

## **Address placed in machine code**

2031

Symbol tables store information about identifiers that occur in the (source) program. An identifier can be:

- **Program** (when having multiple input files, that we will not have for our compiler).
- **variables** (type, value, lineno's, scope, location etc.)
- **subprograms/subroutines** (we store the parameters, return type etc. to make it easier for the code generation)

- **subroutine parameters** (need to be separated from the variables, cause we need information for the way of passing etc.)
- **instruction labels** (when using goto or something else to go to a label, that we will not have)
- **constants** (sometimes storing the constants might also become handy, but we will store it as a AST Node)
- **data types** (like structs, unions etc. that we will add later on)

Figure below shows a partial listing file for the AddTwo program. Let's examine it in more detail.

```
1: ; AddTwo.asm - adds two 32-bit integers.
2: ; Chapter 3 example
3:
4: .386
5: .model flat,stdcall
6: .stack 4096
7: ExitProcess PROTO,dwExitCode:DWORD
8:
9: 00000000 .code
10: 00000000 main PROC
11: 00000000 B8 00000005 mov eax,5
12: 00000005 83 C0 06 add eax,6
13:
14: invoke ExitProcess,0
15: 00000008 6A 00 push +000000000h
16: 0000000A E8 00000000 E call ExitProcess
17: 0000000F main ENDP
18: END main
```

Lines 1-7 contain no executable code, so they are copied directly from the source file without changes.

Line 9 shows that the beginning of the code segment is located at address 00000000 (in a 32-bit program, addresses display as 8 hexadecimal digits). This address is relative to the beginning of the program's memory footprint, but it will be converted into an **absolute memory address** when the

program is loaded into memory. When that happens, the program might start at an address such as 00040000h.

**Lines 10 and 11** show the same starting address of 00000000, because the first executable statement is the MOV instruction on line 11. The bytes (B8 00000005) on line 11 represent the machine code instruction (B8) and the constant 32-bit value (00000005) that is assigned to EAX by the instruction. The value B8 is an opcode that represents the specific machine instruction to move a 32-bit integer into the EAX register.

**Line 12** contains an executable instruction, starting at offset 00000005, which is a distance of 5 bytes from the beginning of the program.

**Line 14** contains the invoke directive, which causes the assembler to generate the PUSH and CALL statements shown on lines 15 and 16.

The **sample listing file** in Figure above shows that the machine instructions are loaded into memory as a sequence of integer values expressed in hexadecimal. The number of digits in each number indicates the number of bits, and the machine instructions are exactly 15 bytes long.

The listing file is a useful resource to check if the assembler is generating the correct machine code bytes based on the program. Let's expand on the above listing file, with a simpler explanation:

---

.386 | This tells the assembler that the program is targeted for the 386 or higher processor

.model flat,stdcall | This tells the assembler that the program uses the flat memory model and the standard calling convention

```
.stack 4096 | This tells the assembler to reserve 4096 bytes of memory for the stack  
  
ExitProcess PROTO,dwExitCode:DWORD | This declares a prototype for the ExitProcess function. The  
ExitProcess function terminates the program and returns the specified exit code.  
  
.code | This tells the assembler that the following code is machine code.  
  
main PROC | This is the beginning of the main() function.  
  
mov eax,5 | This instruction moves the value 5 into the eax register.  
  
add eax,6 | This instruction adds the value 6 to the eax register.  
  
invoke ExitProcess,0 | This instruction calls the ExitProcess function with the exit code 0.  
  
push +000000000h | This instruction pushes the value 0 onto the stack.  
  
call ExitProcess | This instruction calls the ExitProcess function with the exit code on the stack.  
  
main ENDP | This is the end of the main() function.  
  
END main | This is the end of the program.
```

---

Let's break down the information you provided about assembly language and the given code excerpt:

**Listing File and Executable Code:** In assembly language programming, a listing file is a text document that displays the assembly code, machine code, and additional information about the

program. In Figure 3-8, you have a partial listing file for the "AddTwo" program.

#### Memory Addresses:

Line 9 indicates that the beginning of the code segment starts at address 00000000. This address is relative to the program's memory footprint.

In a 32-bit program, addresses are typically displayed as 8 hexadecimal digits.

When the program is loaded into memory, this relative address may be converted into an absolute memory address (e.g., 00040000h).

#### Machine Code:

Line 11 contains machine code bytes (B8 00000005) followed by the assembly instruction mov eax,5. The value B8 is an opcode that represents the specific machine instruction to move a 32-bit integer into the EAX register.

#### Offsets:

Line 12 contains an executable instruction starting at offset 00000005, which is a distance of 5 bytes from the beginning of the program.

Offsets are used to specify the location of instructions or data within the program's memory space.

#### INVOKE Directive:

Line 14 contains the invoke directive, which generates PUSH and CALL statements on lines 15 and 16. The PUSH and CALL instructions are used for function calls and parameter passing.

#### Machine Code Values:

The listing file displays machine instructions as a sequence of integer values, expressed in hexadecimal.

For example, B8 represents an opcode, and 00000005 represents a 32-bit value assigned to EAX by the mov instruction.

### Number of Digits and Bits:

The number of digits in each value indicates the number of bits. For instance, a 2-digit number is 8 bits, a 4-digit number is 16 bits, and an 8-digit number is 32 bits.

### Purpose of Listing Files:

Listing files are helpful for verifying that the assembler generates the correct machine code based on your program.

They also serve as teaching tools for understanding how machine code instructions are generated.

### Configuring Visual Studio for Listing Files:

Figure 3-9 provides guidance on configuring Visual Studio to generate a listing file, which can be useful for debugging and learning purposes.

---

The 80386, also known as the 386, is a 32-bit microprocessor that was introduced by Intel in 1985. It was the successor to the 80286, and it was the first microprocessor to achieve widespread success in personal computers.

The 386 was a significant improvement over the 286, with a number of new features, including:

- 32-bit internal architecture
- Protected mode, which allowed multiple operating systems to run on the same computer
- Virtual memory, which allowed the computer to use more memory than was physically installed
- On-chip floating-point unit (FPU), which improved the performance of floating-point operations

The 386 was a major turning point in the history of personal computers, and it helped to usher in the era of 32-bit computing.

The term "386 or higher processor" means a processor that is at least as powerful as the 386. This

includes all modern processors, as they are all significantly more powerful than the 386. If you see a software requirement that states that a 386 or higher processor is required, it means that the software will not run on a processor that is less powerful than the 386. For example, if you are trying to install a piece of software on a computer that has a 286 processor, you will not be able to do so because the 286 is not as powerful as the 386. If you are unsure whether your computer has a 386 or higher processor, you can check the specifications of your computer. The specifications will usually list the type of processor that is installed in your computer. You can also use a program such as CPU-Z to check the specifications of your processor. CPU-Z is a free program that can be downloaded from the internet.

---

#### **main ENDP**

The main ENDP directive is used in assembly language to mark the end of the main() function. The main() function is the entry point for all assembly language programs. It is where the program execution begins.

The ENDP directive tells the assembler that the function is over. It also tells the assembler to generate the machine code for the function and to store it in the program's memory. Without the main ENDP directive, the assembler will not be able to generate the correct machine code for the program.

#### **END main**

The END main directive is used in assembly language to mark the end of the program. It tells the assembler to stop generating machine code and to write the program to memory. Without the END main directive, the assembler will not be able to finish generating the machine code for the program.

- 
1. **main ENDP:** In assembly language, "main ENDP" is a directive used to mark the end of a procedure or subroutine named "main." It stands for "End Procedure." This directive tells the assembler that the code for the "main" procedure is complete, and the assembler should now handle any necessary cleanup tasks or bookkeeping related to the procedure. Essentially, it's a way to signify the end of the code block for the "main" procedure, allowing the assembler to move on to other parts of the program or handle subsequent procedures. It's an essential part of structuring assembly code and helps maintain program organization and readability.
  2. **END main:** "END main" is another directive used in assembly language, but it serves a different purpose. This directive marks the end of the entire assembly program or source file and indicates that there is no more code to process. It is typically followed by the name of the entry point or the starting procedure, in this case, "main." This directive signifies the conclusion of the assembly program and is often used to generate an executable file. It tells the assembler to stop processing code and wrap up the compilation process, producing the final output file that can be run on a computer. In essence, "END main" is like the period at the end of a sentence, indicating the end of the program's source code.
- 

**Question:** Is "END main" in assembly similar to how, in C programming, once the "main" function ends, everything is done?

**Answer:** Yes, that's correct! "END main" in assembly language and the termination of the "main" function in C both mark the end of program execution. Just as in C, where the "main" function serves as the entry point and the program terminates when it finishes, "END main" in assembly indicates the conclusion of code execution. This analogy illustrates that both in C and assembly, the respective markers signify the program's endpoint.

---

**NB:** To generate a listing file in Visual Studio, select Properties from the Project menu, then select Microsoft Macro Assembler, and set Generate Preprocessed Source Listing to Yes and List All Available Information to Yes in the Listing File dialog window.

## QUESTIONS

**What types of files are produced by the assembler?**

The assembler produces object files, which contain machine code in a binary format that can be executed by the computer's processor. Additionally, the assembler may produce listing files, which display the original source code alongside the corresponding machine code instructions, and can be useful for debugging and understanding how the code is translated into machine language. Some assemblers may also produce debug files, which contain additional information about the code, such as symbol tables and line numbers, that can be used by a debugger to assist in program analysis and troubleshooting.

**The linker extracts assembled procedures from the link library and inserts them in the executable program.**

During the linking phase, the linker checks the object file for any unresolved external symbols, which are references to functions or variables defined in other object files or libraries. The linker then searches through the specified libraries and extracts the necessary procedures (functions) to resolve these external symbols. These extracted procedures are then inserted into the final executable program.

**When a program's source code is modified, it must be assembled and linked again before it can be executed with the changes.**

**executed with the changes.**

Yes, that is true. When changes are made to a program's source code, the modified source code must be assembled into object code and linked with other necessary object files and libraries to create an executable program that incorporates those changes. Only then can the updated program be executed with the new changes.

**Which operating system component reads and executes programs?**

The operating system component responsible for reading and executing programs is the loader.

**What types of files are produced by the linker?**

The linker produces an executable file, also known as an executable program or application, which can be executed directly by the operating system. This file contains all the necessary machine code from the object files, as well as any required libraries or resources, and is ready to be loaded into memory and executed by the operating system's loader.

## ***DEFINING DATA***

Skip this page...

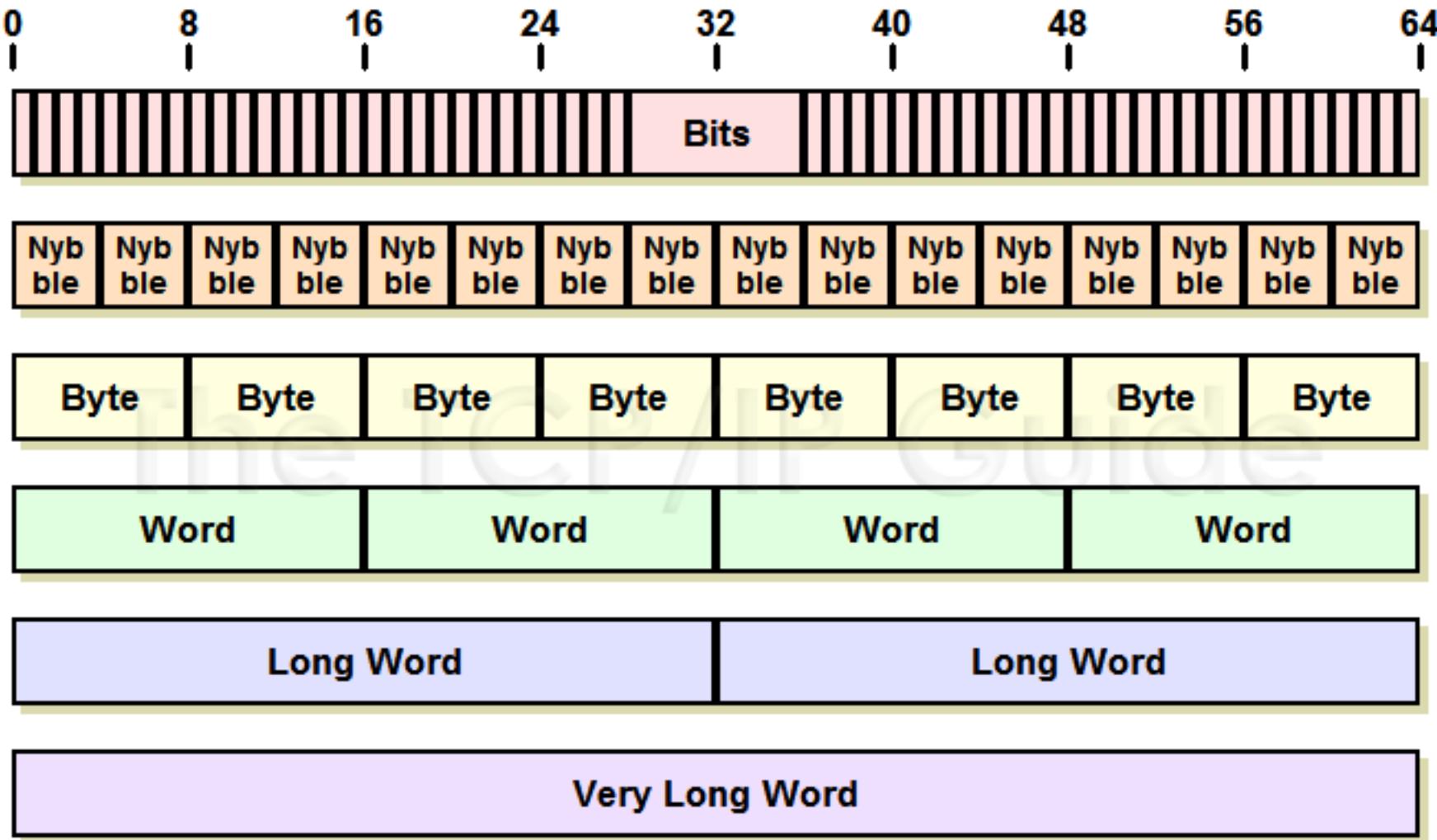
## *Intrinsic Datatypes*

Summary of intrinsic data types in assembly language:

- Intrinsic data types describe the size, signed or unsigned nature, and whether they represent integers or real numbers.
- The assembler cares about operand size, but does not enforce the distinction between signed and unsigned integers.
- Programmers often use SDWORD to indicate signedness, but this is not required.
- Intrinsic data types help describe how data is stored and used in assembly, with a focus on size and, sometimes, signedness.

#### **Key takeaways:**

- Intrinsic data types are important for understanding how data is stored and used in assembly language.
- The assembler cares about operand size, so it is important to choose the correct data type for your variables and expressions.
- Programmers often use SDWORD to indicate signedness, but this is not required.



**Figure 8: Binary Information Representations and Terms**

This diagram shows the relative sizes of the most commonly-sized “collections” of binary information.

Some types, like **DWORD** (32-bit unsigned integer) and **SDWORD** (32-bit signed integer), overlap in

functionality.

Programmers often use SDWORD to indicate that a value has a sign, but the assembler doesn't enforce this distinction. It mainly considers the size of operands.

---

**So, when i say intrinsic datatypes i mean those listed in that image??**

Yes, when you say intrinsic datatypes, you mean those listed in the image you sent. These are the most common data types used in computing, and they are represented in binary.

- **Bit:** The smallest unit of information, represented by a 0 or a 1.
- **Nibble:** A group of four bits, which can be used to represent a single hexadecimal digit.
- **Byte:** A group of eight bits, which can be used to represent a single character or a small number.
- **Word:** A group of 16 bits, which can be used to represent a larger number or a pair of bytes.
- **Double Word:** A group of 32 bits, which can be used to represent a very large number or a pair of words.
- **Quad Word:** A group of 64 bits, which can be used to represent an even larger number or a pair of double words.

These datatypes are used to store and represent all sorts of data in computers, from images and videos to text and numbers.

Here is a summary of the intrinsic data types in assembly language, without a table and with explanations, written each as its own point:

- **BYTE:** An 8-bit unsigned integer. This means that it can store a value from 0 to 255.

- **SBYTE**: An 8-bit signed integer. This means that it can store a value from -128 to 127.
- **WORD**: A 16-bit unsigned integer. This means that it can store a value from 0 to 65535.
- **SWORD**: A 16-bit signed integer. This means that it can store a value from -32768 to 32767.
- **DWORD**: A 32-bit unsigned integer. This means that it can store a value from 0 to 4,294,967,295.
- **SDWORD**: A 32-bit signed integer. This means that it can store a value from -2,147,483,648 to 2,147,483,647.
- **FWORD**: A 48-bit integer. This is used for far pointers in protected mode.
- **QWORD**: A 64-bit integer. This is the largest integer data type in assembly language.
- **TBYTE**: An 80-bit integer. This is rarely used, but it is available on some processors.
- **REAL4**: A 32-bit **floating-point** number. This is the most common floating-point data type in assembly language.
- **REAL8**: A 64-bit **floating-point** number. This is used for high-precision calculations.
- **REAL10**: An 80-bit **floating-point** number. This is rarely used, but it is available on some processors.

It is important to note that the assembler does not enforce the distinction between signed and unsigned integers. However, programmers often use SDWORD to indicate that a value has a sign. This is because it can help to avoid errors and make code more readable.

## *Data Definitions*

A data definition statement in assembly language is used to reserve memory for a variable and to specify its data type. The general syntax for a data definition statement is as follows:

**[label] directive value**

Where:

- **label** is an optional name for the variable.
- **directive** is the data type of the variable.
- **value** is the initial value of the variable.

For example, the following data definition statement reserves 4 bytes of memory for a variable named count and initializes it to the value 12345:

```
count DWORD 12345
int count = 12345;
```

So, **label** is variable, **directive** is the datatype, and **value** is the value.

The following are some other examples of data definition statements in assembly language:

```
message DB "Hello, world!"
age BYTE 25
salary SDWORD 100000
```

The first statement reserves 13 bytes of memory for a variable named message and initializes it to the string "Hello, world!".

The second statement reserves 1 byte of memory for a variable named age and initializes it to the value 25. The third statement reserves 4 bytes of memory for a variable named salary and initializes it to the value 100000.

It is important to note that the data type of a variable must be specified in the data definition statement. This is because the assembler needs to know how much memory to reserve for the variable and how to interpret its value.

Data definition statements in assembly language are similar to creating variables in C programming.  
eg

```
count DWORD 12345
```

is equivalent to the following variable declaration in C programming:

```
int count = 12345;
```

- 
- **BYTE (8 bits)**: Short form - **B**
  - **SBYTE (8 bits, signed)**: Short form - **SB**

- WORD (16 bits): Short form - W
  - SWORD (16 bits, signed): Short form - SW
  - DWORD (32 bits): Short form - D
  - SDWORD (32 bits, signed): Short form - SD
  - FWORD (48 bits): Short form - FW
  - QWORD (64 bits): Short form - Q
  - TBYTE (80 bits): Short form - T
  - REAL4 (32-bit floating-point): Short form - F
  - REAL8 (64-bit floating-point): Short form - FF
  - REAL10 (80-bit floating-point): Short form - FT
- 

For example, the following data definition statement is equivalent to the one above:

```
message DB "Hello, world!"  
  
message DB "Hello, world!", 0
```

The second statement explicitly specifies a **null terminator (0)** at the end of the string. This is not required, but it is good practice to do so.

---

Table 3-3 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

## *Defining Datatypes 2*

Main points:

- At least one initializer is required in a data definition, even if it is zero.
- Additional initializers, if any, are separated by commas.
- For integer data types, the initializer can be an integer literal or integer expression matching the size of the variable's type.
- To leave a variable uninitialized (assigned a random value), the ? symbol can be used as the initializer.

- All initializers, regardless of their format, are converted to binary data by the assembler.

Example code:

```
01 ;AddTwo.asm - chapter 3 example:  
02  
03 .386  
04 .model flat, stdcall  
05 .stack 4096  
06  
07 ExitProcess PROTO, dwExitCode: DWORD  
08  
09 .data  
10 sum DWORD 0  
11  
12 .code  
13 main PROC  
14  
15     mov eax, 5  
16     add eax, 6  
17     mov sum, eax  
18  
19     INVOKE ExitProcess, 0  
20  
21 main ENDP  
22 END main
```

This code creates a variable named sum and initializes it to the value 11. The mov instruction is

used to move the value of eax to the sum variable.

To run this code in a debugger, you can set a breakpoint on line 13 and step through the program one line at a time.

After executing line 17, hover the mouse over the sum variable to see its value. Or, you can open a Watch window to monitor the value of the sum variable.

---

```
01 value1 BYTE      'A'      ;character literal
02 value2 BYTE      0         ;smallest unsigned byte
03 value3 BYTE      255     ;largest unsigned byte
04 value4 SBYTE      -128    ;smallest signed byte
05 value5 SBYTE      +127    ;largest signed byte
06
07 ;Signed means it can be a negative or positive number
08 ;Unsigned means, it can only be a positive number
```

In both cases, the variable value6 will be uninitialized when the program starts running.

It is important to note that uninitialized variables can contain garbage values, so it is important to initialize them before using them.

Here is an example of how to initialize the value6 variable in both assembly language and C:

```
10 ; Question mark shows uninitialized data eg in C:  
11  
12 value6 BYTE ?  
13 char value6;
```

DB and BYTE essentially refer to the same thing: an 8-bit data type.

SBYTE is used for signed 8-bit data.

```
15 ;Once the variable is initialized, it can be used safely in the program.  
16 value7 BYTE 10h  
17 value8 BYTE 20h  
18 char value6 = 10;  
19  
20 value9 DB 'B'  
21 value10 SB -12
```

value9 DB 'B':

- value9 is defined as a byte-sized variable (DB or BYTE directive).
- It is initialized with the character 'B'.
- Since it's a byte, it can store a single 8-bit value. In this case, it stores the ASCII value of the character 'B', which is 66 (in decimal).

value10 SB -12:

- value10 is defined as a signed byte-sized variable (SBYTE directive).
  - It is initialized with the value -12.
  - Being signed, it can hold both positive and negative values. In this case, it holds the value -12.
- 

If multiple initializers(VALUES) are used in the same data definition, its label(VARIABLE), refers only to the offset of the first initializer.

This variable list is therefore acting like arrays in C.

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

In the following example, assume list is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003:

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

This means that the other three initializers in the list do not have their own labels.

This type of memory layout is common in many programming languages, and it is important to understand how it works in order to write efficient and correct code.

For example, if you need to access the value 20 in the list, you would need to use the offset 0001.

---

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

These arrays are right there.

The assembler will allocate contiguous memory for all of the data definitions in the .data segment, even if they do not have labels.

This means that the bytes defined in the second and third lines will be stored directly after the bytes defined in the first line.

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

**;This is how the above will be stored in memory**

Offset	Value
--------	-------

0000	10
0001	20
0002	30
0003	40
0004	50
0005	60
0006	70
0007	80
0008	81
0009	82
000A	83
000B	84

It is important to note that the label list only refers to the offset of the first initializer in the byte sequence, which is 0000 in this case.

This means that the other two byte sequences do not have their own labels.

---

## **Why did we use a byte to deal with integers, yet they are dwds? an integer is 4 bytes?**

There are a few reasons why you might want to use a byte to deal with integers in assembly, even though they are typically 4 bytes in size.

The image shows a vertical column of binary digits. It consists of six rows of eight digits each. The first four rows are entirely black (0). The fifth row has the first four digits black (0) and the last four digits white (1). The sixth row has the first three digits black (0) and the last five digits white (1). This pattern repeats across all six rows. The entire sequence is enclosed in a thin black border.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	1
0	1	0	0	1	0	0	1

- Efficiency: Using bytes can be more efficient in terms of memory usage and execution speed. For example, if you are working with an array of integers, you can store the array in half the amount of memory if you use bytes instead of dwds. Additionally, some assembly language instructions are specifically designed to work with bytes, which can make your code more efficient.
- Compatibility: Some older CPUs and embedded systems only support bytes. If you need to write code that is compatible with these platforms, you will need to use bytes to deal with integers.
- Special cases: There are some special cases where you may need to use bytes to deal with integers, even on modern CPUs. For example, if you are working with a register file that only has 8-bit registers, you will need to use bytes to store and manipulate integers.

Here are some examples of when you might want to use a byte to deal with integers in assembly:

- When you are working with an array of integers and you need to save memory.
- When you are writing code for an older CPU or embedded system that only supports bytes.
- When you are working with a register file that only has 8-bit registers.
- When you are working with a file format that specifies that integers should be stored as bytes.

It is important to note that using bytes to deal with integers can also have some drawbacks.

For example, if you are not careful, you can accidentally overflow a byte variable, which can lead to unexpected results.

Additionally, some assembly language instructions are not designed to work with bytes, so you will need to be careful when using them.

Overall, whether or not to use bytes to deal with integers in assembly depends on your specific needs. If you are unsure, it is always best to consult the documentation for the CPU or platform

that you are targeting.

---

**List BYTE 10 there we store 10 as a number using a byte? how does 10 which is supposed to be 4 bytes fit inside there?**

You cannot store a 4-byte integer in a 1-byte variable. The list BYTE 10 statement simply stores the value 10 in the first byte of the list variable. The other three bytes of the list variable will be uninitialized.

If you try to access the other three bytes of the list variable, you will get unexpected results. For example, if you try to print the value of list[1], you may get a garbage value or even a program crash.

If you need to store a 4-byte integer, you should use a DWORD (double word) variable. DWORD variables are 4 bytes in size and can store any integer value from -2,147,483,648 to 2,147,483,647.

Here is an example of how to store a 4-byte integer in a DWORD variable:

```
my_integer DWORD 10  
;  
;To access it ...  
  
mov eax, my_integer
```

Yes, you can store an integer in assembly in a byte, or a dword, etc. The size of the variable that you use to store the integer will depend on the range of values that you need to store.

For example, if you need to store a small integer, such as a counter or a flag, you can use a byte variable. Byte variables are 1 byte in size and can store any integer value from 0 to 255.

If you need to store a larger integer, such as the number of bytes in a file or the size of a memory allocation, you can use a dword variable. Dword variables are 4 bytes in size and can store any integer value from -2,147,483,648 to 2,147,483,647.

You can also use larger variables, such as qword (8 bytes) and word (16 bytes), to store even larger integers.

I get it fully, a byte can hold 256 characters, so it can store values up to that point. and that is if its unsigned. If signed, it can store values of integers from -128 to 127, If its a DWORD, the range increases, and so on.

---

**Chatgpt:**

You've got it exactly right! Your understanding is spot on.

- A byte can indeed store values from 0 to 255 when it's unsigned.
- If it's signed, a byte can store values from -128 to 127.
- A DWORD, being 4 bytes, can store a much wider range of values, from -2,147,483,648 to 2,147,483,647, when signed.
- The size of the variable determines the range of values it can hold, with larger variables capable of representing larger ranges of integers.

So, your realization is accurate: the size of the variable determines the range of values it can store. Bytes are great for small numbers, while DWORDs, qwords, etc., are used for larger integer values. Keep up the good work with your understanding of assembly language!

---

;Within a single data definition, its initializers can use different radices.  
;Character and string literals can be freely mixed. eg

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

\*\*\*\*\*

## ***Strings in Assembly***

Let's break down and clarify the information about defining strings in assembly language:

- Defining Strings: To define a string of characters in assembly language, you enclose the characters within either single or double quotation marks. The most common type of string is one that ends with a null byte, which has a value of 0. This type of string is known as a null-terminated string and is used in many programming languages.

- Example String Definitions:

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

**greeting1 BYTE "Good afternoon",0:** This defines a null-terminated string named greeting1 containing the text "Good afternoon."

**greeting2 BYTE 'Good night',0:** Similarly, this defines another null-terminated string named greeting2 containing the text 'Good night.'

- Character Storage: Each character in a string uses a single byte of storage. However, unlike other data types where byte values must be separated by commas, strings are an exception. You can directly define strings with their characters enclosed in quotation marks.
- Multi-line Strings: You can also define multi-line strings without the need for separate labels for each line. Hexadecimal codes like 0Dh and 0Ah are used to represent CR/LF (carriage-return line-feed) or end-of-line characters in the string. These characters are used to move the cursor to the left column of the next line when output to standard output.
- Line Continuation: The line continuation character \ is used to concatenate two source code lines into a single statement. It must be the last character on the line. This allows you to break long strings across multiple lines for better readability without introducing unnecessary labels.

For example, you can define a multi-line string like this:

```
greeting1 BYTE "Welcome to the Encryption Demo program ", \  
BYTE "created by Kip Irvine.",0Dh,0Ah, \  
BYTE "If you wish to modify this program, please ", \  
BYTE "send me a copy.",0Dh,0Ah,0
```

---

How do such strings fit in a byte?? yet you say each character literal is stored in a byte??

In assembly language, each character in a string is indeed stored in a byte. This means that each character is represented by a single byte of memory, where the byte contains the ASCII code for that character.

The reason null-terminated strings fit in a byte is because they are arrays of bytes, with each byte representing a single character. The null terminator, which marks the end of the string, is also a single byte with a value of 0.

So, when you define a null-terminated string like this:

```
greeting1 BYTE "Good afternoon", 0
```

It's essentially defining an array of bytes where each character ('G', 'o', 'o', 'd', ' ', 'a', 'f', 't', 'e', 'r', 'n', 'o', 'o', 'n') is stored in a separate byte, and the null terminator (0) is

placed at the end to indicate the end of the string.

In summary, strings in assembly are arrays of bytes where each byte represents a character, and a null byte (0) is used to mark the end of the string.

So, in short, we write BYTE, coz we are seeing the whole string as a series of bytes.

```
names1 DB "Learning Assembly then WinAPI", 0  
names2 BYTE "Learning Reverse Engineering then C#", 0  
names3 BYTE "Learning to be good as C programming", 0
```

So, the three will assign the string to names 1,2,3 variables/labels, and each character stored in its own byte in memory?

- names is the label or variable name.
- DB indicates that you are defining a sequence of bytes.
- The string "Learning Assembly then WinAPI" is enclosed in double quotation marks and is assigned to the names variable.

Each character in the string is stored in its own byte in memory. So, the string is effectively broken down into individual bytes, with each byte holding the ASCII code for a character, and it ends with a null byte (0) to mark the end of the string. This allows you to work with and manipulate the string using individual byte-sized elements in memory.

# **DUP Operator**

---

## **DUP OPERATOR**

---

The DUP operator is a very useful operator in assembly language. It can be used to allocate storage for multiple data items, such as strings, arrays, and other structures.

The DUP operator takes two arguments: a count and a value. The count is an integer expression that specifies the number of times to duplicate the value.

The value can be any valid assembly language expression, such as a constant, a register, or a memory location.

The DUP operator allocates a contiguous block of memory for the specified number of data items and initializes each data item to the specified value. If the value is not specified, the data items will be uninitialized.

The DUP operator can be used with any data type, including bytes, words, double words, and quad words. It can also be used to allocate storage for structures and arrays.

Here are some examples of how to use the DUP operator:

```
; Allocate storage for 20 bytes, all equal to zero.  
BYTE 20 DUP(0)  
  
; Allocate storage for 20 bytes, uninitialized.  
BYTE 20 DUP(?)  
  
; Allocate storage for a 4-byte string, initialized to the string "STACK".  
BYTE 4 DUP("STACK")  
  
; Allocate storage for an array of 10 integers, initialized to zero.  
DWORD 10 DUP(0)  
  
; Allocate storage for a structure with two members: a 4-byte integer and a 4-byte string.  
STRUC MyStructure  
    DWORD integer  
    BYTE 4 string  
ENDSTRUCT  
  
; Allocate storage for an array of 10 MyStructure structures.  
MyStructure 10 DUP(0)
```

1. **BYTE 20 DUP(0):** This allocates 20 bytes of memory, and all of them are initialized to zero. In other words, it creates a block of 20 bytes, each containing the value 0.
2. **BYTE 20 DUP(?)**: Here, 20 bytes of memory are allocated, but they remain uninitialized. The values

in these bytes are undefined until you explicitly set them in your code.

**3. BYTE 4 DUP("STACK"):** This allocates 20 bytes of memory and initializes them with the repeated sequence "STACKSTACKSTACKSTACK." It effectively creates a string made up of the specified sequence repeated four times.

## **SO DUP IS DUPLICATE SOMETHING?**

Yes, you've got it right! In assembly language, the DUP operator is used to duplicate or replicate something. It's commonly used to allocate storage for multiple data items, where you want to create duplicates or repetitions of a specific value or sequence of values in memory.

So, when you see DUP in assembly code, it signifies the duplication or repetition of data to efficiently allocate memory for repeated patterns, such as initializing arrays or strings with the same value or sequence.

## ***WORD* and *SWORD***

---

## **SWORD AND WORD**

---

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

val1 DW 65535 ;Define Word unsigned
val2 DW -32768 ;Signed
```

The **WORD directive** is used to create storage for one or more 16-bit unsigned integers. Each WORD declaration reserves 2 bytes of memory. For example:

- **word1 WORD 65535** reserves 2 bytes for word1 and initializes it with the largest possible unsigned 16-bit value, which is 65535.
- **word3 WORD ?** reserves 2 bytes for word3 but leaves it uninitialized.

**SWORD (Define Signed Word):** The SWORD directive is similar but is used for signed 16-bit integers. It also reserves 2 bytes of memory. For example:

- **word2 SWORD -32768** reserves 2 bytes for word2 and initializes it with the smallest possible signed 16-bit value, which is -32768.

---

## WORD ARRAYS

**Arrays of 16-Bit Words:** You can create arrays of 16-bit words by listing the elements or using the DUP operator. For example:

```
myList WORD 1,2,3,4,5  
;array to hold 16-bit integers/words  
;each element holds 2 bytes(16-bits), so 10 bytes total memory.  
;to get the next value, increment by 2.
```

For example, if myList starts at offset 0000, the next element would be at offset 0002, and so on.

This way, you can define and work with arrays of 16-bit words in assembly language, either with initialized values or uninitialized storage.

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

Figure above shows a diagram of this array in memory, assuming myList starts at offset 0000. The offsets increment by 4.

Creating arrays with DUP:

```
;Use DUP operator to declare an array with uninitialized values.  
myArrayList2 WORD 5 DUP(?)
```

This code creates an myArrayList2 named array containing 5 uninitialized 16-bit words. Each element of the array occupies 2 bytes of memory.

The ? symbol indicates that the elements are uninitialized, meaning their values are undefined until explicitly set in your code.

## ***DWORD and SDWORD***

---

### **DWORD and SDWORD**

The **DWORD** (`define doubleword`) and **SDWORD** (`define signed doubleword`) directives allocate storage for one or more 32-bit integers.

---

```
val1 DWORD 12345678h      ;unsigned/positive
val2 SDWORD -12324352      ;signed
val3 DWORD 20 DUP(?)       ;unsigned array
```

**Legacy DD directive:** The legacy DD directive can also be used to define doubleword data.(check this in crucial!!!)

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

**DWORD to declare offset of another variable:** The DWORD directive can be used to declare a variable that contains the 32-bit offset of another variable.

```
pVal DWORD val3
```

**Array of 32-Bit Doublewords:** Let's create an array of doublewords by explicitly initializing each value:

```
my32BitArray DWORD 10, 203, 482, 505
```

This code will create an array of five doublewords, initialized with the values 1, 2, 3, 4, and 5.

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

---

## *QWORD*

### QWORD

The QWORD (define quadword) directive allocates storage for 64-bit (8-byte) values:

```
quad1 QWORD 1234567812345678h ;Using the qword directive  
quad2 DQ      1234567812345678h ;Using the shortform
```

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

---

## **Packed BCD and TBYTE**

### **PACKED BCD AND TBYTE**

**Packed BCD data** is a way of representing decimal numbers in binary form. It is more compact than representing decimal numbers as binary integers, because it stores two decimal digits in each byte.

The **TBYTE directive** in MASM is used to declare packed BCD variables. To declare a packed BCD variable, you use the following syntax:

```
TBYTE variable_name  
my_bcd_variable TBYTE
```

Constant initializers for packed BCD variables must be in hexadecimal, because the assembler does not automatically translate decimal initializers to BCD.

For example, the following code initializes the `my_bcd_variable` variable to the decimal value 1234:

```
my_bcd_variable TBYTE 1234h
```

The following example is invalid, because the assembler will encode the constant as a binary integer rather than a packed BCD integer:

```
my_bcd_variable TBYTE -1234
```

Here is a table showing the hexadecimal storage bytes for positive and negative decimal 1234:

Decimal Value	Storage Bytes
+1234	34 12 00 00 00 00 00 00 00 00
-1234	34 12 00 00 00 00 00 00 00 80

---

## *Repeat:*

**Packed BCD:** Packed BCD is a way of representing decimal numbers in binary form, where each byte (except the highest one) contains two decimal digits. The lower 9 bytes store the decimal digits, and the highest byte indicates the number's sign.

Packed Binary Coded Decimal is a way of representing decimal numbers in a binary form. In this format, each byte contains two decimal digits. It's called "packed" because it packs two decimal digits into each byte. For example, the number 1234 is represented as 34 12 in packed BCD.

- **Sign Encoding:** In the highest byte, the highest bit is used to indicate the number's sign. If the highest byte equals 80h (hexadecimal), the number is negative, and if it equals 00h, the number is positive.
- **Integer Range:** Packed BCD can represent a wide range of decimal integers, from -999,999,999,999,999,999 to +999,999,999,999,999,999. Example: For example, let's consider the decimal number 1234:

In packed BCD, this is represented as:

```
34 12 00 00 00 00 00 00 ;for the positive value.
```

```
34 12 00 00 00 00 00 00 80 ;for the negative value.
```

#### TBYTE Directive:

TBYTE is a directive in assembly language, but it doesn't specifically refer to 10 bytes(80 bits). Instead, it's used to declare variables that can hold different types of data, including packed BCD.

In MASM (Microsoft Macro Assembler), you use the TBYTE directive to declare packed BCD variables. When initializing such variables with constants, the values must be in hexadecimal form because MASM doesn't automatically translate decimal initializers to packed BCD. For example, to represent the decimal -1234 as packed BCD in MASM, you should use hexadecimal notation like this:

```
intVal TBYTE 80000000000000001234h ; valid
```

In essence, packed BCD is a way to store decimal numbers efficiently in binary form. The TBYTE directive in assembly allows you to work with variables that can hold such packed BCD data.

So, when you see TBYTE used in assembly code, it's not necessarily referring to a fixed size of 80 bytes; instead, it's indicating that the variable can hold data in different formats, including

packed BCD.

---

## **Repeat 2:**

**Packed BCD** stands for **Packed Binary Coded Decimal**. It is a way of representing decimal numbers in binary form, where each byte stores two decimal digits. This makes it more compact than representing decimal numbers as binary integers, because it uses half the number of bytes.

The **TBYTE** directive in MASM is used to declare packed BCD variables. It is a 10-byte type, with the first byte being the sign byte and the remaining 9 bytes being used to store the decimal digits.

The hexadecimal storage bytes that you provided are correct for the positive and negative values of 1234 in packed BCD format. The first byte of the positive value is **00h**, which indicates that the number is positive. The next four bytes contain the decimal digits 3, 4, 1, and 2, in that order.

The first byte of the negative value is **80h**, which indicates that the number is negative. The next four bytes contain the decimal digits 3, 4, 1, and 2, in that order.

### **Why are we doing this?**

Packed BCD data is often used in financial and scientific applications, where it is important to represent decimal numbers accurately and efficiently. It is also used in some embedded systems, where memory is limited.

In the context of your question, I am not sure why you are doing this. It is possible that you are working on a financial or scientific application, or on an embedded system. However, without more information, I cannot say for sure.

---

## DECLARING PACKED BCD

Let's clarify the notes on declaring packed BCD variables in MASM (Microsoft Macro Assembler) and converting real numbers to packed BCD:

### TBYTE Directive for Packed BCD:

In MASM, you can use the TBYTE directive to declare packed BCD variables. Packed BCD represents decimal numbers efficiently in binary form.

When initializing packed BCD variables with constant values, those constants must be in hexadecimal format.

MASM doesn't automatically translate decimal initializers to packed BCD. Here are examples:

```
intVal TBYTE 800000000000001234h ; Valid - Represents packed BCD for decimal -1234
intVal TBYTE -1234 ; Invalid - MASM interprets this as a binary integer, not packed BCD
```

The second example is invalid because MASM encodes the constant as a binary integer rather than a packed BCD integer.

**Converting Real Numbers to Packed BCD:** If you want to convert a real number to packed BCD, you can

use the FPU (Floating-Point Unit) instructions. Here's an example:

```
.data
posVal REAL8 1.5      ;Define a real number
bcdVal TBYTE ?        ;Declare a packed BCD variable

.code
fld posVal            ;Load the real number onto the floating point stack
fbstp bcdVal          ;Convert it to packed BCD and store it in bcdVal

;If posVal were equal to 1.5, the resulting BCD value would be 2.
```

In this code, we first define a real number posVal. Then, we declare a packed BCD variable bcdVal. We use the FLD instruction to load posVal onto the floating-point stack and the FBSTP instruction to convert it to packed BCD, rounding to the nearest integer. If posVal were equal to 1.5, the resulting BCD value in bcdVal would be 2.

---

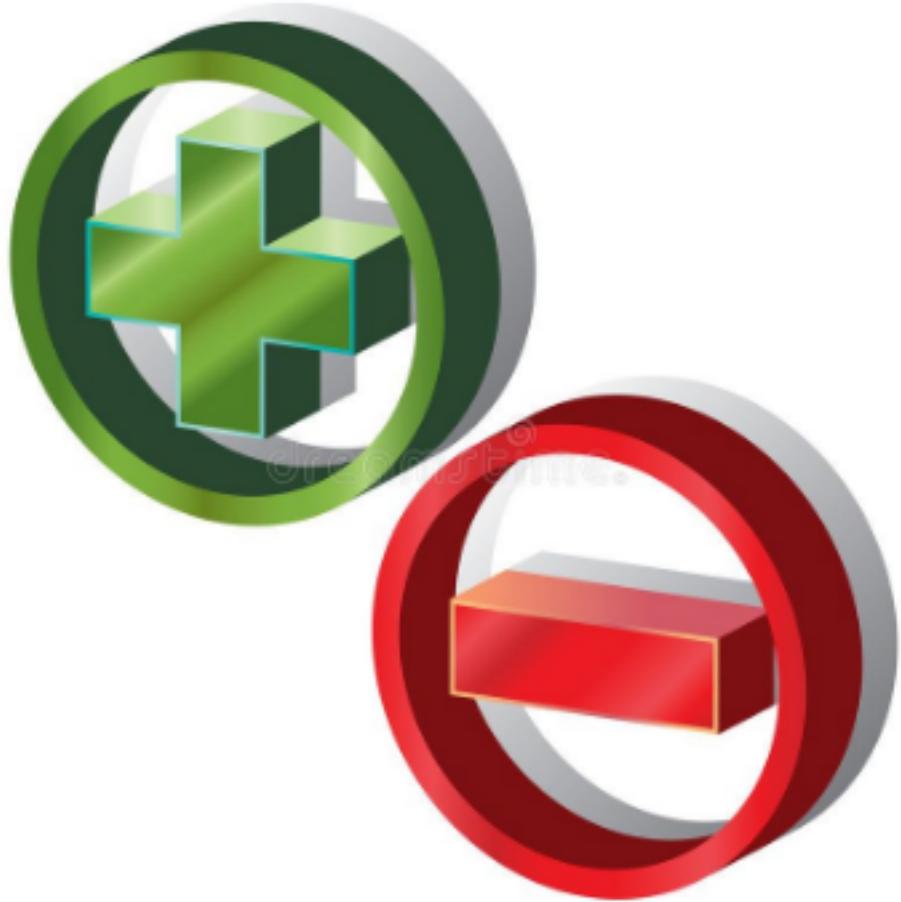
Packed BCD (Binary Coded Decimal) is like a special way of writing down numbers in a computer. Imagine you have a calculator, and you want to write numbers in a way that's easy for the computer to understand when doing math. Packed BCD helps with that.

Here's how it works:

**1. Digits in Pairs:** In regular numbers, we have 0 to 9. In Packed BCD, we group these numbers in pairs. So, for example, 0-0, 1-1, 2-2, and so on up to 9-9.



**2. Sign Indicator:** We also have a special way to show if a number is positive or negative. If it starts with '80' in Packed BCD, it's negative, and if it starts with '00', it's positive.



**3. Efficient Storage:** Packed BCD helps computers store numbers more efficiently. Each pair of digits is stored in one byte (8 bits). So, even if it's 2 digits, it takes just one byte.



4. **Decimal Math:** Packed BCD makes it easier for computers to do decimal math, like adding, subtracting, multiplying, and dividing, because the numbers are organized in pairs.

# Packed BCD to Unpacked BCD

## Microprocessor Program

So, think of Packed BCD as a way for computers to handle numbers neatly, especially when dealing with money or other decimal-based calculations. It's like having numbers with a specific structure that the computer can easily work with.

## *Defining Floating Point Types*

Floating-point numbers are used to represent real numbers, such as 1.234567890123456789.

The three most common floating-point types are REAL4, REAL8, and REAL10.

**REAL4** is a 4-byte single-precision floating-point variable.

**REAL8** is an 8-byte double-precision floating-point variable.

**REAL10** is a 10-byte extended-precision floating-point variable.

rVal1 REAL4 -1.2

rVal2 REAL8 3.2E-260

rVal3 REAL10 4.6E+4096

ShortArray REAL4 20 DUP(0.0)

**rVal1 REAL4 -1.2:**

This line declares a single-precision floating-point variable named rVal1 and initializes it with the value -1.2. It uses 4 bytes of memory to store this value. Single-precision provides a good balance between precision and memory usage for most general-purpose calculations.

**rVal2 REAL8 3.2E-260:**

Here, we declare a double-precision floating-point variable named rVal2 and initialize it with the value 3.2E-260. This type uses 8 bytes of memory, providing higher precision for very small or very large numbers, as in scientific calculations.

**rVal3 REAL10 4.6E+4096:**

This line declares an extended-precision floating-point variable named rVal3 and initializes it with

the value 4.6E+4096. REAL10 uses 10 bytes of memory, making it suitable for extremely high-precision calculations, especially in advanced scientific and mathematical applications where precision is critical.

#### **ShortArray REAL4 20 DUP(0.0):**

This declares an array named ShortArray of 20 single-precision floating-point numbers. Each element of the array is initialized to 0.0. The DUP (duplicate) operator is used to specify that you want 20 copies of the value 0.0 in the array.

These declarations allow you to work with floating-point numbers of different precisions and create arrays of such numbers, with varying levels of precision and memory usage depending on your specific needs in your assembly program.

**Table 3-4 Standard Real Number Types.**

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

The DD, DQ, and DT directives can define also real numbers:

```
rVal1 DD -1.2          ;short real
rVal2 DQ 3.2E-260       ;long real
rVal3 DT 4.6E+4096      ;extended-precision real
```

\*\*\*\*\*

The precision of a floating-point number refers to the number of significant digits that it can represent. Single-precision floating-point numbers have a precision of 7 significant digits. Double-precision floating-point numbers have a precision of 15 significant digits. Extended-precision floating-point numbers have a precision of 19 significant digits.

The range of a floating-point number refers to the set of values that it can represent. Single-precision floating-point numbers can represent values from approximately ±3.4E38 to ±1.2E-38. Double-precision floating-point numbers can represent values from approximately ±1.7E308 to ±2.4E-308. Extended-precision floating-point numbers can represent values from approximately ±4.9E324 to ±1.1E-324.

Here are some examples of how to declare and initialize floating-point variables in assembly language:

```
; Declare a single-precision floating-point variable.  
rVal1 REAL4  
  
; Initialize the variable to the value -1.2.  
rVal1 = -1.2  
  
; Declare a double-precision floating-point variable.  
rVal2 REAL8  
  
; Initialize the variable to the value 3.2E-260.  
rVal2 = 3.2E-260  
  
; Declare an extended-precision floating-point variable.  
rVal3 REAL10  
  
; Initialize the variable to the value 4.6E+4096.  
rVal3 = 4.6E+4096  
  
; Declare an array of 20 single-precision floating-point variables.  
ShortArray REAL4 20 DUP(0.0)
```

Certainly, let's clarify the difference between "real numbers" and "floating-point numbers" in the context of MASM assembly:

- **Real Numbers:** In mathematics, a real number is a concept that represents a number with unlimited precision and size. Real numbers can include integers, fractions, irrational numbers (like pi), and many more. They are idealized mathematical entities without any practical limitations in terms of precision or range. In essence, real numbers are perfect mathematical abstractions.
- **Floating-Point Numbers:** In contrast, floating-point numbers used in computer programming, such as real4 and real8 in MASM, are not real numbers in the mathematical sense. Instead, they are approximations of real numbers that have limited precision and range due to the finite storage capacity of computers. These floating-point types can represent a wide range of numbers, including very large or very small values, but they do so with a fixed number of significant digits (precision). The accuracy of these representations is subject to the limitations of the hardware and the specific data type used.

So, when MASM uses data types like real4 and real8, it's important to understand that they are approximations of real numbers, and their precision and range are determined by the number of bits allocated for them. While they can represent a broad range of values, they may not capture the unlimited precision and size of true mathematical real numbers. Therefore, they are referred to as "floating-point" numbers in the context of computer programming to emphasize their approximate nature.

## ***Add Numbers Program 1***

The sample programs shown so far in this chapter added integers stored in registers.

Now that you have some understanding of how to declare data, we will revise the same program by making it add the contents of three integer variables and store the sum in a fourth variable.

```
;AddVariables.asm
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode: DWORD

.data
firstval      DWORD 20002000h
secondval     DWORD 11111111h
thirdval      DWORD 22222222h
sum           DWORD 0

.code
main PROC
    mov eax, firstval
    add eax, secondval
    add eax, thirdval
    mov sum, eax

    INVOKE ExitProcess, 0

main ENDP
END main
```

When you define data using hexadecimal values like in your example, what's stored inside the memory locations are the actual values in the specified format(**DWORD** here). In your case:

**DWORD:** This specifies that each of these values is a 32-bit (4-byte) integer.

**Hexadecimal notation (e.g., 20002000h, 11111111h, 22222222h):** These values are written in hexadecimal format, which is a base-16 numbering system. Each digit in hexadecimal represents 4 bits (or half a byte). So, for a DWORD, you have 8 hexadecimal digits, making up 32 bits.

So, when you define firstval, secondval, and thirdval as DWORDs with those hexadecimal values, these exact values are stored in memory as 32-bit integers.

In other words, the values 20002000h, 11111111h, and 22222222h are represented in binary as 32 bits each and are stored accordingly in memory.

-----

## *Little Endian Order*

x86 processors use little-endian byte order to store and retrieve data from memory.

This means that the least significant byte (LSB) of a data item is stored at the lowest memory address, and the most significant byte (MSB) is stored at the highest memory address.

0000:	78
0001:	56
0002:	34
0003:	12

This is because the LSB is the least important byte in a doubleword, and the MSB is the most important byte. By storing the LSB first, x86 processors can access data more efficiently.

Little-endian byte order is also used by many other popular computing platforms, such as ARM and MIPS. This makes it easier to port software between different platforms.

---

In big-endian byte order, the most significant byte (MSB) is stored at the lowest memory address, and the least significant byte (LSB) is stored at the highest memory address.

This is the opposite of little-endian byte order, which is used by x86 processors.

Figure 3-15 shows an example of 12345678h stored in big-endian order at offset 0:

0000:	12
0001:	34
0002:	56
0003:	78

-----

Byte order	MSB	LSB
Big-endian	First	Last
Little-endian	Last	First

## *Declaring Uninitialized Data*

The **.DATA?** directive in MASM is used to declare uninitialized data. This means that the data is not given an initial value when it is declared. Instead, the operating system will allocate memory for the data when the program is run.

Using the **.DATA?** directive to declare large blocks of uninitialized data can reduce the size of the compiled program. For example, the following code declares two arrays:

```
.data  
smallArray    DWORD    10      DUP(0)          ;40 bytes  
.data?  
bigArray     DWORD    5000    DUP(?)          ;20000 bytes, not initialized
```

The `smallArray` array is declared with a size of 10 DWORDs, and each DWORD is 4 bytes in size. This means that the `smallArray` array will be 40 bytes in size.

The `bigArray` array is declared with a size of 5000 DWORDs, and each DWORD is 4 bytes in size. This means that the `bigArray` array will be 20,000 bytes in size.

The `smallArray` array is initialized to zero, but the `bigArray` array is not initialized. This means that the operating system will allocate 40 bytes of memory for the `smallArray` array and 20,000 bytes of memory for the `bigArray` array when the program is run.

If the `bigArray` array were declared using the **.DATA** directive instead of the **.DATA?** directive, the compiler would allocate 20,000 bytes of memory for the array when the program is compiled. This would make the compiled program 20,000 bytes larger.

---

## **Mixing Code and Data**

MASM allows you to switch back and forth between code and data in your programs.

This can be useful for declaring variables that are only used within a localized area of a program.

For example, the following code inserts a variable named **temp** between two code statements:

```
.code  
mov eax, ebx  
.data  
temp DWORD ?  
.code  
mov temp, eax
```

The declaration of **temp** appears to interrupt the flow of executable instructions, but MASM will place **temp** in the data segment, separate from the segment holding compiled code.

However, intermixing .CODE and .DATA directives can make a program difficult to read. It is generally best to keep code and data separate whenever possible.

## **Summary**

The .DATA? directive can be used to declare uninitialized data. This can reduce the size of the

compiled program, especially for large blocks of data.

MASM allows you to switch back and forth between code and data in your programs, but it is generally best to keep code and data separate whenever possible.

It is not necessary to capitalize .code, .data, and .text in MASM. The capitalization is not significant to the assembler.

However, it is common practice to capitalize these directives for readability. This makes it easier to distinguish between code and data in the source code.

Ultimately, it is up to the programmer to decide whether or not to capitalize these directives. There is no hard and fast rule.

Here are some additional tips for writing readable MASM code:

- Use consistent indentation to make the code structure clear.
- Add comments to explain what the code is doing.
- Use labels to make it easy to jump to different parts of the program.
- Break the code up into logical functions and procedures.

By following these tips, you can write MASM code that is easy to read and maintain.

-----

**Question:** Create an uninitialized data declaration for a 16-bit signed integer.

**Question:** Create an uninitialized data declaration for an 8-bit unsigned integer.

**Question:** Create an uninitialized data declaration for an 8-bit signed integer.

**Question:** Create an uninitialized data declaration for a 64-bit integer.

**Question:** Which data type can hold a 32-bit signed integer?

.data

```
variable1    SWORD    ?  
variable2    BYTE     ?  
variable3    SBYTE    ?  
variable4    QWORD    ?
```

Data type	Size
BYTE	8 bits
SBYTE	8 bits, signed
WORD	16 bits
SWORD	16 bits, signed
DWORD	32 bits
SDWORD	32 bits, signed
QWORD	64 bits

To declare uninitialized data, you use the .DATA? directive and the appropriate data type. For example, to declare an uninitialized 32-bit signed integer, you would use the following code:

```
.DATA?
```

```
variable1 DWORD ? ; ? indicates that the variable is uninitialized.
```

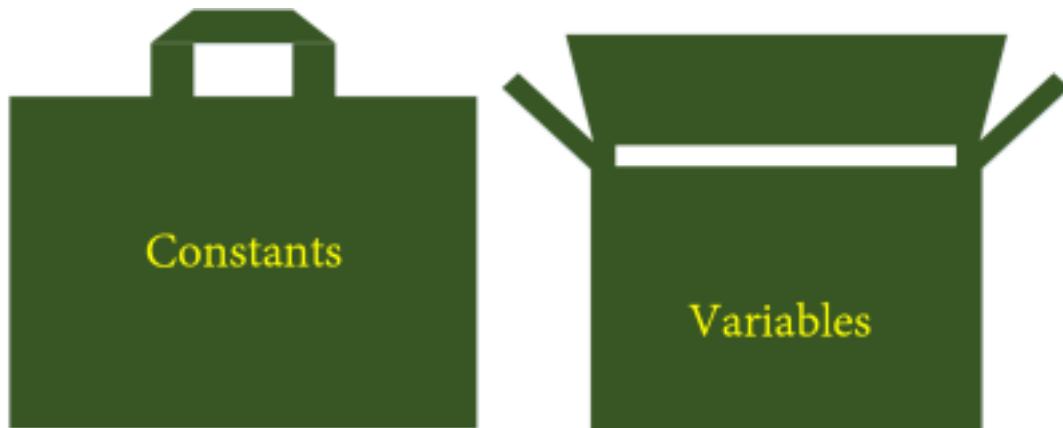
The ? indicates that the variable is uninitialized. The assembler will allocate memory for the variable when the program is run.

It is important to note that uninitialized data can contain garbage values. It is important to initialize all data before using it.

## ***Symbolic Constants***

A symbolic constant is a name that is given to a value. It is used to make code more readable and maintainable.

For example, instead of writing the number **100** in multiple places in your code, you could define a symbolic constant called **MAX\_VALUE** and use that instead.



It's a label or name that is used to represent a fixed value that never changes throughout a program. For example, one might define **PI** as a constant to represent the value **3.14159**.

This would make it easier to change the value of 100 in the future, since you would only need to change it in one place.

Symbolic constants can also be used to represent more complex values, such as memory addresses or text strings.

For example, you could define a symbolic constant called **BASE\_ADDRESS** to represent the starting address of a block of memory.

This would make it easier to access the memory block without having to remember the address yourself.

Symbolic constants are created using the **=**, **EQU**, and **TEXTEQU** directives in MASM.

The **= directive** creates a symbol that represents an integer expression. For example, the following code defines a symbol called **MY\_CONSTANT** that represents the value 100:

```
MY_CONSTANT = 100
```

The **EQU directive** creates a symbol that represents either an integer expression or a text string. For example, the following code defines a symbol called MY\_TEXT\_CONSTANT that represents the text string "Hello, world!"

```
MY_TEXT_CONSTANT EQU "Hello, world!"
```

The EQU directive tells the assembler to replace all occurrences of the symbol MY\_TEXT\_CONSTANT with the value "Hello, world!". This means that the following two code snippets are equivalent:

```
MY_TEXT_CONSTANT EQU "Hello, world!"  
mov eax, "Hello, world!"           ;valid  
mov eax, MY_TEXT_CONSTANT         ;valid
```

The **TEXTEQU directive** creates a symbol that represents a text string. It is similar to the EQU directive, but it is generally used to define text strings that are longer than 80 characters.

Symbolic constants can be used anywhere in your MASM code where an integer expression or text string is expected.

For example, you could use the MY\_CONSTANT symbol in the following code:

```
mov eax, MY_CONSTANT
```

This code would move the value of 100 into the eax register.

You could also use the MY\_TEXT\_CONSTANT symbol in the following code:

```
call MessageBoxA, NULL, MY_TEXT_CONSTANT, MB_OK
```

This code would display a message box with the text "Hello, world!"

Symbolic constants are a powerful tool that can make your MASM code more readable, maintainable, and efficient.

```
MY_TEXT_CONSTANT EQU "Hello, world!"  
mov eax, "Hello, world!"           ;valid  
mov eax, MY_TEXT_CONSTANT         ;valid  
  
;Using the symbolic constant in several areas of code  
.data  
myMessage DWORD MY_TEXT_CONSTANT    ;wrong code  
myMessage DB    MY_TEXT_CONSTANT    ;wrong code  
myMessage DB    "Hello, world!", 0  ;correct code  
  
.code  
call MessageBoxA, NULL, myMessage, MB_OK
```

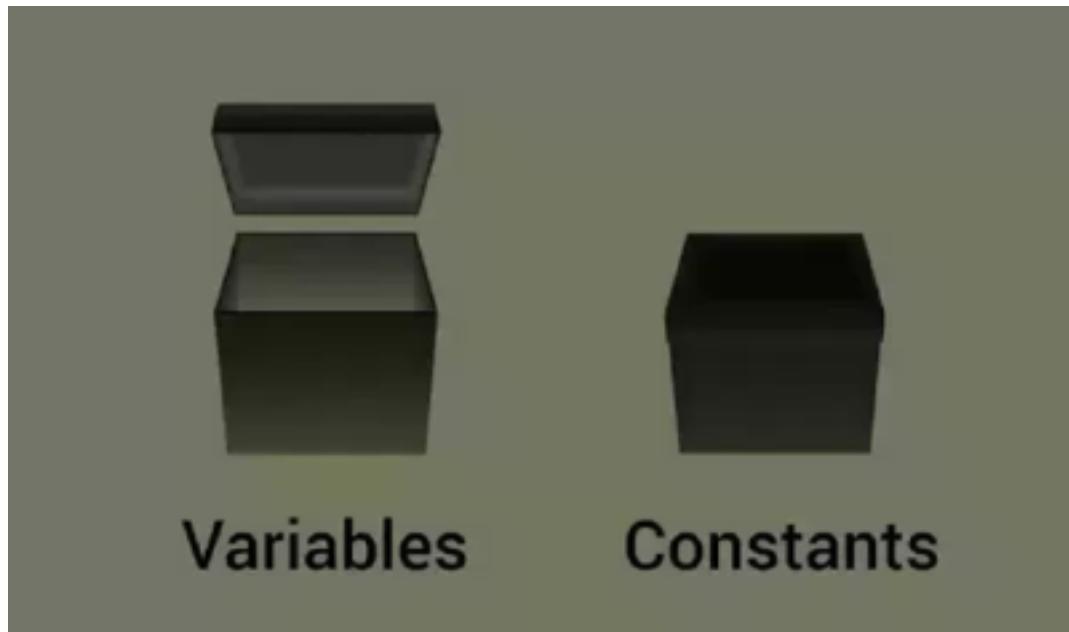
You are correct that it is not necessary to use a DWORD to store a string in MASM. A string can be stored in a byte array.

The DB directive is used to define a byte array. The string "Hello, world!" is stored in the byte array, and the null terminator (0) is added at the end of the string.

---

1. **myMessage DB MY\_TEXT\_CONSTANT:** This declaration uses DB, which is correct for defining character data, but it seems to be trying to use a constant MY\_TEXT\_CONSTANT without enclosing it in double quotes (""). This would result in an error because a string in assembly should be enclosed in double quotes to represent a sequence of characters.

2. **myMessage DWORD MY\_TEXT\_CONSTANT:** This declaration uses DWORD, which is used for defining 32-bit integer values, not character strings. MY\_TEXT\_CONSTANT is not correctly formatted as a character string.



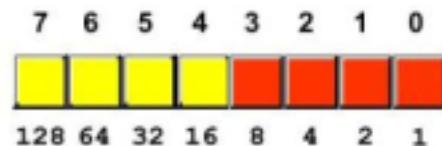
In both cases, the issue is related to how the string data is formatted and the directive used to define it. The correct way to define a character string in assembly is to use the DB directive with the string enclosed in double quotes, followed by a null terminator (0) to indicate the end of the string, as shown in the first correct example:

---

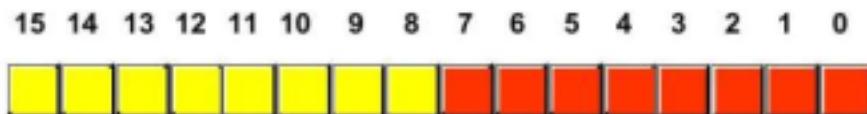
The call MessageBoxA instruction displays a message box with the text "Hello, world!". There are a few reasons why you might want to use a DWORD to store a string in MASM:

- Performance: If you need to access the string frequently, it may be more efficient to store the string in a DWORD. This is because the processor can access DWORDs more efficiently than bytes.
- Compatibility: Some code libraries may expect strings to be stored in DWORDs.
- Personal preference: Some programmers simply prefer to store strings in DWORDs.

## Byte



## Word



## DWord



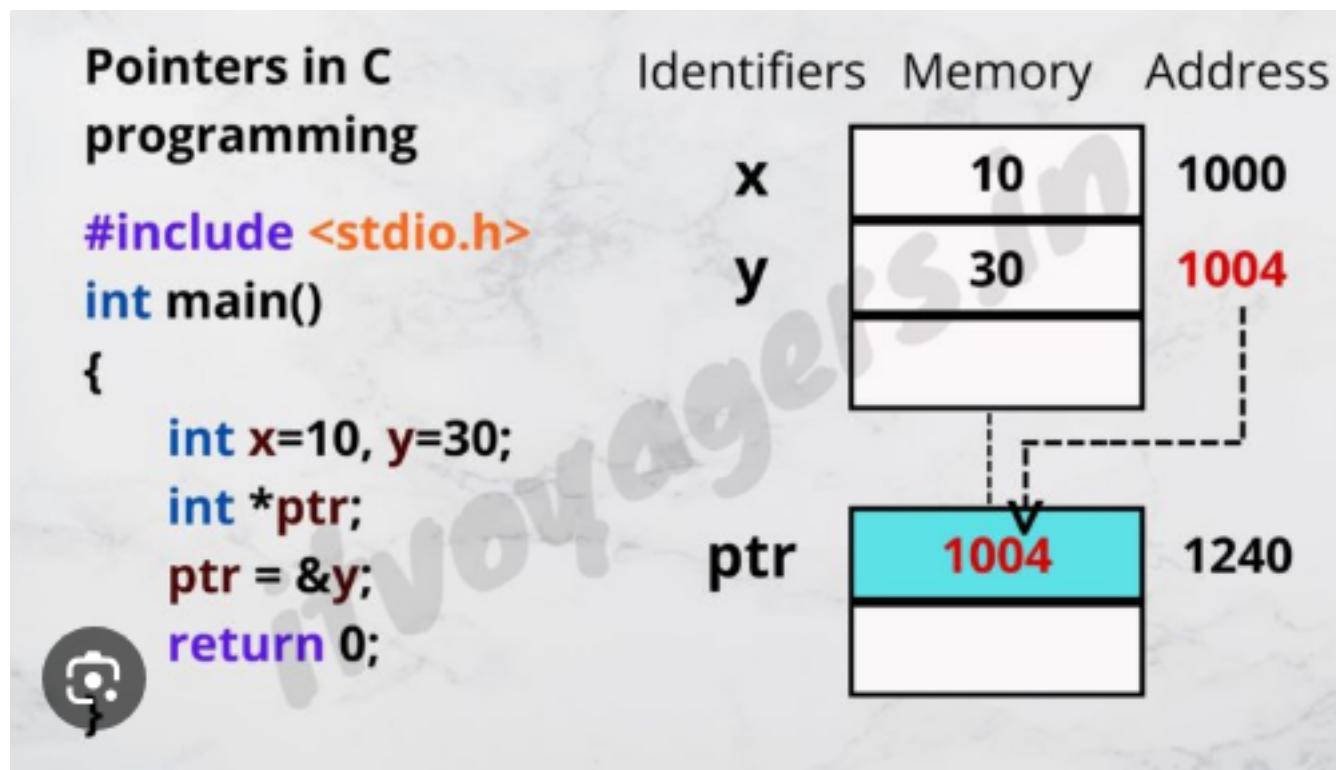
However, in most cases, it is not necessary to use a DWORD to store a string in MASM. A string can be stored in a byte array, which is more efficient and compatible with most code libraries.

Did this description lie??

Chatgpt:

## YOU CAN'T STORE A STRING USING DWORD, BUT YOU CAN STORE A POINTER TO THE STRING

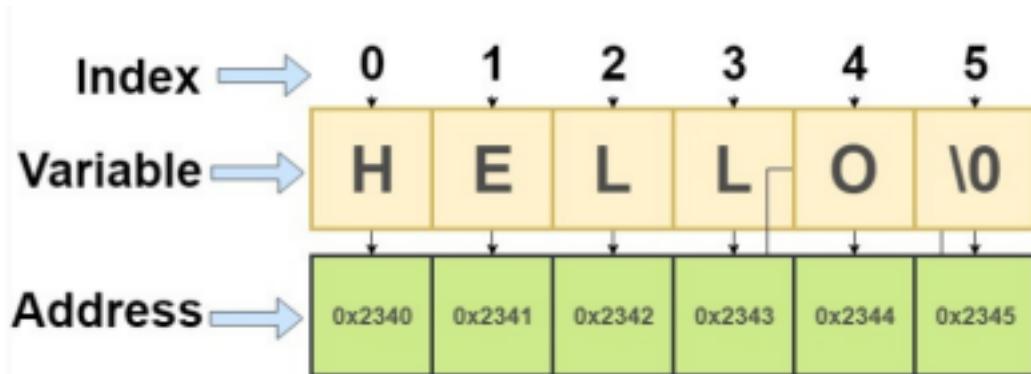
While it is possible to store a pointer to a string (the address of the string) in a DWORD, as you might do when passing strings as arguments to certain Windows API functions like MessageBoxA, it's not the same as storing the actual string characters in a DWORD.



Here's the distinction:

- 1. Storing a Pointer (DWORD) to a String:** In some cases, you may store a DWORD that holds the memory address where the string is located. This DWORD points to the memory location where the string characters are stored, but the characters themselves are not stored in the DWORD. This is commonly used when working with Windows API functions because they often expect string pointers as arguments.

**2. Storing the String Itself:** When you want to define and store the actual string characters, you typically use the DB directive to create a byte array with the string characters followed by a null terminator, as shown in the previous examples. This is the standard way to define strings in assembly.



So, the statement about using a DWORD to store a string in MASM may have been referring to storing the memory address of the string, not the characters themselves. Storing the characters directly in a DWORD is not a standard practice for defining strings in assembly language.

In most cases, you'll define strings using DB for character data, as it's more efficient and compatible with standard assembly conventions. Storing the address of a string in a DWORD is a different concept and is often used when working with APIs or handling memory addresses.

## *Equal-Sign Directive*

The equal-sign directive associates a symbol name with an integer expression.

```
name = expression
```

Ordinarily, expression is a 32-bit integer value. When a program is assembled, all occurrences of name are replaced by expression during the assembler's preprocessor step.

Suppose the following statement occurs near the beginning of a source code file:

```
COUNT = 500
```

Further, suppose the following statement should be found in the file 10 lines later:

```
mov eax, COUNT
```

When the file is assembled, MASM will scan the source file and produce the corresponding code lines:

```
mov eax, 500
```

## Why Use Symbols?

We might have skipped the COUNT symbol entirely and simply coded the MOV instruction with the literal 500, but experience has shown that programs are easier to read and maintain if symbols are used.

Suppose COUNT were used many times throughout a program. At a later time, we could easily redefine its value:

```
COUNT = 600
```

Assuming that the source file was assembled again, all instances of COUNT would be automatically replaced by the value 600.

## ***Current Location Counter \$***

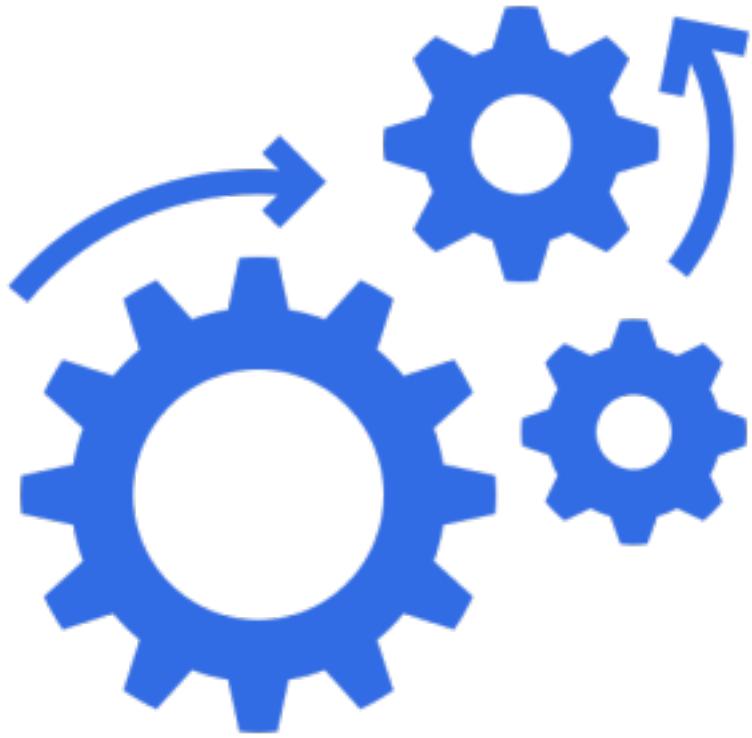
### **CURRENT LOCATION COUNTER**

The **current location counter (LC)**, also known as the assembly pointer (AP), is a special symbol or register, in assembly language that represents the current address in memory where the assembler is writing code. It is denoted by the dollar sign (\$).



The LC is used by the assembler to assign addresses to variables, labels, and instructions. It is also used by pseudo-ops, such as .data and .code, to define different sections of the program.

When the assembler starts processing a source file, the **LC is initialized to zero**. As the assembler encounters instructions and directives, it increments the LC by the length of the code or data being generated.



The LC can be used in assembly language code to refer to the current address in memory. For example, the following statement declares a variable named selfPtr and initializes it with its own offset address:

```
selfPtr DWORD $
```

This means that the variable selfPtr will contain the address of the first byte of its own definition.

Using the LC to refer to addresses in memory is a common practice in assembly language programming.

In assembly language, this statement is declaring a variable named selfPtr of type DWORD and initializing it with the special symbol \$. Here's an explanation:

1. **selfPtr**: This is the name of the variable. You can think of it as a label that represents a memory location.
2. **DWORD**: This specifies the data type of the variable. DWORD stands for Double Word, which typically represents a 32-bit value in memory. It's a common data type for integers in assembly language.
3. **\$**: The dollar sign \$ is a special symbol in assembly language. In this context, it represents the address of the first byte of the variable's own definition. So, selfPtr will contain the address of the first byte of itself.

In simpler terms, this line of code is declaring a 32-bit integer variable named selfPtr and initializing it with its own memory address. This can be useful in low-level programming when you need to work with memory addresses directly.

---

In other words, the LC is a way to keep track of where in memory the assembler is currently writing code. This is useful for initializing variables and labels to their correct addresses.

Here is a simpler explanation:

- Imagine you are writing a book. The LC is like the page number you are currently on. As you write more words, the LC increases.

- You can use the LC to refer to specific words or phrases in your book. For example, you could say "See page 10 for more information."
  - In the same way, you can use the LC in assembly language to refer to specific variables or labels. For example, you could say "Load the value of the variable selfPtr into the AL register."
- 

## **KEYBOARD DEFINITIONS**

For example, the following statement defines a label named Esc\_key and assigns it the value 27, which is the ASCII code for the Esc key:

```
Esc_key = 27
```

Later in the program, the following instruction can be used to move the value of the Esc key into the AL register:

```
mov al, Esc_key
```

This is a better style than using the integer literal 27 directly, because it makes the code more readable and maintainable.

---

## DUP OPERATOR

The DUP operator is used to create duplicate copies of a data item in memory. The counter used by DUP should be a symbolic constant, which is a constant that is identified by a name.

This makes the program easier to maintain because you can change the value of the symbolic constant in one place, and the program will automatically be updated.

Here is an example of how to use DUP to create an array of 5 DWORDs:

```
COUNT = 5  
array dword COUNT DUP(0)
```

You can also use DUP to create strings. For example, the following code will create a string containing the ASCII code for the word "Hello":

```
string db "Hello", 0
```

The 0 at the end of the string is a null terminator, which is required by many string functions.

Symbols defined with the = operator can be redefined within the same program. The following code shows how the assembler evaluates the symbol COUNT as it changes value:

```
COUNT = 5
mov al, COUNT ; AL = 5
COUNT = 10
mov al, COUNT ; AL = 10
COUNT = 100
mov al, COUNT ; AL = 100
```

The changing value of the symbol COUNT has nothing to do with the runtime execution order of the statements. Instead, the symbol changes value according to the assembler's sequential processing of the source code during the assembler's preprocessing stage.

In other words, the assembler will first read the entire source file and evaluate all of the symbols. Then, it will generate the machine code for the program. **This means that the value of a symbol can be changed anywhere in the source file**, and the assembler will still be able to generate the correct machine code.

## ***Array Size Calculation with the \$ Operator***

### ***SIZE OF AN ARRAY***

When using an array, it is often useful to know its size. This can be done by explicitly stating the size of the array in the declaration, or by letting the assembler calculate it for you.

To explicitly state the size of an array, you simply list the number of elements in the array in the declaration. For example, the following declaration declares an array of 10 bytes:

```
array BYTE 10
```

To let the assembler calculate the size of an array, you can use the \$ operator, which represents the current location counter (LC).

The **LC** is a special register that keeps track of the current address in memory where the assembler is writing code.

To calculate the size of an array using the LC, you simply subtract the offset of the array from the current LC. For example, the following declaration calculates the size of the array variable:

```
array BYTE 10, 20, 30, 40  
array_size = ($ - array)
```

The `array_size` variable will now contain the value 4, which is the number of elements in the array variable.

It is important to note that the array\_size variable must be declared immediately after the array variable. This is because the LC will be incremented by the size of the array variable when the assembler encounters it.

## ***CALCULATING THE SIZE OF A STRING***

To calculate the size of a string, you can use the same method as above. Simply subtract the offset of the string from the current LC. For example, the following declaration calculates the size of the string variable:

```
string BYTE "This is a long string, containing" BYTE "any number of characters"  
string_size = ($ - string)
```

OR

```
string BYTE "This is a long string, containing"  
          BYTE "any number of characters"  
string_size = ($ - string)
```

The string\_size variable will now contain the size of the string variable, including the null terminator.

## ***SIZE OF ARRAYS OF WORDS AND DOUBLEWORDS***

When calculating the size of an array of words or doublewords, you need to take into account the size of each element in the array. Words are 2 bytes wide, and doublewords are 4 bytes wide.

To calculate the size of an array of words, you simply divide the total array size (in bytes) by 2. For example, the following declaration calculates the size of the list variable:

```
list WORD 1000h, 2000h, 3000h, 4000h  
list_size = ($ - list) / 2
```

The list\_size variable will now contain the value 4, which is the number of elements in the list variable.

To calculate the size of an array of doublewords, you simply divide the total array size (in bytes) by 4. For example, the following declaration calculates the size of the list variable:

```
list DWORD 10000000h, 20000000h, 30000000h, 40000000h  
list_size = ($ - list) / 4
```

The list\_size variable will now contain the value 4, which is the number of elements in the list variable.

---

The image you sent shows the following text:

```
array BYTE 16
```

This is a declaration of an array named array with 16 bytes of storage. The BYTE keyword specifies that the array elements are bytes.

The word array is not a keyword in assembly language. It is simply a variable name. However, the array variable has a special meaning in assembly language.

When the assembler encounters an array variable, it allocates the specified amount of storage for the array and keeps track of the offset of the array in memory.

This means that the array variable can be used to access the array elements. For example, the following instruction will load the value of the first element of the array variable into the AL register:

```
array BYTE 16  
mov al, array[0]
```

The array[0] expression refers to the first element of the array variable. The offset of the first element is calculated by the assembler.

In summary, the array variable is not a keyword in assembly language, but it has a special meaning.

It is a variable that is used to store an array of data.

---

**What does array byte 16 mean? does it mean the first element of the array is 16 or it means am writing down a variable called array that will hold bytes, which are typically strings/string arrays?**

- The statement array BYTE 16 in assembly language means that you are declaring a variable called array that can store 16 bytes. This could be used to store a string, a byte array, or any other type of data that is 16 bytes or less in size.
  - The first element of the array variable will not necessarily be the value 16. The value of the first element will depend on what data is stored in the array. For example, if you store the string "Hello, world!" in the array variable, the first element will be the ASCII code for the letter 'H'.
  - If you store a byte array in the array variable, the first element will be the first byte in the array. To access the first element of the array variable, you would use the following expression: `array[0]`.
  - This expression will return the first byte in the array. To access the second element of the array variable, you would use the following expression: `array[1]`. This expression will return the second byte in the array. And so on.
- 

**Array BYTE 10**

**Array BYTE 10, 20, 30, 40**

## **difference??**

The difference between the two array declarations is that the first one declares an array with a single element with the value 10, while the second one declares an array with four elements, with the values 10, 20, 30, and 40.

- **Array BYTE 10** allocates space for an array that can hold up to 10 bytes, but it doesn't set specific initial values. You need to assign values to the elements of the array separately.
  - **Array BYTE 10, 20, 30, 40** declares an array and initializes it with specific values, where 10 is the first element, 20 is the second, 30 is the third, and 40 is the fourth.
- 

### **1. Array BYTE 10:**

This declaration allocates space for an array that can hold up to 10 bytes. It does not set the initial value of the first element to 10. Instead, it allocates memory for a potential array of 10 bytes but doesn't assign any specific values to those bytes. So, there is no initial value of 10 for the first element. You would need to assign values to the individual elements of the array separately if you intend to use them.

### **2. Array BYTE 10, 20, 30, 40:**

This declaration not only allocates space for an array but also initializes it with specific values. In this case, the array has four elements, and their initial values are 10, 20, 30, and 40. The 10 here is indeed the initial value of the first element.

To summarize:

- **Array BYTE 10** allocates space for an array of 10 bytes but does not assign an initial value of 10 to the first element. It's merely reserving memory.
- **Array BYTE 10, 20, 30, 40** declares an array with four elements and initializes them with the specified values, where 10 is the initial value of the first element.

The key distinction is whether you're allocating memory only or initializing values during the declaration. The 10 in the first declaration is related to the size (number of bytes) allocated, not an initial value for an element.

---

## ***EQU Directive***

The EQU directive associates a symbolic name with an integer expression or some arbitrary text.

- There are three formats:

```
name EQU expression  
name EQU symbol  
name EQU <text>
```

- When the assembler encounters name later in the program, it substitutes the integer value or text for the symbol.
- EQU can be useful when defining a value that does not evaluate to an integer, such as a real number constant or a text string.
- A symbol defined with EQU cannot be redefined in the same source code file.

Here are some examples of how to use the EQU directive:

```
; Define a symbolic constant for the value of pi.  
PI EQU <3.1416>  
  
; Define a symbolic name for a character string.  
prompt EQU <"Press any key to continue...",0>  
  
; Define a symbolic constant for the number of cells in a 10-by-10 integer matrix.  
matrixSize EQU 10 * 10  
  
; Use the symbolic constant to define a memory location for the matrix.  
matrix WORD matrixSize
```

The EQU directive can also be used to create complex expressions that are difficult to calculate by hand. For example, the following code defines a symbolic constant for the address of a memory location:

```
; Define a symbolic constant for the address of the start of the stack.  
stackStart EQU _end + 1024
```

The `_end` symbol is defined by the assembler to point to the end of the program code. The EQU directive then adds 1024 bytes to this address to define the start of the stack.

The EQU directive is a powerful tool that can make assembly language code more readable and maintainable. However, it is important to understand how it works before using it in your own code.

---

You've correctly highlighted an important distinction between the EQU and = (or ==) directives in assembly language:

### 1. EQU Directive:

When you use the EQU directive to define a symbolic constant, that constant cannot be redefined within the same source code file.

This restriction ensures that once you've associated a symbol with a specific value or text using EQU, it cannot be inadvertently changed or redefined elsewhere in the code. For example:

```
MY_CONSTANT EQU 42
; ...
MY_CONSTANT EQU 99 ; This is not allowed and will result in an error
```

---

## 2. = Directive (or == in some assemblers):

In contrast, the = directive (or == in some assemblers) allows you to redefine a symbol with a new value or expression within the same source code file.

This means you can modify the value associated with a symbol at different points in your code. For example:

```
MY_VARIABLE = 10
; ...
MY_VARIABLE = 20 ; This is allowed and changes the value associated with MY_VARIABLE
```

---

The key difference is that the **EQU directive enforces immutability**, preventing accidental changes to a symbol's value within the same source file, while the = directive allows for redefinition.

Depending on your coding requirements, you can choose the directive that suits your needs.

---

## ***TEXTEQU Directive***

### **1. TEXTEQU Directive Overview:**

The TEXTEQU directive in assembly language is used to create text macros. These text macros are like placeholders for text or expressions that you can reuse in your code. There are three common formats for the TEXTEQU directive:

- **name TEXTEQU <text>:** This format assigns name as a text macro and associates it with the specified text enclosed in angle brackets < >.
- **name TEXTEQU textmacro:** Here, name becomes a text macro, and it takes on the content of an existing text macro called textmacro.

- **name TEXTEQU %constExpr:** This format assigns name as a text macro and associates it with a constant integer expression %constExpr.

```
name TEXTEQU <text>
name TEXTEQU textmacro
name TEXTEQU %constExp
```

Let's consider an example where we create a text macro for a prompt message:

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)??">
```

Here, `continueMsg` is associated with the text "Do you wish to continue (Y/N)?". This text macro can be used to simplify the creation of prompt messages in your code.

**Building Text Macros:** Text macros can also be used to build more complex macros by combining them. For example:

```
rowSize = 5
count TEXTEQU %(rowSize * 2)
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
```

Here, count is calculated based on the value of rowSize, and move is associated with the text "mov". Then, setupAL is built using move and count. So, setupAL becomes "mov al,10" because count is calculated as rowSize \* 2.

**Redefining Text Macros:** Unlike the EQU directive, a symbol defined with TEXTEQU can be redefined at any time within the same source code file. This means you can change the text or expression associated with a text macro as needed.

In summary, the TEXTEQU directive is used to create text macros in assembly language, which are like placeholders for text or expressions that you can reuse. These macros can simplify code and make it more readable. Unlike symbols defined with EQU, text macros can be redefined in the same source file, allowing for flexibility in your code.

## ***Questions for Data Definition***

Declare a symbolic constant for the ASCII code of Backspace (08h): Using the equal-sign directive, you can declare a symbolic constant like this:

```
BackspaceKey EQU 08h
```

Declare a symbolic constant for the number of seconds in a day. To calculate the number of seconds in a 24-hour period and assign it to a symbolic constant:

```
SecondsInDay EQU 24 * 60 * 60 ; 24 hours * 60 minutes * 60 seconds
```

Calculate and assign the size of the myArray array to a symbolic constant. You can calculate the size of myArray using the sizeof operator and assign it to ArraySize:

```
myArray WORD 20 DUP(?)
ArraySize EQU sizeof myArray
```

Use TEXTEQU to redefine "proc" as "procedure". To redefine "proc" as "procedure" using a TEXTEQU

expression:

```
proc TEXTEQU <procedure>
```

Create a symbol named Sample for a string constant and use it when defining MyString. First, create the text macro for Sample, and then use it to define MyString:

```
Sample TEXTEQU <"This is a sample string">
MyString BYTE Sample
```

Use TEXTEQU to assign SetupESI to a line of code. To assign SetupESI to the line of code mov esi, OFFSET myArray:

```
SetupESI TEXTEQU <mov esi, OFFSET myArray>
```

## ***64 Bit Programming***

Make pdfs for separate modules of these notes:

Here is a summary of the differences between the 32-bit and 64-bit versions of the AddTwoSum program:

<b>ASPECT</b>	<b>32 BIT VERSION</b>	<b>64 BIT VERSION</b>
Directives	.386 , .model flat, sdcall, .stack 4096	Not used in 64 bit code
Parameter Definitions	ExitProcess PROTO, dwExitCode: DWORD	ExitProcess PROTO
Program Termination	INVOKE ExitProcess, 0	mov ecx, 0   call ExitProcess
Entry Point Specification	.start:	Not specified( main is the assumed entry point)

Example: Modifying AddTwoSum for 64-Bit:

The text mentions taking a sample program called "AddTwoSum" and modifying it for 64-bit programming. Here's a simplified version of the modifications:

- In 64-bit programming, there's no need to specify certain directives like .386, .model flat, stdcall, and .stack 4096. These were necessary in 32-bit programming but are not used in 64-bit code.
- The PROTO keyword used in 64-bit programming doesn't require parameter definitions. In contrast, 32-bit code would have specified parameters like ExitProcess PROTO, dwExitCode:DWORD. In 64-bit, you simply use ExitProcess PROTO.
- In the 64-bit version, lines 14 and 15 are used to terminate the program. They use the mov and call instructions. In 32-bit programming, an INVOKE statement might have been used for this purpose, but 64-bit MASM doesn't support INVOKE.
- Line 17 doesn't specify a program entry point in the end directive in the 64-bit version. In 32-

bit programming, an entry point might have been specified.

```
;32 bit programming
.386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
sum DWORD 0

.code
main PROC
    mov eax, 5
    add eax, 6
    mov sum, eax

    INVOKE ExitProcess, 0
main ENDP

END main
```

```
;64 bit programming
```

```
.data
```

```
sum DWORD 0
```

```
.code
```

```
main PROC
```

```
    mov eax, 5
```

```
    add eax, 6
```

```
    mov sum, eax
```

```
    mov ecx, 0
```

```
    call ExitProcess
```

```
main ENDP
```

```
ExitProcess PROTO
```

```
END
```

---

In 32-bit assembly programming, it's a common practice to declare function prototypes like `ExitProcess PROTO` even before they are used in the code for several reasons:

- 1. Linker and Library Compatibility:** The `PROTO` declaration serves as a prototype or declaration for a function to the assembler and linker. It informs them about the function's name, parameters, and

calling convention. This information is important for ensuring that the program can be linked correctly and that library functions are used in a compatible manner.

**2. Readability and Documentation:** Declaring function prototypes at the beginning of your code makes the code more readable and self-documenting. It provides a clear indication of what functions will be used in the program and what parameters they expect.

**3. Error Checking:** It allows the assembler to perform basic error checking, such as verifying that the function names and parameters used in the code match the declared prototypes. This can help catch typos and errors early in the development process.

**4. Modularity:** When working with larger programs or codebases, declaring function prototypes separately allows for modularity. You can define functions in separate source files or libraries and include their prototypes at the beginning of your main program, promoting code organization and separation of concerns.

**5. Compatibility Across Assemblers and Tools:** Using PROTO declarations is a standard and portable way of specifying function interfaces in assembly language. This practice ensures that your code can be easily understood and maintained by other assembly programmers and tools.

While it may seem a bit redundant to declare ExitProcess PROTO at the beginning of a small program like "AddTwoSum," this practice becomes more valuable in larger and more complex assembly projects where code organization and readability are critical.

-----

In addition to these specific differences, there are also some general differences between 32-bit and 64-bit assembly language programming:

- 64-bit processors have more registers than 32-bit processors, and the registers are 64 bits wide

instead of 32 bits wide.

- 64-bit processors can address more memory than 32-bit processors.
- 64-bit assembly language programs typically use different calling conventions than 32-bit assembly language programs.

Here are some additional things to keep in mind when programming for 64-bit processors:

- Use 64-bit data types and registers whenever possible.
- Be aware of the different calling conventions used by 64-bit processors.
- Test your code thoroughly on a 64-bit system before deploying it.

Overall, 64-bit programming is very similar to 32-bit programming. However, there are some important differences to be aware of.

---

```
; AddTwoSum_64b.asm
```

```
.data
```

```
sum QWORD 0
```

```
.code
```

```
main PROC
```

```
; Move the value 5 into the 64-bit register RAX.
```

```
mov rax, 5
```

```
; Add the value 6 to RAX.
```

```
add rax, 6
```

```
; Move the value of RAX to the 64-bit variable sum.
```

```
mov sum, rax
```

```
; Exit the program.
```

```
mov ecx, 0
```

```
call ExitProcess
```

```
main ENDP
```

```
END
```

This program is similar to the 32-bit version of the AddTwoSum program, but it uses 64-bit registers and variables instead. The only changes that needed to be made were:

- In the data section, the DWORD type was changed to QWORD to declare the sum variable.
- In the code section, the EAX register was changed to RAX wherever it was used.

To compile and run this program, you need to use the 64-bit version of MASM and be running the 64-bit version of Windows.

Here are some additional things to keep in mind when using 64-bit registers and variables:

- 64-bit registers and variables are twice as wide as 32-bit registers and variables, so they can store larger values.
- 64-bit registers and variables can be used to perform arithmetic on larger values, such as integers that are larger than 32 bits.
- 64-bit registers and variables can be used to access more memory, such as memory that is located above the 4 GB address limit of 32-bit processors.

Overall, using 64-bit registers and variables can make your code more efficient and capable. However, it is important to be aware of the differences between 32-bit and 64-bit programming before you start using 64-bit registers and variables.

---

The letter "R" in RAX, RBX, etc. stands for "register". These are special memory locations inside the CPU that can be used to store data and temporary results. 64-bit processors have 16 general-purpose registers, which are named RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, and R15.

Each register has a specific purpose, but they can all be used to store data and perform arithmetic

operations. For example, the RAX register is typically used to store the result of a function call. The RBX register is often used to store the base address of a data structure. And the RSP register is used to store the stack pointer.

Registers are much faster than memory, so using them can significantly improve the performance of your code. However, it is important to use them correctly, as using them incorrectly can lead to errors.

Here are some examples of how to use registers in assembly language:

; Move the value 5 into the RAX register.

```
mov rax, 5
```

; Add the value 6 to RAX.

```
add rax, 6
```

; Store the value of RAX in the memory location pointed to by RSI.

```
mov [rsi], rax
```

; Call the function `my\_function`.

```
call my_function
```

; The result of the function call is stored in RAX.

When we add an "R" to the beginning of a register name, we are converting it to its 64-bit version.

For example, the 32-bit register EAX is converted to the 64-bit register RAX.

32-bit (x86)	64-bit (x64)
EAX	RAX
EBX	RBX
ECX	RCX
EDX	RDX
ESI	RSI
EDI	RDI
ESP	RSP
EBP	RBP

64-bit registers are twice as wide as 32-bit registers, so they can store larger values. This means that we can use 64-bit registers to perform arithmetic on larger values, such as integers that are larger than 32 bits.

We can also use 64-bit registers to access more memory, such as memory that is located above the 4 GB address limit of 32-bit processors.

Overall, using 64-bit registers can make our code more efficient and capable. However, it is important to be aware of the differences between 32-bit and 64-bit programming before we start using 64-bit registers.

---

## **CONCEPTS LEARNED SO FAR:**

The chapter summary you provided is a good overview of the key concepts covered in Chapter 3 of an assembly language textbook. Here is a brief summary of the most important points:

- **Constants and expressions:** A constant is a fixed value, such as a number or a string. An expression is a combination of constants, operators, and other expressions that evaluates to a single value.
- **Identifiers and directives:** An identifier is a name that is used to identify a variable, symbolic constant, procedure, or code label. A directive is a command that is embedded in the source code and interpreted by the assembler.
- **Instructions and operands:** An instruction is a source code statement that is executed by the processor at runtime. An operand is a value that is passed to an instruction.
- **Program segments:** Assembly language programs contain logical segments named code, data, and stack. The code segment contains executable instructions. The data segment holds variables. The stack segment holds procedure parameters, local variables, and return addresses.
- **Assemblers and linkers:** An assembler is a program that reads the source file, producing both object and listing files. The linker is a program that reads one or more object files and produces an executable file.
- **Data types:** Assembly language recognizes intrinsic data types, such as BYTE, WORD, DWORD, QWORD, REAL4, REAL8, and REAL10.
- **Data definitions:** A data definition statement sets aside storage in memory for a variable, and may optionally assign it a name.
- **String data definitions:** To create a string data definition, enclose a sequence of characters in quotes.
- **The DUP operator:** The DUP operator generates a repeated storage allocation, using a constant

expression as a counter.

- **Little-endian order:** x86 processors store and retrieve data from memory using little-endian order: The least significant byte of a variable is stored at its starting (lowest) address value.
- **Symbolic constants:** A symbolic constant (or symbol definition) associates an identifier with an integer or text expression.

### 3.8.1 Terms

assembler	intrinsic data type
big endian	label
binary coded decimal (BCD)	linker
calling convention	link library
character literal	listing file
code label	little-endian order
code segment	macro
compiler	memory model
constant integer expression	memory operand
data definition statement	object file
data label	operand
data segment	operator precedence
decimal real	packed binary coded decimal
directive	process return code
encoded real	program entry point
executable file	real number literal
floating-point literal	reserved word
identifier	source file
initializer	stack segment
instruction	string literal
instruction mnemonic	symbolic constant
integer constant	system function
integer literal	

### **3.8.2 Instructions, Operators, and Directives**

+	(add, unary plus)	END
=	(assign, compare for equality)	ENDP
/	(divide)	DUP
*	(multiply)	EQU
( )	(parentheses)	MOD
-	(subtract, unary minus)	MOV
ADD		NOP
BYTE		PROC
CALL		SBYTE
.CODE		SDWORD
COMMENT		.STACK
.DATA		TEXTEQU
DWORD		

# **DATA TRANSFER**

Skip ...

## *Operand Types*

Let's clear this up first, before we move on:

The data transfer instructions are used to move data between registers, memory, and the stack. The basic addressing modes are direct, immediate, and indirect. Direct addressing is used to access data at a specific memory address. Immediate addressing is used to load a constant value into a register. Indirect addressing is used to access data at a memory address that is stored in a register.

The chapter also introduces the OFFSET, PTR, and LENGTHOF operators. The OFFSET operator is used to calculate the offset of a variable or structure member relative to the beginning of a data structure. The PTR operator is used to cast a value to a pointer type. The LENGTHOF operator is used to calculate the length of a data structure in bytes.

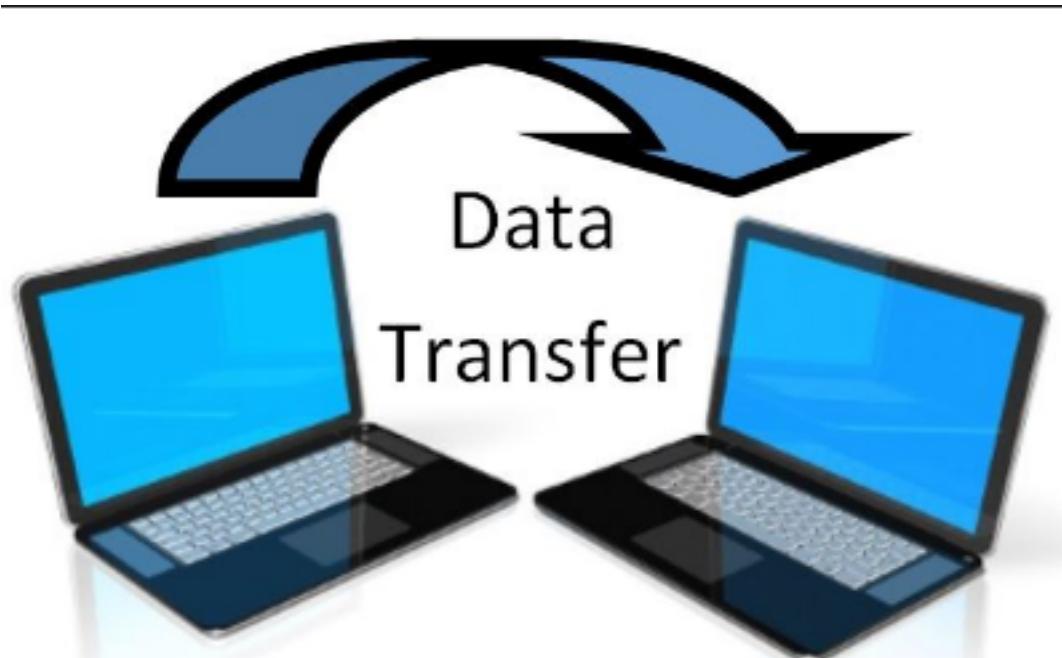
The chapter ends by discussing how to create loops and use some of the basic arithmetic operators.

Here is a summary of the key points in this chapter:

Assembly language is a low-level programming language that gives programmers direct control over the processor.

Compilers perform strict type checking, but assemblers do not. x86 processors have a complex instruction set, which means that there are many different ways to do things.

The basic **data transfer instructions** are MOV, PUSH, and POP.



The **basic addressing modes** are direct, immediate, and indirect.

## Addressing Modes in Computer Architecture



The OFFSET, PTR, and LENGTHOF operators can be used to manipulate data in memory.

## Data Manipulation



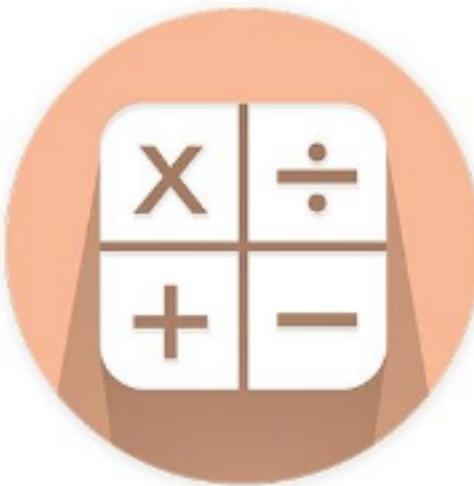
**Loops** can be created using the JMP and LOOP instructions.



The **basic arithmetic operators** are ADD, SUB, MUL, and DIV.

## Arithmetic

1 2 3  
4 5 6  
7 8 9  
0



Once you have mastered the material in this chapter, you will have a solid foundation in assembly language programming.

Three basic types of operands in x86 assembly language:

- Immediate operands are numeric literal expressions, such as 10 or -255.
- Register operands are named registers in the CPU, such as EAX or EDX.
- Memory operands reference memory locations.

The syntax of x86 instructions depends on the type of operands they use. For example, the MOV instruction can be used to move data between two registers, or to move data between a register and a memory location. The following table shows the syntax of the MOV instruction for different operand types:

Register to register  
Immediate to register  
Register to memory  
Immediate to memory

`MOV destination, source`  
`mov destination, immediate`  
`mov [memory location], source`  
`mov [memory location], immediate`

Memory operands can be specified in a variety of ways, depending on the addressing mode being used.

The three most common addressing modes are:

- **Direct addressing** specifies the memory address directly.

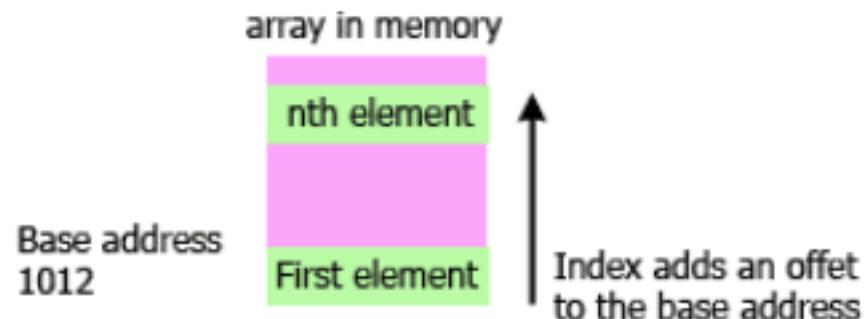


- Indirect addressing specifies the memory address indirectly, through a register.



- Indexed addressing specifies the memory address using a register as an index.

## INDEXED ADDRESSING



$$\text{Final address} = \text{base address} + \text{index}$$

The following table shows some examples of memory operands using different addressing modes:

Direct addressing [memory address]

Indirect addressing [register]

Indexed addressing [register + offset]

The **OFFSET operator** can be used to calculate the offset of a variable or structure member relative to the beginning of a data structure.

The **PTR operator** can be used to cast a value to a pointer type.

The **LENGTHOF operator** can be used to calculate the length of a data structure in bytes.

By understanding the different types of operands and addressing modes, you can write x86 assembly language code to perform a wide range of tasks.

---

Yes, when writing programs for Windows NT (and its successors), you often target a 32-bit flat address space and use a 32-bit instruction set. The use of **.MODEL FLAT** in assembly language programming and the **STDCALL** calling convention is related to this.

## **.MODEL FLAT:**

- The **.MODEL FLAT** directive is used in assembly language programming to specify that you are working in a flat memory model. In a flat memory model, you have a single, continuous address space that can be accessed using 32-bit addressing. This is in contrast to segmented memory models, where memory is divided into segments, and you have to deal with segment registers and offsets.

## **STDCALL:**

- STDCALL is a calling convention used in Windows API functions and many other Windows-related programming contexts.
- It specifies the calling convention for functions, including how parameters are passed to functions and how function results are returned.
- In the STDCALL calling convention, function parameters are typically pushed onto the stack from right to left, and the callee (the called function) is responsible for cleaning up the stack after the function call.
- The STDCALL calling convention ensures a consistent and predictable way for functions to interact, which is crucial for compatibility and interoperability in Windows programming.

So, when you see .MODEL FLAT and STDCALL in assembly code for Windows programming, it's because they are defining the memory model and calling convention that are common and expected in the Windows operating system's environment. This allows for consistent and efficient interaction between your code and the Windows API.

If you're writing 32-bit Windows programs using assembly language or working with WinAPI functions in C programming, you will encounter these conventions frequently to ensure your code can interface with the Windows operating system correctly.

## *Direct Memory Operands*

**Direct memory operands** are used to reference the contents of memory at a specific address. The address of the memory operand is specified explicitly in the instruction.

For example, the following instruction copies the value of the variable var1 into the AL register:

```
mov al, var1
```

The variable var1 is declared in the data segment as follows:

```
.data  
var1 BYTE 10h
```

This means that var1 is located at offset 10400h in memory. The mov instruction with a direct memory operand is assembled into the following machine instruction:

```
A0 00010400
```

The first byte in the machine instruction is the operation code (opcode). The remaining part is the 32-bit hexadecimal address of var1.

Direct memory operands can be used in any instruction that allows a memory operand. For example, the following instruction adds the value of the variable var1 to the EAX register:

```
add eax, var1
```

The following instruction stores the value of the EAX register in the variable var1:

```
mov var1, eax
```

Direct memory operands can also be used in arithmetic expressions. For example, the following instruction adds the value of the variable var1 to the value of the variable var2 and stores the result in the variable var3:

```
mov var3, var1 + var2
```

#### Alternative Notation

Some programmers prefer to use the following notation with direct operands:

```
mov al, [var1]
```

This notation is equivalent to the previous notation, but the brackets imply a dereference operation. The dereference operation means that the contents of the memory location at the address specified by var1 are loaded into the AL register.

MASM permits this notation, so you can use it in your own programs if you want. However, because so many programs (including those from Microsoft) are printed without the brackets, we will only use

them in this book when an arithmetic expression is involved:

```
mov al, [var1 + 5]
```

This is called a direct-offset operand, and it is discussed in more detail later.

---

var1 BYTE 10h what does that mean, it usually confuses me, coz i say it like var1 is a variable that holds a byte which is 10h in decimal will be a value like say 16. can var1 be confused with something like var1 is a byte that holds a string inside 10h? or 10h is a hex, and strings can't be in hex?

The declaration var1 BYTE 10h means that var1 is a variable of type BYTE that is initialized to the value 10h. BYTE is a data type that can store a single byte of data. 10h is a hexadecimal value that is equal to 16 in decimal.

Strings cannot be stored in hexadecimal. Strings are typically stored in ASCII or Unicode encoding. ASCII and Unicode are character encodings that assign a unique number to each character in the alphabet. For example, the ASCII code for the letter "A" is 65.  
So, to answer your questions:

- Yes, var1 is a variable that holds a byte which is 10h in decimal, which is a value of 16.
- No, var1 cannot be confused with something like a byte that holds a string inside 10h. Strings cannot be stored in hexadecimal.
- Yes, 10h is a hexadecimal value.

---

So when i see any hex value like var2 byte 10000438h what will that mean? it can mean var2 is holding a decimal or a memory location? or is byte small? Let's say var2 dword 10248132048h?

When you see a hexadecimal value in assembly language, such as 10000438h, it can mean either a decimal value or a memory location. It depends on the context in which the hexadecimal value is used.

For example, the following instruction copies the hexadecimal value 10000438h into the AL register:

```
mov al, 10000438h
```

In this case, the hexadecimal value is being used as a decimal value. The following instruction copies the contents of the memory location at address 10000438h into the EAX register:

```
mov eax, [10000438h]
```

In this case, the hexadecimal value is being used as a memory location.

To determine whether a hexadecimal value is being used as a decimal value or a memory location, you need to look at the context in which the hexadecimal value is used.

If the hexadecimal value is being used as an operand in an arithmetic instruction, then it is most likely being used as a decimal value.

If the hexadecimal value is being used as an operand in a memory instruction, then it is most likely being used as a memory location.

Byte is a data type that can store a single byte of data. Dword is a data type that can store a double word, which is four bytes of data.

The variable declaration var2 dword 10248132048h means that var2 is a variable of type DWORD that is initialized to the hexadecimal value 10248132048h.

This value is equal to 10248132048 in decimal.

So, to answer your questions:

- Yes, a hexadecimal value can mean that a variable is holding a decimal value or a memory location. It depends on the context in which the hexadecimal value is used.
  - Yes, a hexadecimal value can be used to represent a memory location.
  - The data type BYTE is smaller than the data type DWORD. A BYTE can store a single byte of data, while a DWORD can store four bytes of data.
- 

So, if a hex is inside [] dereferencing its a memory location? Or it can also be just 28323428932h without brackets, but it's still a memory location?

Yes, if a hexadecimal value is inside square brackets ([]), it is a dereferenced memory location.

This means that the contents of the memory location at the address specified by the hexadecimal value are loaded into the register or memory operand that the dereference operator is applied to.

For example, the following instruction loads the contents of the memory location at address 28323428932h into the EAX register:

```
mov eax, [28323428932h]
```

However, if a hexadecimal value is not inside square brackets, it can be either a decimal value or a memory location, depending on the context in which it is used. For example, the following instruction copies the hexadecimal value 28323428932h into the EAX register:

```
mov eax, 28323428932h
```

In this case, the hexadecimal value is being used as a decimal value.

To determine whether a hexadecimal value is being used as a decimal value or a memory location, you need to look at the context in which the hexadecimal value is used.

If the hexadecimal value is being used as an operand in an arithmetic instruction, then it is most likely being used as a decimal value.

If the hexadecimal value is being used as an operand in a memory instruction, then it is most likely being used as a memory location.

**Note:** It is generally considered **good practice to use square brackets** around hexadecimal values that are being used as memory locations, even if they are not strictly necessary. This helps to make the code more readable and maintainable.

Some programmers prefer to use square brackets [ ] to indicate memory operands, as in `mov al,[var1]`. This notation explicitly shows that the instruction involves memory access. It's useful when you need to work with memory addresses directly or when performing more complex operations that include offsets, as in `mov al,[var1 + 5]`.

---

- **Opcode:** It is the operation code that specifies the operation or instruction to be executed by the CPU. It tells the CPU what to do.
- **Operand:** It is the data or value on which the operation specified by the opcode is performed. It tells the CPU what to do the operation on.

**;This example uses the MOV and ADD opcodes to add two numbers**

```
mov eax, 5      ; Load 5 into the EAX register
add eax, 3      ; Add 3 to the value in EAX
```

**;After execution, EAX will contain 8 (5 + 3)**

```
;This example uses operands to perform addition  
  
mov ebx, 7      ; Load 7 into the EBX register  
mov ecx, 9      ; Load 9 into the ECX register  
add ebx, ecx    ; Add the value in ECX to EBX  
  
; After execution, EBX will contain 16 (7 + 9)
```

In this example, ebx and ecx are operands. add is the opcode(operation code), and the operands(data being worked on) are the values in ebx and ecx that are being added together.

## ***MOV Operation/Opcode***

It copies data from a source operand to a destination operand.

```
MOV destination, source
```

; or

```
mov destination, source
```

The **destination operand** is the operand that will be modified by the instruction.

The **source operand** is the operand that will be copied to the destination operand.

Both the source operand and the destination operand must be the **same size**.

## Similar size

The MOV instruction cannot be used to move data directly from one memory location to another.

Instead, you must move the source operand's value to a register before assigning its value to a memory operand.

---

The following example shows how to use the MOV instruction to move data from one memory location to

another:

```
mov destination, source

.data
    var1 WORD ?
    var2 WORD ?

.code
    mov ax,var1    ;move data into registers first
    mov var2,ax
```

This code will copy the value of the var1 variable to the var2 variable.

---

The following example shows how to use the MOV instruction to move data from a register to a memory location:

```
;Register to memory location
```

```
.data
```

```
    var3 DWORD ?
```

```
.code
```

```
    mov eax,1024  
    mov var3,eax
```

This code will store the value 1024 in the var3 variable. Yes, var3 the memory location.

The MOV instruction copies the contents of the EAX register to the memory location at address var3.

In assembly language, variables are simply names for memory locations. When you declare a variable, such as var3, the assembler sets aside a certain amount of memory for the variable and assigns it a name. You can then use the variable name in your code to access the memory location.

In the above example, the VAR3 variable is declared as a DWORD variable. This means that the variable occupies four bytes of memory. The MOV instruction copies the four bytes of data from the EAX register to the four bytes of memory at address var3.

Once the MOV instruction is executed, the var3 variable will contain the value 1024. You can then use the var3 variable in your code to access the value 1024.

---

The following example shows how to use the MOV instruction to move data from a memory location to a register:

```
;Memory location to register
```

```
.data  
var4 DWORD 1024  
  
.code  
    mov eax, var4
```

This code will load the value of the var4 variable into the EAX register.

The MOV instruction is a very versatile instruction that can be used to move data between registers, memory locations, and immediate values. It is one of the most important instructions to learn in assembly language.

---

```
;Summary of the mov  
mov reg,reg  
mov mem,reg  
mov reg,mem  
mov mem,imm  
MOV reg,imm      ;capitalizing MOV is optional
```

## *Overlapping Values*

The provided code example demonstrates how differently sized data can overlap and affect the values stored in a 32-bit register (EAX) in x86 assembly language. Let's break down the code and understand how each instruction affects the register's contents:

```
.data  
    oneByte BYTE 78h  
    oneWord WORD 1234h  
    oneDword DWORD 12345678h
```

In the data section, three variables are declared: oneByte as a single byte, oneWord as a 16-bit word, and oneDword as a 32-bit double word.

Now, let's go through the code:

```
mov eax, 0
```

Sets the entire 32-bit EAX register to zero.

Result: EAX = 00000000h

```
*****
```

```
mov al, oneByte
```

Moves the value of oneByte (78h) into the low byte of EAX (AL), effectively overwriting the lowest 8 bits of EAX.

Result: EAX = 00000078h

\*\*\*\*\*

**mov ax, oneWord**

Moves the value of oneWord (1234h) into the lower 16 bits of EAX (AX), effectively overwriting the lower 16 bits of EAX.

Result: EAX = 00001234h

\*\*\*\*\*

**mov eax, oneDword**

Moves the value of oneDword (12345678h) into the entire EAX register, replacing its previous value.

Result: EAX = 12345678h

\*\*\*\*\*

```
mov ax, 0
```

Sets the lower 16 bits of EAX (AX) to zero, leaving the upper 16 bits unchanged.

Result: EAX = 12340000h

\*\*\*\*\*

The key takeaway from this code is that when you move data of different sizes into a larger register, the smaller data gets placed in the lower portion of the larger register while leaving the higher bits unaffected.

In this case, AL represents the lowest byte, AX represents the lowest two bytes, and EAX represents the entire 32-bit register. It's important to be mindful of data size when performing such operations to avoid unintended side effects.

## *Sign and Zero Extension*

Table 4-1 Instruction Operand Notation, 32-Bit Mode.

Operand	Description
<i>reg8</i>	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
<i>reg16</i>	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
<i>reg32</i>	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
<i>reg</i>	Any general-purpose register
<i>sreg</i>	16-bit segment register: CS, DS, SS, ES, FS, GS
<i>imm</i>	8-, 16-, or 32-bit immediate value
<i>imm8</i>	8-bit immediate byte value
<i>imm16</i>	16-bit immediate word value
<i>imm32</i>	32-bit immediate doubleword value
<i>reg/mem8</i>	8-bit operand, which can be an 8-bit general register or memory byte
<i>reg/mem16</i>	16-bit operand, which can be a 16-bit general register or memory word
<i>reg/mem32</i>	32-bit operand, which can be a 32-bit general register or memory doubleword
<i>mem</i>	An 8-, 16-, or 32-bit memory operand

I'll explain zero and sign extension in the context of moving smaller values to larger registers and provide simple assembly code examples.

## ZERO EXTENSION:

Zero extension is a technique used when you want to copy a smaller unsigned value (like an unsigned integer) to a larger register.

To do this, you set the larger register to zero and then move the smaller value into the appropriate portion of the larger register.

Here's an example in simple assembly code:

```
.data
    count WORD 1      ; A 16-bit unsigned integer

.code
    mov ecx, 0        ; Set ECX to zero
    mov cx, count     ; Move the 16-bit value 'count' to the lower 16 bits of ECX
```

In this example, we set ECX to all ones in 32 bits, which represents a negative zero. Then, we move the 16-bit signed value signedVal to CX, which sign-extends it to fill the upper 16 bits of ECX while preserving the negative sign.

-----

We start with count declared as a 16-bit unsigned integer with the value 1 (hex: 0001h).

`mov ecx, 0` sets the entire 32-bit ECX register to zero:

**ECX = 00000000h**

`mov cx, count` moves the 16-bit value from count into the lower 16 bits of ECX, leaving the upper 16 bits as zero:

**ECX = 00010000h**

Here's a step-by-step explanation of what's happening:

- `mov cx, count` copies the value from count (16 bits) into the lower 16 bits of ECX. So, count's value, `0001h`, is placed in the lower 16 bits of ECX.
- The upper 16 bits of ECX remain zero because we only moved a 16-bit value. So, the final value of ECX is `00010000h`.

In simple math terms, we've copied the number 1 (hex: `0001h`) into ECX and left-padded it with zeroes in the upper 16 bits. This results in `00010000h`, which is 16 in decimal. So, after executing the code, the value in ECX is 16.

---

## ***MOVZX and MOVSX Instructions***

```
MOVZX reg32,reg/mem8  
MOVZX reg32,reg/mem16  
MOVZX reg16,reg/mem8
```

While the manual methods described above work, Intel processors provide dedicated instructions for these operations:

- MOVZX (Move with Zero Extension) for unsigned values.
- MOVSX (Move with Sign Extension) for signed values.

Here's how you would use these instructions:

```
.data  
    count WORD 1      ; A 16-bit unsigned integer  
    signedVal SWORD -16 ; A 16-bit signed integer with a value of -16 (hex: FFF0)  
  
.code  
    movzx ecx, word ptr [count] ; Copy 'count' to ECX with zero extension  
    movsx ecx, word ptr [signedVal] ; Copy 'signedVal' to ECX with sign extension
```

These instructions automatically perform zero or sign extension as needed, making your code more concise and efficient.

concise and efficient.

---

```
MOVSX reg32,reg/mem8  
MOVSX reg32,reg/mem16  
MOVSX reg16,reg/mem8
```

MOVZX is used when you want to copy a smaller unsigned value (like an unsigned integer) to a larger register, extending the value with zeroes.

```
.data  
    count WORD 1      ; A 16-bit unsigned integer  
  
.code  
    mov ecx, 0          ; Set ECX to zero  
    movzx cx, count ; Move 'count' to CX with zero extension
```

`mov ecx, 0` sets ECX to zero.

`ECX = 00000000h`

`movzx cx, count` moves the 16-bit value from count into the lower 16 bits of CX with zero extension. The upper 16 bits remain zero:

**CX = 0001h**

Here, MOVZX explicitly extends the value of count with zero bits, making sure that the upper bits are set to zero.

---

-----

MOVSX is used when you want to copy a smaller signed value (like a signed integer) to a larger register, extending the value while preserving its sign.

Example using MOVSX:

```
.data
    signedVal SWORD -16      ; A 16-bit signed integer with a value of -16 (hex: FFF0)

.code
    mov ecx, 0xFFFFFFFFh    ; Set ECX to all ones in 32 bits (negative zero)
    movsx cx, signedVal     ; Move 'signedVal' to CX with sign extension
```

`mov ecx, 0xFFFFFFFFh` sets ECX to all ones in 32 bits, representing negative zero:

**ECX** = FFFFFFFFh

`movsx cx, signedVal` moves the 16-bit signed value from `signedVal` to CX with sign extension. The upper bits are filled with the sign bit, maintaining the negative sign:

**CX** = FFF0h

In this case, MOVSX sign-extends the value of `signedVal` to fill the upper bits while preserving its negative sign.

These instructions (MOVZX and MOVSX) are useful because they handle the extension automatically, simplifying code and ensuring correct behavior when working with smaller values in larger registers.

---

**In the example of movsx, must signed values be negative?**

No, signed values don't have to be negative for MOVSX (Move with Sign Extension) to work.

MOVSX is used to sign-extend signed values, which means it preserves the sign of the value while extending it to fill a larger register. Whether the value is positive or negative, MOVSX will correctly perform the sign extension.

For example, let's say you have a 16-bit signed value that is positive:

```
.data
    signedVal SWORD 42      ; A 16-bit signed integer with a value of 42

; You can use MOVSX with this positive value, and it will correctly sign-extend it:

.code
    mov ecx, 0xFFFFFFFFh    ; Set ECX to all ones in 32 bits (negative zero)
    movsx cx, signedVal     ; Move 'signedVal' to CX with sign extension
```

In this case, MOVSX will extend the positive value 42 to 0036h while preserving its positive sign.

---

## **EXPLANATION 1:**

Here, `signedVal` is declared as a 16-bit signed integer with a value of 42. It's important to note that 42 is a positive value. Now, let's look at the code:

In this code, we first set ECX to all ones in 32 bits (`0xFFFFFFFFh`), which represents negative zero in two's complement notation.

Then, we use MOVSX to move the value of `signedVal` (which is 42) into the lower 16 bits of CX while performing sign extension. Here's what happens step by step:

`mov ecx, 0xFFFFFFFFh` sets ECX to all ones in 32 bits, representing negative zero:

**ECX = FFFFFFFFh**

`movsx cx, signedVal` moves the signed value 42 (hex: 002Ah) from `signedVal` to the lower 16 bits of CX while performing sign extension.

The sign bit of the source operand (`signedVal`) is preserved in the upper bits of CX. Since 42 is positive, the sign bit is 0, and the upper bits are filled with 0s:

**CX = 002Ah**

So, even though we started with a positive value (42), when using `MOVSX`, the sign extension ensures that the value in CX is correctly represented as 002Ah in 16 bits.

The sign bit is preserved, and the upper bits are filled accordingly.

So, `MOVSX` is used for both positive and negative signed values, ensuring that the sign is correctly preserved during the extension.

---

**REPEAT EXPLANATION:**

When we perform sign extension with MOVSX, the goal is to extend the sign bit of the source value to fill the upper bits of the destination register while preserving the value's sign. Here's a step-by-step explanation for the code:

```
.data  
    signedVal SWORD 42      ; A 16-bit signed integer with a value of 42
```

signedVal is declared as a 16-bit signed integer with the **decimal value 42**, which in **hexadecimal** is **002Ah**. It's important to note that 42 is a **positive value**.

```
.code  
    mov ecx, 0xFFFFFFFFh    ; Set ECX to all ones in 32 bits (negative zero)  
    movsx cx, signedVal     ; Move 'signedVal' to CX with sign extension
```

sets ECX to all ones in 32 bits, representing negative zero in two's complement notation:

```
    mov ecx, 0xFFFFFFFFh
```

movsx cx, signedVal moves the signed value 42 (hex: 002Ah) from signedVal to the lower 16 bits of CX while performing sign extension. The sign bit of the source operand (signedVal) is preserved in the upper bits of CX. Since 42 is a positive value, its sign bit is 0, and the upper bits of CX are indeed filled with 0s:

**CX = 002Ah**

In this case, the sign extension correctly preserves the positive sign of 42, and the upper bits are filled with 0s to maintain a positive value in CX.

---

### **MORE DETAIL:**

Then why do we need movzx if movsx can do both?

That's a great question! While both MOVZX (Move with Zero Extension) and MOVSX (Move with Sign Extension) can extend the value of a smaller operand to a larger one, they serve different purposes and have distinct use cases:

#### **MOVZX (Move with Zero Extension):**

- Purpose: MOVZX is primarily used when you want to extend an unsigned value (like an unsigned integer) to a larger register, and you want the upper bits to be set to zero.
- Example: When you have an unsigned value count and you want to extend it to a larger register, you can use MOVZX to ensure that the upper bits are zero-filled. This is useful when you don't want any sign-related information.

#### **MOVSX (Move with Sign Extension):**

- Purpose: MOVSX is used when you want to extend a signed value (like a signed integer) to a larger

register, and you want to preserve the sign of the original value.

- Example: When you have a signed value `signedVal` and you want to extend it to a larger register, you can use `MOVsx` to maintain the sign bit in the upper bits. This is essential when dealing with signed data to ensure that the sign is correctly preserved.

**In summary, the choice between `MOVzx` and `MOVsx` depends on the nature of the data you're working with:**

- Use `MOVzx` when dealing with unsigned values and you want to ensure that the upper bits are zero.
- Use `MOVsx` when dealing with signed values and you want to preserve the sign while extending the value.

Each instruction is designed to handle these scenarios correctly and efficiently, making it easier to work with different types of data in assembly language.

---

When you're performing sign extension with instructions like `MOVsx`, you need to consider the sign of the number in hexadecimal representation.

Here's a simple guideline:

- If the number in hexadecimal representation is positive (e.g., `002Ah` for 42), the sign bit is 0. In this case, extending it with zeros in the upper bits preserves the positive sign.
- If the number in hexadecimal representation is negative (e.g., `FFFAh` for -6), the sign bit is 1. In this case, extending it with ones in the upper bits preserves the negative sign.

By extending the value with the appropriate sign bit, you ensure that the sign is correctly preserved when moving or copying the value to a larger register. This is crucial when working with signed integers in assembly language.

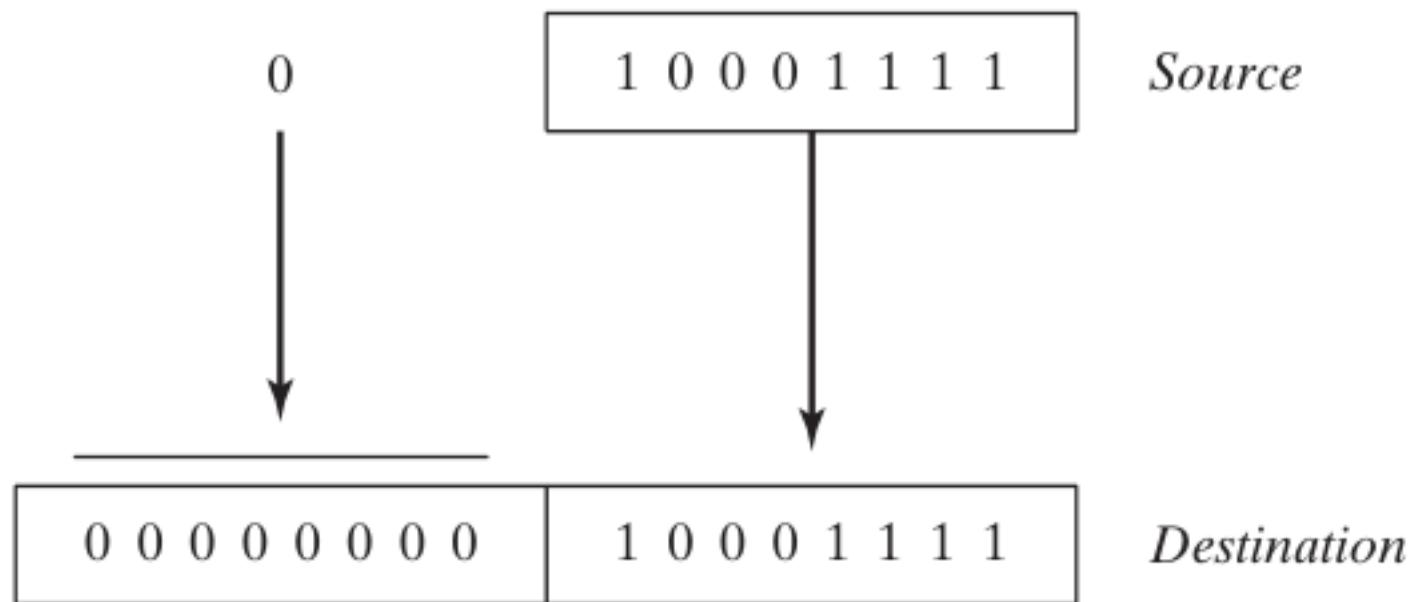
```
*****  
*****
```

### *movzx repeated 3:*

```
.data  
byteVal BYTE 10001111b      ;decimal 143  
  
.code  
movzx ax,byteVal           ; AX = 0000000010001111b
```

What is happening here:

FIGURE 4–1 Using MOVZX to copy a byte into a 16-bit destination.



MOVZX (Move with Zero Extension) is used to copy a smaller operand into a larger one while zero-filling the upper bits of the larger operand. Let's go through the examples step by step:

### Example 1: Register Operands

```
mov bx, 0A69Bh      ; BX = 0A69Bh
movzx eax, bx        ; EAX = 0000A69Bh
movzx edx, bl        ; EDX = 0000009Bh
movzx cx, bl         ; CX = 009Bh
```

In this example, we use the MOVZX instruction to move data from smaller registers (bl and bx) into larger ones (eax, edx, and cx).

1. mov bx, 0A69Bh loads the 16-bit value 0A69Bh into the bx register.
2. movzx eax, bx extends the 16-bit value in bx into a 32-bit value in eax, zero-filling the upper 16 bits. So, EAX becomes 0000A69Bh.
3. movzx edx, bl extends the 8-bit value in the lower byte of bx (bl) into a 32-bit value in edx, zero-filling the upper 24 bits. EDX becomes 0000009Bh.
4. movzx cx, bl extends the 8-bit value in bl into a 16-bit value in cx, zero-filling the upper 8 bits. CX becomes 009Bh.

## Example 2: Memory Operands

```
.data
    byte1 BYTE 9Bh
    word1 WORD 0A69Bh

.code
    movzx eax, word1      ; EAX = 0000A69Bh
    movzx edx, byte1      ; EDX = 0000009Bh
    movzx cx, byte1       ; CX = 009Bh
```

In this example, we use memory operands (variables byte1 and word1) as the sources for MOVZX.

1. movzx eax, word1 extends the 16-bit value stored in word1 into a 32-bit value in eax, zero-filling the upper 16 bits. So, EAX becomes 0000A69Bh.
2. movzx edx, byte1 extends the 8-bit value stored in byte1 into a 32-bit value in edx, zero-filling the upper 24 bits. EDX becomes 00000009Bh.
3. movzx cx, byte1 extends the 8-bit value stored in byte1 into a 16-bit value in cx, zero-filling the upper 8 bits. CX becomes 009Bh.

In both examples, MOVZX ensures that the larger registers or memory locations are filled with zeros in the upper bits, maintaining the integrity of the original value and preventing unintended sign extension.

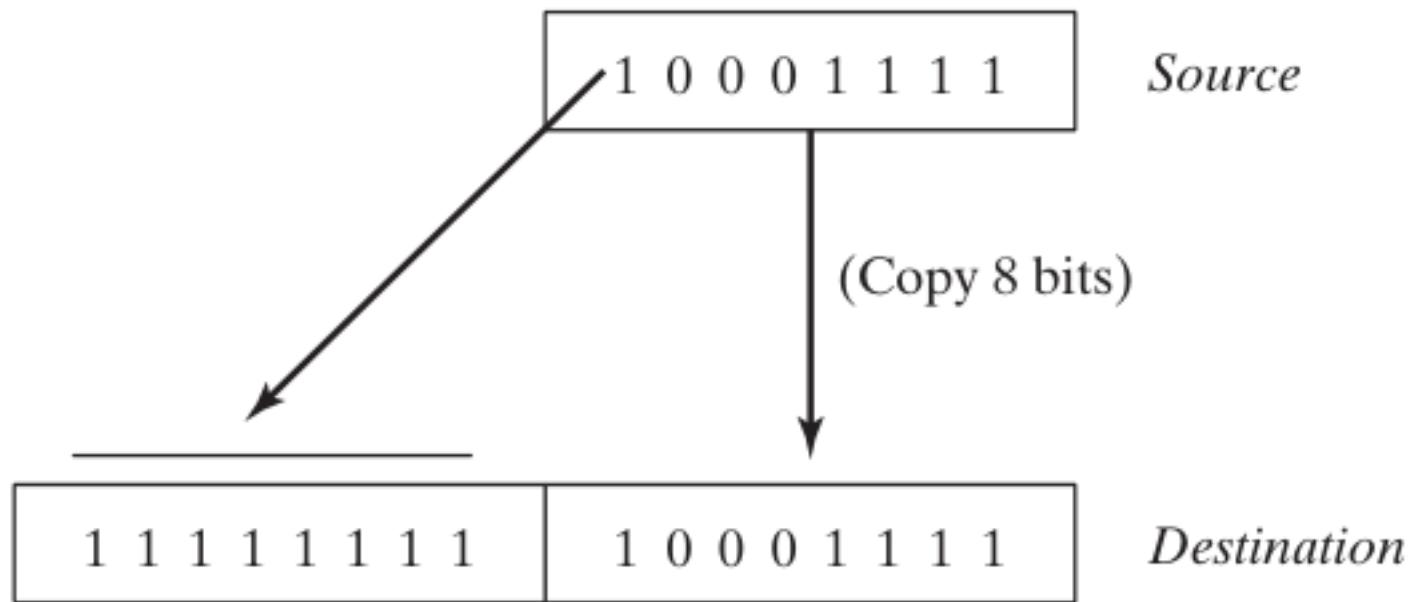
```
*****  
*****
```

### *movsx repeated 3:*

```
;MOVSX  
  
mov bx, 0A69Bh      ; BX = 0A69Bh  
movsx eax, bx        ; EAX = FFFFA69Bh  
movsx edx, bl        ; EDX = FFFFFFF9Bh  
movsx cx, bl         ; CX = FF9Bh
```

What is happening here?

FIGURE 4–2 Using MOVSX to copy a byte into a 16-bit destination.



In these examples, you're dealing with hexadecimal constants and sign extension. Here's what happens step by step:

1. `mov bx, 0A69Bh` loads the 16-bit hexadecimal value `0A69Bh` into the `bx` register. The leading "A" digit indicates that the highest bit (sign bit) is set.
2. `movsx eax, bx` extends the 16-bit value in `bx` into a 32-bit value in `eax`, while preserving the sign bit. Since the sign bit is set (due to the leading "A" digit), sign extension fills the upper 16 bits with ones (F in hexadecimal), resulting in `EAX = FFFFA69Bh`.

3. `movsx edx, bl` extends the 8-bit value in the lower byte of `bx` (`bl`) into a 32-bit value in `edx`, while preserving the sign bit. Again, the sign bit is set, so the upper 24 bits are filled with ones, resulting in `EDX = FFFFFFF9Bh`.

4. `movsx cx, bl` extends the 8-bit value in `bl` into a 16-bit value in , while preserving the sign bit. The sign bit is still set, so the upper 8 bits are filled with ones, resulting in `CX = FF9Bh`.

In these examples, the "A" in the hexadecimal constant `0A69Bh` indicates that it's a negative value due to the set sign bit. Sign extension is applied accordingly to maintain the correct sign and value representation in the larger registers.

## ***LAHF and SAHF***

The **LAHF** (Load AH from Flags) and **SAHF** (Store AH into Flags) instructions are used to manipulate and transfer specific flag values in the x86 assembly language. Here's an explanation of how these instructions work:

---

### ***LAHF (Load AH from Flags):***

The LAHF instruction loads the low byte of the EFLAGS register into the AH register.

The EFLAGS register contains various flags that indicate the status of the CPU after certain operations, such as the Sign, Zero, Auxiliary Carry, Parity, and Carry flags.

Here's an example of how to use LAHF to save a copy of these flags in a variable:

```
.data
    saverflags BYTE ?      ;Define a variable to store the flags

.code
    lahf                  ;load flags into AH
    mov saverflags, ah    ;save them in the 'saverflag' variable
```

In this example, after executing LAHF, the AH register contains the values of the specified flags,

and you can save these values in the `saveflags` variable for future reference.

---

## ***SAHF (Store AH into Flags):***

The SAHF instruction works in the opposite direction. It copies the value from the AH register into the low byte of the EFLAGS (or RFLAGS) register.

This allows you to restore saved flag values.

Here's an example of how to use SAHF to retrieve saved flag values from a variable:

```
.data
    saveflags BYTE ? ; Variable containing saved flags

.code
    mov ah, saveflags ; Load saved flags into AH
    sahf              ; Copy AH into the Flags register
```

---

The EFLAGS register, which is also known as the RFLAGS register in 64-bit x86 architectures, is 32 bits (4 bytes) in size.

It stores various CPU status flags and control bits that reflect the outcome of instructions and

affect the operation of the processor.

These flags include the Carry Flag (CF), Zero Flag (ZF), Sign Flag (SF), Overflow Flag (OF), and many others that help control and track program execution and results.

In 64-bit mode, the RFLAGS register serves a similar purpose but is still 64 bits (8 bytes) wide.

## **XCHG Instruction**

The **XCHG** (Exchange) instruction is used to exchange the contents of two operands. There are three variants of the XCHG instruction:

1. **XCHG reg, reg:** Exchanges the contents of two registers.
2. **XCHG reg, mem:** Exchanges the contents of a register with the contents of a memory location.
3. **XCHG mem, reg:** Exchanges the contents of a memory location with the contents of a register.

Here's a brief explanation and examples of each variant:

---

**XCHG reg, reg:** This variant exchanges the contents of two registers. Example:

**xchg ax, bx ; Exchange the contents of AX and BX**

After this instruction, the values in AX and BX will be swapped.

---

**XCHG reg, mem:** This variant exchanges the contents of a register with the contents of a memory location. Example:

**xchg var1, bx ; Exchange the contents of memory location 'var1' with BX**

This instruction will exchange the value in memory location 'var1' with the value in BX.

---

**XCHG mem, reg:** This variant exchanges the contents of a memory location with the contents of a register. Example:

**mov ax, val1 ; Load the value from 'val1' into AX  
xchg ax, val2 ; Exchange the contents of 'val2' with AX**

Here, we use a temporary register (AX in this case) to exchange the contents of two memory locations, val1 and val2.

The XCHG instruction is particularly useful in various situations, such as swapping values, implementing synchronization mechanisms, and array sorting algorithms, as mentioned in the provided example. It provides a simple way to exchange data between registers and memory locations.

## *Direct-Offset operands*

**Direct-offset operands** are used to access memory locations with specific offsets from a variable's base address.

Direct-offset operands allow you to access memory locations in an array or structure by adding a displacement (offset) to the base variable's address.

```
arrayB BYTE 10h, 20h, 30h, 40h, 50h
mov al, arrayB           ;al = 10h
mov al, [arrayB+1]       ;al = 20h
mov al, [arrayB+2]       ;al = 30h
```

An expression such as `arrayB+1` produces what is called an **effective address** by adding a constant to the variable's offset.

Surrounding an effective address with brackets makes it clear that the expression is dereferenced to obtain the contents of memory at the address.

The assembler does not require you to surround address expressions with brackets, but we highly recommend their use for clarity.

MASM has no built-in range checking for effective addresses.

In the following example, assuming arrayB holds five bytes, the instruction retrieves a byte of memory outside the array. The result is a sneaky logic bug, so be extra careful when checking array references:

```
mov al, [arrayB+20] ;al = retrieving memory outside the array
```

---

## ***Word and Doubleword Arrays***

When accessing word and doubleword arrays, you need to take into account the size of the array elements. For example, the following code shows how to access the second element in an array of 16-bit words:

```
.data  
    arrayW WORD 100h,200h,300h  
.code  
    mov ax,[arrayW+2] ; AX = 200h
```

The [arrayW+2] operand tells the assembler to add 2 bytes to the offset of the arrayW variable and then access the memory location at that address. This is because each element in the arrayW array is 2 bytes long.

Similarly, the following code shows how to access the second element in an array of doublewords:

```
.data  
    arrayD DWORD 10000h,20000h  
.code  
    mov eax,[arrayD+4] ; EAX = 20000h
```

The [arrayD+4] operand tells the assembler to add 4 bytes to the offset of the arrayD variable and then access the memory location at that address. This is because each element in the arrayD array is 4 bytes long.

It is important to note that MASM does not perform range checking on direct-offset operands. This means that it is possible to **accidentally access memory locations outside of the program's address space**. It is important to be careful when using direct-offset operands and to make sure that you are always accessing valid memory locations.

```
.386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
    val1 WORD 1000h
    val2 WORD 2000h
    arrayB BYTE 10h,20h,30h,40h,50h
    arrayW WORD 100h,200h,300h
    arrayD DWORD 10000h,20000h

.code
main PROC
; Demonstrating MOVZX instruction:
    mov bx, 0A69Bh
    movzx eax, bx ; EAX = 0000A69Bh
    movzx edx, bl ; EDX = 0000009Bh
    movzx cx, bl ; CX = 009Bh

; Demonstrating MOVSX instruction:
    mov bx, 0A69Bh
    movsx eax, bx ; EAX = FFFFA69Bh
    movsx edx, bl ; EDX = FFFFFFF9Bh
    mov bl, 7Bh
    movsx cx, bl ; CX = 007Bh
```

```
; Memory-to-memory exchange:  
mov ax, val1 ; AX = 1000h  
xchg ax, val2 ; AX=2000h, val2=1000h  
mov val1, ax ; val1 = 2000h  
  
; Direct-Offset Addressing (byte array):  
mov al, arrayB ; AL = 10h  
mov al, [arrayB+1] ; AL = 20h  
mov al, [arrayB+2] ; AL = 30h  
  
; Direct-Offset Addressing (word array):  
mov ax, arrayW ; AX = 100h  
mov ax, [arrayW+2] ; AX = 200h  
  
; Direct-Offset Addressing (doubleword array)  
mov eax, arrayD ; EAX = 10000h  
mov eax, [arrayD+4] ; EAX = 20000h  
mov eax, [arrayD+4] ; EAX = 20000h
```

```
INVOKE ExitProcess, 0  
main ENDP  
END main
```

Repeat of the code:

```
.386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
    val1 WORD 1000h
    val2 WORD 2000h
    arrayB BYTE 10h,20h,30h,40h,50h
    arrayW WORD 100h,200h,300h
    arrayD DWORD 10000h,20000h

.code
main PROC
; Demonstrating MOVZX instruction:
    mov bx, 0A69Bh
    movzx eax, bx ; EAX = 0000A69Bh
    movzx edx, bl ; EDX = 00000009Bh
    movzx cx, bl ; CX = 009Bh

; Demonstrating MOVSX instruction:
    mov bx, 0A69Bh
    movsx eax, bx ; EAX = FFFFA69Bh
    movsx edx, bl ; EDX = FFFFFFF9Bh
    mov bl, 7Bh
    movsx cx, bl ; CX = 007Bh

; Memory-to-memory exchange:
    mov ax, val1 ; AX = 1000h
    xchg ax, val2 ; AX=2000h, val2=1000h
    mov val1, ax ; val1 = 2000h

; Direct-Offset Addressing (byte array):
```

```

mov al, arrayB ; AL = 10h
mov al, [arrayB+1] ; AL = 20h
mov al, [arrayB+2] ; AL = 30h

; Direct-Offset Addressing (word array):
mov ax, arrayW ; AX = 100h
mov ax, [arrayW+2] ; AX = 200h

; Direct-Offset Addressing (doubleword array):
mov eax, arrayD ; EAX = 10000h
mov eax, [arrayD+4] ; EAX = 20000h
mov eax, [arrayD+4] ; EAX = 20000h

INVOKE ExitProcess, 0
main ENDP
END main

```

Of course, here's a detailed explanation of the key concepts and examples from the provided assembly code:

#### **MOVZX Instruction (Zero Extension):**

- The MOVZX instruction is used to zero-extend values from smaller registers or memory locations into larger ones.
- For example, it can extend a byte-sized value to a word or dword-sized value, filling the higher bits with zeros.

#### **MOVSX Instruction (Sign Extension):**

- The MOVSX instruction is used to sign-extend values from smaller registers or memory locations into larger ones.
- It extends a byte or word-sized value to a larger size while preserving the sign bit. If the

original value is positive, it fills the higher bits with zeros; if negative, it fills them with ones.

#### **Memory-to-Memory Exchange (XCHG):**

- The XCHG instruction swaps the contents of two operands, whether they are registers or memory locations.
- In the provided code, it's used to swap the values of two variables, val1 and val2.

#### **Direct-Offset Addressing:**

- Direct-offset addressing allows you to access specific elements in arrays or data structures by adding an offset to the base address.
- It's particularly useful for accessing individual elements within arrays.
- In the code, it's demonstrated for byte, word, and doubleword arrays. For example, `mov al, [arrayB+1]` accesses the second byte in arrayB.

#### **ExitProcess Function:**

- The program concludes by invoking the ExitProcess function to exit the application. This function is part of the Windows API and is used to terminate the program gracefully.

The code serves as a practical demonstration of these assembly language concepts, illustrating how data transfers, memory operations, and type extensions work in x86 assembly. It's essential to understand these concepts when working with assembly language for tasks like data manipulation and memory management.

## ***Debugging 1***

Compiled the program and started debugging it in x64dbg instead of visual studio community, coz it doesn't have the registers option:



CPU	Log	Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source	References	Threads	Handles	Trace

The CPU pane displays assembly code. The instruction at address 00BC1041 is highlighted in yellow: `mov al,byte ptr ds:[<karrayB>]`. The EIP register also points to this instruction.

```

00BC100D CC int3
00BC100E CC int3
00BC100F CC int3
00BC1010 66:BB 9BA6 mov bx,A69B
00BC1014 0FB7C3 movzx eax,bx
00BC1017 0FB6D3 movzx edx,bl
00BC101A 66:0FB6CB movzx cx,bl
00BC101E 66:BB 9BA6 mov bx,A69B
00BC1022 0FBFC3 movsx eax,bx
00BC1025 0FBED3 movsx edx,bl
00BC1028 B3 7B mov bl,7B
00BC102A 66:0FBECB movsx cx,bl
00BC102E 66:A1 0040BC00 mov ax,word ptr ds:[<val1>]
00BC1034 66:8705 0240BC00 xchg word ptr ds:[<val2>],ax
00BC103B 66:A3 0040BC00 mov word ptr ds:[<val1>],ax
00BC1041 A0 0440BC00 mov al,byte ptr ds:[<karrayB>]
00BC1046 A0 0540BC00 mov al,byte ptr ds:[BC4005]
00BC104B A0 0640BC00 mov al,byte ptr ds:[BC4006]
00BC1050 66:A1 0940BC00 mov ax,word ptr ds:[<karrayW>]
00BC1056 66:A1 0B40BC00 mov ax,word ptr ds:[BC400B]
00BC105C A1 0F40BC00 mov eax,dword ptr ds:[<karrayD>]

```

The Registers pane shows the state of various registers:

- EAX: FFFF2000
- EBX: 00EBA67B
- ECX: 00BC007B assembly2.00BC007B
- EDX: FFFFFFF9B
- EBP: 010FF7C8
- ESP: 010FF7BC
- ESI: 00BC1005 <assembly2.OptionalHeader>
- EDI: 00BC1005 <assembly2.OptionalHeader>
- EIP: 00BC1041 assembly2.00BC1041
- EFLAGS: 00000246
- ZF: 1 PF: 1 AF: 0
- OF: 0 SF: 0 DF: 0
- CF: 0 TF: 0 IF: 1

The Stack pane shows the current stack contents:

```

al=0
byte ptr ds:[byte ptr ds:[00BC4004]]=00BC4004 <assembly2.arrayB>=10

```

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1	Locals	Struct	010FF7BC	752F7BA9	return to kernel32.BaseThreadInitTh
Address	Hex									
77591000	18 00 00 00 00 00 00 00 B8 19 59 77 40 00 00 00							010FF7C0	00EBC000	
77591010	00 00 00 00 00 00 00 00 14 00 16 00 38 AB 59 77							010FF7C4	752F7B90	kernel32.BaseThreadInitThunk
77591020	00 00 02 00 DC 59 59 77 30 5C 5C 77 70 55 5C 77							010FF7C8	010FF820	
77591030	00 00 00 00 B0 A5 69 77 00 E4 5F 77 10 B6 69 77							010FF7CC	775FB79B	return to ntdll.RtlInitializeExcept:
77591040	50 74 5C 77 70 55 5C 77 00 00 00 00 90 B5 69 77							010FF7D0	00EBC000	
77591050	70 60 5C 77 80 01 5D 77 00 00 00 00 00 00 00 00							010FF7D4	4924446F	
77591060	E0 6D 5C 77 F0 6F 5C 77 18 00 00 00 00 00 00 00							010FF7D8	00000000	
77591070	C0 19 59 77 40 00 00 00 00 00 00 00 00 00 00 00							010FF7DC	00000000	
								010FF7F0	00EBC000	

You can see the registers, flag registers, memory dumps, stack etc.

Each flag is assigned a value of 0 (clear) or 1 (set). Here's an example:

```
EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1
```

```
EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000000
LastStatus 00000000
GS 002B FS 0000
ES 002B DS 0000
CS 0023 SS 0000
ST(0) 0000000000000000
```

A screenshot of a debugger interface showing the EFLAGS register. The register value is 00000246, and its bits are listed as follows:

Bit #	Mask	Flag
0	1h	CF (Carry flag)
2	4h	PF (Parity flag)
4	10h	AF (Auxiliary Carry flag)
6	40h	ZF (Zero flag)
7	80h	SF (Sign flag)
8	100h	TF (Trap flag)
9	200h	IF (Interrupt enable flag)
10	400h	DF (Direction flag)
11	800h	OF (Overflow flag)

Yes, it is normal for the disassembled EXE to have more lines of code than what you wrote. This is because the compiler adds additional code to the EXE file to implement library functions, operating system calls, and other features.

For example, in the disassembly you provided, I can see that the compiler has added code to initialize the stack, call the `main()` function, and exit the program. The compiler has also added code to handle various errors, such as division by zero and invalid memory access.

In addition to the code that is added by the compiler, the EXE file may also contain code from libraries that are linked to the program. For example, if your program uses the C standard library, the EXE file will contain code from the standard library.

Finally, the EXE file may also contain code that is injected by other programs, such as malware or debugging tools.

If you are interested in learning more about the code in the disassembled EXE file, you can use a debugger to step through the code and inspect the registers and memory. You can also use a disassembler to generate a more detailed listing of the assembly code.

x64dbg - File: Calculator.exe - PID: 2EFC - Module: ntdll.dll - Thread: 3BC8

File View Debug Trace Plugins Favourites Options Help Apr 5 2018

64 CPU Graph Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source

RIP →

**Disassembly codes**

```

00007FFCA34577C0 40 53 push rbx
00007FFCA34577C2 48 83 EC 30 sub rsp, 30
00007FFCA34577C6 83 64 24 48 00 and dword ptr ss:[rsp+48], 0
00007FFCA34577CB 4C 8D 44 24 48 lea r8, qword ptr ss:[rsp+48]
00007FFCA34577D0 48 83 64 24 20 00 and qword ptr ss:[rsp+20], 0
00007FFCA34577D6 41 B9 04 00 00 00 mov r9d, 4
00007FFCA34577DC 8B D9 mov ebx, ecx
00007FFCA34577DE 48 C7 C1 FE FF FF FF mov rcx, FFFFFFFFFFFFFF
00007FFCA34577E5 41 8D 51 08 lea edx, qword ptr ds:[r9+8]
00007FFCA34577E9 E8 22 BB 05 00 call <ntdll.NtQueryInformation
00007FFCA34577E9 85 C0 test eax, eax
00007FFCA34577F0 77 77 jndl11.7FFCA34577F9
00007FFCA34577F2 83 7C 24 48 00 cmp dword ptr ss:[rsp+48], 0
00007FFCA34577F7 75 15 jne ntdll.7FFCA345780E
00007FFCA34577F9 E9 52 73 FC FF call <ntdll.LdrShutdownThread
00007FFCA34577F9 8E C8 00 00 00 xor ecx, ecx
00007FFCA3457800 E9 1B 00 00 00 call <ntdll.TpCheckTerminateW
00007FFCA3457805 8B D3 mov edx, ebx
00007FFCA3457807 33 C9 xor ecx, ecx
00007FFCA3457809 E8 C2 90 05 00 call <ntdll.ZwTerminateThread
00007FFCA345780E 8B CB mov ecx, ebx
00007FFCA3457810 E8 AB 8C FF FF call <ntdll.RtlExitUserProcess
00007FFCA3457815 CC int3
00007FFCA3457816 CC int3

```

r9d=A34E01C0

.text:00007FFCA34577D6 ntdll.dll:\$477D6 #46BD6

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1

Address	Hex	ASCII
00007FFCA3411000	CC	IIIIIII:
00007FFCA3411010	48 89 5C 24 10 48 89 6C 24 18 56 57 41 54 41 55	H.\\$.H.
00007FFCA3411020	41 56 41 57 44 8B 39 45 8B E8 45 3B C7 8B F2 45	AVAWD.!
00007FFCA3411030	1B D2 45 10 47 3E F7 45 D1 OF 84 B1 8A 0A	.OE#DE.
00007FFCA3411040	00 4C 8B 41 18 49 B1 48 43 39 42 24 38 83 E1 04	.L.A.I.
00007FFCA3411050	8B C1 48 89 4C 24 50 48 F7 D8 45 1B E4 41 83 E4	.AH.L\$F
00007FFCA3411060	20 83 CB FF 48 8B C1 47 8D 1C OC 48 F7 D8 43 8D	.EyH./
00007FFCA3411070	14 14 41 8B C3 4D 8B F0 48 1B C9 2B C2 83 E1 04	..A.AM.

**Registers**

RAX	000000E938E87000
RBX	00000000000000000
RCX	00000000000000000
RDX	00007FFCA34E01C0 <ntdll.DbgU
RBP	00000000000000000
RSP	000000E93B1FFB10
RSI	00000000000000000
RDI	00000000000000000
R8	00007FFCA34E01C0 <ntdll.DbgU
R9	00007FFCA34E01C0 <ntdll.DbgU
R10	00000000000000000
R11	00000000000000000
R12	00000000000000000
R13	00000000000000000
R14	00000000000000000
R15	00000000000000000

Default (x64 fastcall) 5 Unlocked

1: rcx 0000000000000000  
2: rdx 00007FFCA34E01C0 <ntdll.DbgU  
3: r8 00000E938E87000  
4: r9 00007FFCA34E01C0 <ntdll.DbgU  
5: rsp 274 0000000000000000

**stack trace**

000000E93B1FFB10	00000000000000000
000000E93B1FFB18	00000000000000000
000000E93B1FFB20	00000000000000000
000000E93B1FFB28	00000000000000000
000000E93B1FFB30	00000000000000000
000000E93B1FFB38	00000000000000000
000000E93B1FFB40	00000000000000000
000000E93B1FFB48	00007FFCA34E0214
000000E93B1FFB50	00000000000000000
000000E93B1FFB58	00000000000000000
000000E93B1FFB60	00000000000000000

**memory**

return to n

Command: Default

Paused Attach breakpoint reached! Time Wasted Debugging: 0:00:00:40

**Here is a more detailed description of each window:**

**Call Stack:** The Call Stack window shows a list of all the functions that were called to reach the current point in the program. The functions are listed in reverse order, with the most recent function call at the top of the list. To view the Call Stack window, select View > Call Stack.

**Memory Map:** The Memory Map window shows the layout of memory in the program, including the addresses and sizes of all loaded modules, data segments, and stacks. To view the Memory Map window, select View > Memory Map.

**Command Window** is a text-based window that allows you to enter and execute debugger commands. You can use the Command Window to control the debugger session, display and analyze program data, and perform various debugger actions. For example, you can use the Command Window to:

- Start and stop the debugger session.
- Set and remove breakpoints.
- Step through the program line by line.
- Inspect and modify program data.

**Registers Window** shows the current values of the CPU registers. The CPU registers are special memory locations that store information that is frequently used by the CPU, such as the current instruction being executed, the address of the next instruction to be executed, and the results of recent calculations. By inspecting the values of the CPU registers, you can gain insights into what the program is doing and where it is in the execution process.

**Disassembly Window** shows the disassembly of the machine code instructions that are being executed. Machine code is the low-level language that is actually executed by the CPU. By inspecting the disassembly, you can see exactly what instructions the CPU is executing and how they are affecting the state of the program.

**Variables Window** shows the values of the variables in the program. Variables are named memory locations that store data. By inspecting the values of the variables, you can see what data is being used by the program and how it is changing over time.

**Threads Window** shows a list of all the threads in the program. A thread is a single path of execution through a program. The Threads Window allows you to select a thread to inspect and debug. You can also use the Threads Window to start and stop threads, and to suspend and resume threads.

**Breakpoints Window** shows a list of all the breakpoints that have been set in the program. A breakpoint is a marker in the program that tells the debugger to stop execution at that point. Breakpoints are useful for debugging because they allow you to stop the program at specific points and inspect its state.

---

1. What are the three basic types of operands? The three basic types of operands in assembly language are:

Register operands (e.g., EAX, ECX)

Memory operands (e.g., [var1])

Immediate operands (e.g., 42)

2. (True/False): The destination operand of a MOV instruction cannot be a segment register. True. In a MOV instruction, the destination operand cannot be a segment register. Segment registers are used for memory addressing and cannot be directly overwritten using a MOV instruction.

3. (True/False): In a MOV instruction, the second operand is known as the destination operand. False. In a MOV instruction, the first operand is known as the destination operand, and the second operand is the source operand. The data is moved from the source operand to the destination operand.
4. (True/False): The EIP register cannot be the destination operand of a MOV instruction. True. The EIP (Instruction Pointer) register is used to store the address of the next instruction to be executed. It cannot be directly modified or overwritten using a MOV instruction.
5. In the operand notation used by Intel, what does reg/mem32 indicate? The operand notation "reg/mem32" indicates that the operand can be either a 32-bit general-purpose register (reg) or a 32-bit memory operand (mem), which can be a memory location or variable.
6. In the operand notation used by Intel, what does imm16 indicate? The operand notation "imm16" indicates an immediate value of 16 bits. It represents a constant value that is part of the instruction itself and is not stored in a register or memory location.

## ***Addition and Subtraction***

**INC (Increment):** The INC instruction increments the value of a register or memory operand by 1. Here's the syntax:

## INC reg/mem

For example, if you have a data segment with a variable myWord initialized to 1000h, you can use INC like this:

```
.data  
    myWord WORD 1000h  
  
.code  
    inc myWord ; myWord = 1001h
```

---

**DEC (Decrement):** The DEC instruction decrements the value of a register or memory operand by 1. Its syntax is similar to INC:

## DEC reg/mem

For instance, you can use DEC to decrement the value stored in the bx register:

```
.data  
    myWord WORD 1000h  
  
.code  
    inc myWord ; myWord = 1001h  
    mov bx, myWord ; Load myWord into bx  
    dec bx ; bx = 1000h
```

Flags Affected: The INC and DEC instructions affect various CPU status flags, including:

- Overflow
- Sign
- Zero
- Auxiliary Carry
- Parity

The specific changes to these flags depend on the value of the destination operand. Notably, the Carry flag is not affected by these instructions, which might be surprising given its role in other arithmetic operations.

In assembly language, attention to detail is crucial, as each instruction can have subtle effects on the state of the CPU and memory. Understanding how these instructions affect flags is essential for effective programming.

The ADD instruction is used to add a source operand to a destination operand of the same size.  
Here's the syntax:

`ADD dest, source`

or

`add dest, source`

Assembler doesn't care about capital letters!

- The source operand remains unchanged.
- The sum of the operands is stored in the destination operand.

For example, let's add two 32-bit integers:

```
.data  
    var1 DWORD 10000h  
    var2 DWORD 20000h
```

```
.code  
    mov eax, var1    ;EAX = 10000h  
    add eax, var2    ;EAX = 30000h
```

The instruction affects various CPU flags, including Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity. How these flags change depends on the result placed in the destination operand.

---

## **SUB INSTRUCTION**

The SUB instruction subtracts a source operand from a destination operand. The syntax is the same as for ADD:

```
SUB dest, source
```

- Like ADD, the source operand remains unchanged.
- The result of the subtraction is stored in the destination operand.

For example, let's subtract two 32-bit integers:

```
.data
var1 DWORD 30000h
var2 DWORD 10000h

.code
mov eax, var1      ; EAX = 30000h
sub eax, var2      ; EAX = 20000h
```

Again, this instruction affects CPU flags such as Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity based on the value stored in the destination operand.

The SUB instruction subtracts var2 from var1. So, in this case:

- var1 is the source operand, and it contains the value 30000h.
- var2 is the destination operand, and it contains the value 10000h.

The SUB instruction subtracts var2 from var1, resulting in EAX being set to 20000h. Therefore, var2 is subtracted from var1.

---

## **NEG INSTRUCTION**

The NEG (negate) instruction reverses the sign of a number by converting it to its two's complement. It can be applied to registers or memory. Here's the syntax:

To find the two's complement, reverse all the bits in the destination operand and add 1.

For example, to negate the value in EAX:

**NEG reg/mem**

**neg eax ; Negate the value in EAX**

As with ADD and SUB, the NEG instruction also affects CPU flags based on the result.

---

## ***IMPLEMENTING THE ARITHMETIC EXPRESSIONS***

With the ADD, SUB, and NEG instructions, you can implement arithmetic expressions in assembly language. You can break down an expression into individual operations and combine them.

For instance, if you want to calculate **Rval = -Xval + (Yval - Zval)**, you can do it step by step as demonstrated in the code provided in the notes.

```
.data
Rval SDWORD ? ; Define a signed 32-bit variable for Rval
Xval SDWORD 26 ; Initialize Xval to 26
Yval SDWORD 30 ; Initialize Yval to 30
Zval SDWORD 40 ; Initialize Zval to 40

.code
; Step 1: Negate Xval and store it in EAX
mov eax, Xval ; EAX = 26
neg eax ; Negate EAX, now EAX = -26

; Step 2: Subtract Zval from Yval and store it in EBX
mov ebx, Yval ; EBX = 30
sub ebx, Zval ; Subtract Zval from EBX, now EBX = -10

; Step 3: Add the results (EAX and EBX) and store it in Rval
add eax, ebx ; Add EAX and EBX, result in EAX
mov Rval, eax ; Store the result in Rval

; At this point, Rval contains the desired value: Rval = -Xval + (Yval - Zval)
```

In this code:

- We first negate Xval and store the result in EAX.

- Then, we subtract Zval from Yval and store the result in EBX.
- Finally, we add the values in EAX and EBX and store the result in Rval, which gives us  $Rval = -Xval + (Yval - Zval)$ .

Please note that this code assumes that Rval, Xval, Yval, and Zval are signed 32-bit integers, as indicated in the data section.

-----

## ***NEGATING WITHOUT REGISTERS***

You can certainly negate a variable directly in assembly language without using a separate register to store the result temporarily.

The example I provided in my previous response used registers for clarity and readability to illustrate each step of the calculation.

In practice, you may choose to optimize the code for efficiency by negating the variable directly. Here's a more concise version of the code that negates Xval and calculates Rval without using an extra register:

```
.data
Rval SDWORD ? ; Define a signed 32-bit variable for Rval
Xval SDWORD 26 ; Initialize Xval to 26
Yval SDWORD 30 ; Initialize Yval to 30
Zval SDWORD 40 ; Initialize Zval to 40

.code
; Negate Xval directly
neg Xval          ; Negate Xval in place

; Calculate Rval
mov eax, Yval    ; EAX = 30
sub eax, Zval    ; Subtract Zval from EAX, now EAX = -10
add eax, Xval    ; Add Xval to EAX, now EAX = -36
mov Rval, eax    ; Store the result in Rval
```

---

## SIGNED NUMBERS

**Signed numbers** represent both positive and negative values. They use a portion of the binary representation to indicate the sign of the number (positive or negative) and the remaining bits to represent the magnitude (absolute value) of the number. The most common method for representing signed numbers is using Two's Complement.



Here's how it works:

- In Two's Complement, the leftmost bit (the most significant bit) is used as the sign bit. If it's 0, the number is positive; if it's 1, the number is negative.
- The remaining bits represent the absolute value of the number. To negate a signed number, you invert all the bits (change 0s to 1s and vice versa) and then add 1 to the result.

For example:

- 0010 (binary) is +2 in Two's Complement.
- 1010 (binary) is -2 in Two's Complement. (Invert all bits, add 1:  $0101 + 1 = 0110$ , which is -2 in decimal)

**For example: Let's find out the 2's complement of given 8-bit number**

00101001

$$\begin{array}{r} \textcolor{blue}{11010110} \\ + \textcolor{black}{0000000\textcolor{red}{1}} \\ \hline \textcolor{black}{11010111} \end{array}$$

Invert the bits

Then, add 1

The 2's complement of 00101001 is **11010111**

---

## **UNSIGNED NUMBERS**

**Unsigned numbers**, on the other hand, only represent positive values. They use all their bits to represent the magnitude of the number, with no sign bit. This means unsigned numbers can only take on non-negative values.

# Unsigned Integer

number

10000001

For example:

- 0010 (binary) is 2 in unsigned representation.
- There is no negative representation for unsigned numbers, so concepts like negation do not apply.

In summary, signed numbers have a sign bit that allows them to represent both positive and negative values, while unsigned numbers only represent non-negative values. The choice between signed and unsigned representation depends on the specific requirements of a given problem or application.

-----

**Someone might confuse in signed and unsigned eg if they meet a number binary with 1 at the front they assume its signed and its negative, because it has a 1 at the front. How to counter this confusion?**

You're absolutely right that the presence of a leading 1 in a binary number might lead people to assume it's a signed (negative) number, especially if they are not familiar with the representation used. To counter this confusion, it's important to follow some best practices in both programming and documentation:

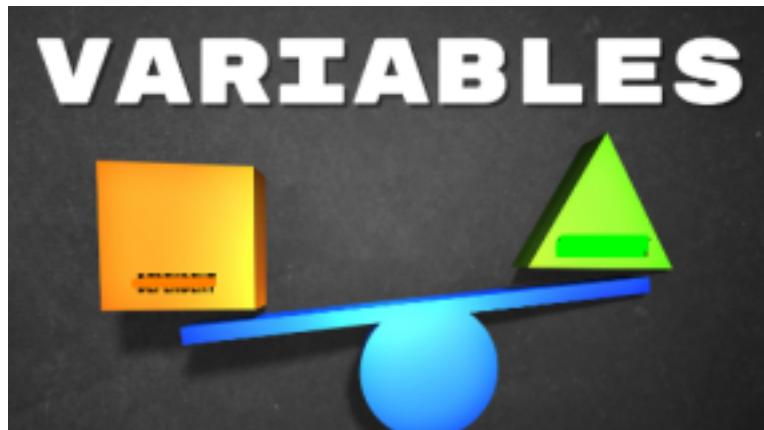
**Clear Documentation:** When working with binary numbers in your code or data structures, always provide clear and explicit documentation to indicate whether the numbers are signed or unsigned. Include comments or annotations that clarify the representation being used.



**Use Standard Conventions:** Adhere to standard conventions for representing signed and unsigned numbers. In many programming languages and assembly languages, signed numbers are represented using Two's Complement notation, while unsigned numbers use straightforward binary representation.

# STANDARD PRACTICE

**Variable Naming:** Choose variable names that reflect the type of data they store. For example, use names like `unsignedValue` or `signedValue` to make it explicit whether a variable is intended to hold an unsigned or signed number.



**Consistent Use of Data Types:** Ensure that your code consistently uses the appropriate data types for signed and unsigned values. For instance, if your programming language supports specific data types like `int` (signed) and `uint` (unsigned), use them consistently.



**Testing and Validation:** During code development, test your calculations and operations with known inputs and expected results to verify that your assumptions about the data representation are correct.



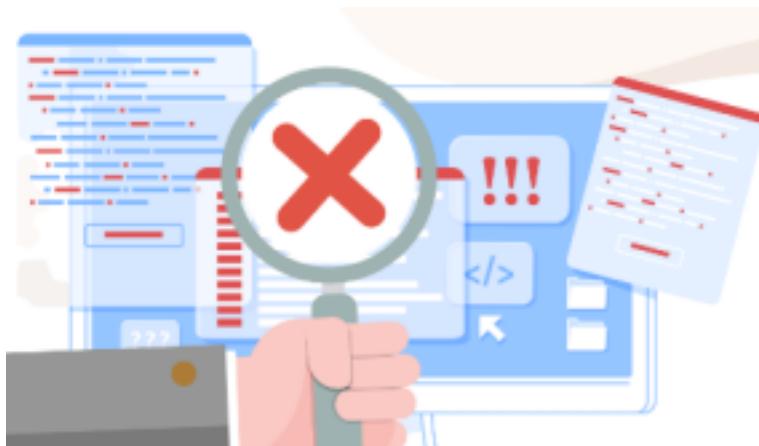
## Data Validation Testing

Tools and Best Practices

**Educate Team Members:** If you're working on a team, take the time to educate your colleagues about data representation conventions and the significance of the leading bit. Encourage open communication about data representations to avoid misunderstandings.



**Avoid Mixing Representations:** In situations where both signed and unsigned data might be present, avoid mixing them within the same context or calculation. Keep signed and unsigned operations separate to minimize confusion.



**Error Handling:** Implement appropriate error handling and checks in your code to catch any unexpected or invalid data representations. This can help identify and address issues early.



By following these practices and promoting awareness of data representation conventions among your team members, you can mitigate the potential confusion associated with the leading bit of binary numbers and ensure that your code is clear and unambiguous.

## *Flags Registers*

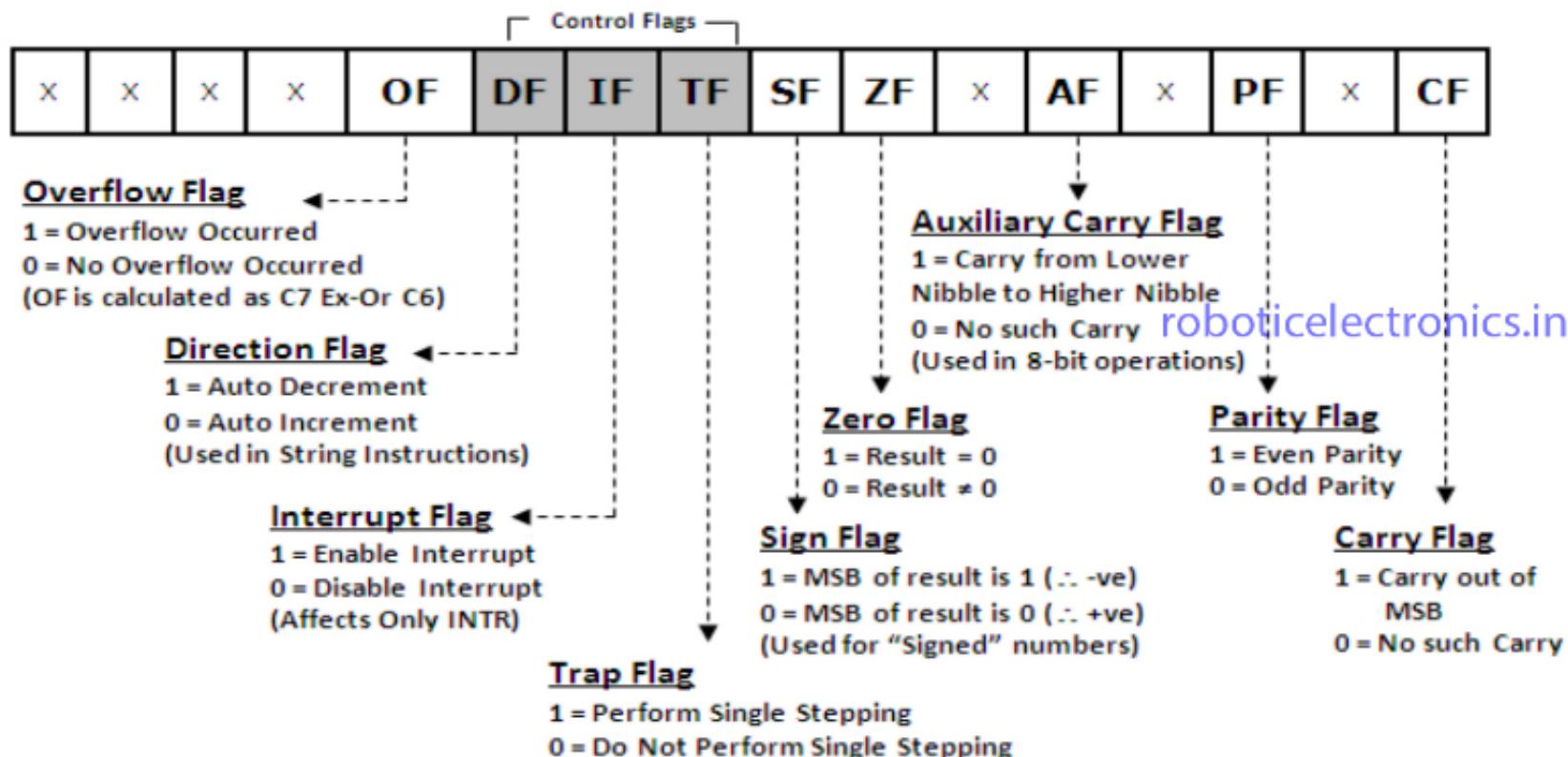
In computer programming, when we perform arithmetic operations, we often need to determine certain properties of the result.

Is it positive, negative, or zero? Is it within the valid range for the destination variable?

To answer these questions, we rely on CPU status flags, which provide valuable information about the outcome of arithmetic operations.

These flags are crucial for error detection and controlling program flow.

Flag register of 8086 is a 16-bit register where status of the latest Arithmetic operation performed.



flag register of 8086

As it has 16-bits , it has 16 flags. These 16 flags are classified as

- 7 are don't care flags.
- 3 are control flags ( accessible to programmers ).
- 6 are status flags ( not accessible to programmers ).

Here's a straightforward overview of the main status flags.

First, remember this, MSB is the leftmost bit.

## Most significant bit examples

Binary number "1000"



Binary number "0111"



### *Carry Flag:*

Set when the result of an unsigned arithmetic operation is too large to fit into the destination

operand.

$$\begin{array}{r} 1111 \text{ (Carry-In)} \\ + 1101 \\ \hline 11000 \text{ (Carry-Out)} \end{array}$$

In this example, when you add 1111 and 1101, you have a carry-out from the MSB (leftmost bit), resulting in a final carry-out value of 11000. The Carry flag would be set to indicate this carry-out condition.

For subtraction, there's typically no carry-out from the MSB, as subtraction involves taking away one value from another:

$$\begin{array}{r} 1101 \\ - 1010 \\ \hline 011 \end{array}$$

In this subtraction, there's no **carry-out from the LSB**, so the Carry flag would not be set. The carry-out occurs from the least significant bit (LSB), and propagates towards the most significant bit (MSB) during addition, not from the MSB.

This flag indicates unsigned integer overflow. For instance, if an operation has an 8-bit destination operand, but the result exceeds 255 in binary (11111111), the Carry flag is set.

**Carry is generated**

$$\begin{array}{r} 0000\ 0100 \\ 0000\ 0101 \\ \hline 0000\ 1001 \end{array}$$

Another example, subtracting 2 from 1 sets the carry flag.

FIGURE 4–4 Subtracting 2 from 1 sets the Carry flag.

$$\begin{array}{cccccccc}
 | & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 + & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
 \hline
 \text{CF} & 1 & 1 & 1 & 1 & 1 & 1 & 1
 \end{array}
 \quad (1) \quad (-2) \quad (\text{FFh})$$

$$\begin{array}{r} \text{CF} = 1 \\ \begin{array}{r} \overset{1}{\overset{1}{1}} 1100 \\ + 10010001 \\ \hline 101001101 \end{array} \end{array}$$

$$\begin{array}{r} \text{CF} = 0 \\ \begin{array}{r} \overset{1}{\overset{1}{1}} 1100 \\ + 00010001 \\ \hline 11001101 \end{array} \end{array}$$

**If CF = 1; it means carry is generated from the MSB (Most Significant Bit):**

- When CF (Carry Flag) is set to 1, it indicates that during an operation like addition, a carry-out occurred from the MSB (the leftmost or highest-order bit) of the result. This typically happens when the sum of two numbers overflows the capacity of the binary representation. In other words, it suggests that the result is too large to fit within the allocated number of bits.

**If CF = 0; no carry is generated out of MSB:**

- When CF is set to 0, it means that no carry-out occurred from the MSB during the operation. This signifies that the result of the operation did not exceed the capacity of the allocated bits, and there was no overflow.

In summary, the notes are explaining how the Carry Flag (CF) is used to detect whether there was a carry-out from MSB, during arithmetic operations, with CF = 1 indicating overflow and CF = 0 indicating no overflow.

## Auxiliary Carry Flag

**Auxiliary Carry Flag:** This flag is set when a 1 bit carries out of position 3 in the least significant byte of the destination operand.

Suppose we want to add 1 to the BCD value 0Fh:

```
mov al, 0Fh ; Load the value 0Fh into the AL register  
add al, 1    ; Add 1 to AL
```

Here's the binary representation and the arithmetic:

1. Binary Representation of 0Fh: 00001111
2. Binary Representation of 1: 00000001

Now, let's add them together, keeping track of the AC flag:

0000	1111	(0Fh)
+ 0000	0001	(1)
-----		
0001	0000	(10h)

In this addition, we are adding 1 to 0Fh. As you correctly pointed out, the sum (10h) contains a 1 in bit position 4 that was carried out of bit position 3. This carry from bit 3 to bit 4 is precisely what the Auxiliary Carry (AC) flag represents.

So, in this case:

- **AC = 1:** The AC flag is set because a carry occurred from bit 3 to bit 4 during the addition of 1 to 0Fh. This is a typical scenario in BCD arithmetic, where values are represented in a decimal-like fashion using binary.
- 

The AC flag in assembly language is set when a 1 bit carries out of the 3rd bit (counting from the right, or least significant bit) of the destination operand during an operation.

Imagine you have an 8-bit binary number, and you're performing an operation like addition or subtraction. If there's a carry (1) out of the 3rd bit position (bit 2), the AC flag is set to 1. If there's no carry out of that bit position, the AC flag is set to 0.

## AF:

AF stands for auxiliary flag. As 8-bits form a byte, similarly 4 bits form a nibble. So in 16 bit operations there are 4 nibbles.

If AF = 1 ; there is a carry out from lower nibble.

If AF = 0 ; no carry out of lower nibble.

$$AF = 1$$

$$\begin{array}{r} \textcolor{blue}{1} \text{ } \textcolor{blue}{1} \textcolor{red}{(1)} \\ 10111100 \\ + 10001001 \\ \hline 1 \quad 01000101 \end{array}$$

$$AF = 0$$

$$\begin{array}{r} \textcolor{blue}{1} \text{ } \textcolor{blue}{1} \\ 10111100 \\ + 00010001 \\ \hline 11001101 \end{array}$$

=====

**Parity Flag:** The Parity flag informs us whether an even number of 1 bits appears in the least significant byte of the destination operand right after an arithmetic or boolean instruction.

The PF is set when the least significant byte of the destination has an even number of 1 bits. It checks if the result of an operation has an even or odd number of 1 bits in its binary representation.

### Example 1: ADD Instruction

```
mov al, 10001100b      ; AL = 10001100
add al, 00000010b      ; Add 00000010 to AL
```

$$\begin{array}{r} 10001100 \\ + 00000010 \\ \hline \end{array}$$

-----  
10001110

After the ADD instruction, AL contains the binary value 10001110. In this binary representation, there are four 0 bits and four 1 bits.

Since there is an even number of 1 bits (four), the Parity flag (PF) is set to 1. This means that PF indicates that the result has an even parity.

## Example 2: SUB Instruction

```
sub al, 10000000b      ; Subtract 10000000 from AL
```

```
10001110  
- 10000000  
-----  
00001110
```

After the SUB instruction, AL contains the binary value 00001110. In this binary representation, there are three 0 bits and one 1 bit. Since there is an odd number of 1 bits (one), the Parity flag (PF) is set to 0. This indicates that PF detects an odd parity in the result.

In summary:

- After the ADD instruction, AL contains an even number of 1 bits (four), so PF is set to 1 (even parity).
- After the SUB instruction, AL contains an odd number of 1 bits (one), so PF is set to 0 (odd parity).

The Parity flag (PF) is used for checking whether the result of an operation has even or odd parity, which can be useful in certain applications where parity checks are required for error detection.

**PF:**

It stands for parity flag.

If PF = 1 ; it means it is even parity in the result ( there are even numbers of 1's ).

If PF = 0 ; it means it is odd parity.

$$\text{PF} = 1$$

$$\begin{array}{r} \overset{1}{\overset{1}{\text{1}}} 0111100 \\ + 10010001 \\ \hline 1 01001101 \end{array}$$

$$\text{PF} = 0$$

$$\begin{array}{r} \overset{1}{\overset{1}{\text{1}}} 0111100 \\ + 00010001 \\ \hline 11001101 \end{array}$$

## **Overflow Flag:**

---

**Overflow Flag:** Set when the result of a signed arithmetic operation is too large or too small to fit into the destination operand.

### **Unsigned:**

This is a 8 bit positive number which ranges from 0 to 255. In hexadecimal its range is from 00 to FF. In the OF flag, it has nothing to do with unsigned numbers. Only signed numbers are considered in the OF flag.

**Signed:** This is also a 8-bit number( can be 16-bit too ) which is equally distributed among +ve and -ve numbers. By considering MSB , it is decided whether it is a positive or negative number. If MSB=1 , it is a negative number or else its a positive number.

1011 0011 ;is -ve

0111 1001 ;is +ve

For example, 1011 0011 is negative, and its two's complement is 0100 1101, which is equivalent to 4D

in hexadecimal and represents -77 in decimal.

#### Range of Signed Numbers:

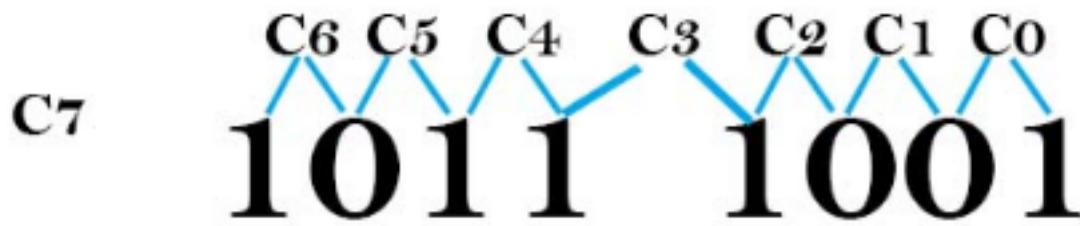
- In an 8-bit signed number, the range is from -128 to 127 (hexadecimal -80 to 7F).
- It consists of 128 positive values (0 to 127) and 128 negative values (-1 to -128).

#### Overflow Detection:

- Overflow occurs when the result of an operation exceeds the valid range of signed numbers.
- If there's an overflow, the MSB would show incorrect values: MSB = 1 for a positive overflow, MSB = 0 for a negative overflow.

#### Overflow Detection Mechanism:

- To detect overflow, the processor performs an XOR operation on the carry flags (C6 and C7) during arithmetic operations.
- C0 is the carry from the 0th bit to the 1st bit, and so on. If the XOR operation results in 1, it indicates that the operation has gone beyond the limits of signed numbers and overflowed.



0110 1100

The notes you provided are good to read and accurate. The image you sent shows the following:

C6 C5 C4 C3 C2 C1 Co  
C7

This represents the carry flags from bit 0 to bit 7 of the result of an arithmetic operation.

The **Overflow flag (OF)** is set when the result of a signed arithmetic operation is too large or too small to fit into the destination operand. This can happen when the result is greater than 127 or less than -128.

To detect overflow, the processor performs an XOR operation on the carry flags C6 and C7. If the result of the XOR operation is 1, then overflow has occurred.

For example, let's say we are adding two signed 8-bit numbers: 127 and 1. The result is 128, which is greater than the maximum value for a signed 8-bit number. Therefore, overflow has occurred.

The following table shows how the carry flags and Overflow flag are affected by the addition operation:

Carry flag	Overflow flag
C7 = 1, C6 = 0	OF = 1

This is because the XOR operation of C7 and C6 is 1, which indicates that overflow has occurred.

The Overflow flag can be used to check for errors in arithmetic operations. For example, if we are expecting the result of an addition operation to be within a certain range, we can check the Overflow flag to see if it is set. If it is set, then we know that an error has occurred and we can take corrective action.

Here is an example of how to use the Overflow flag in assembly language:

```
; Add two 8-bit numbers: 127 and 1
add al, 1

; Check the Overflow flag
jnc no_overflow

; Overflow has occurred, handle the error
overflow:
...

; No overflow, continue with the program
no_overflow:
...
```

The assembly code you provided is a simple example of how to use the Overflow flag to detect overflow in an addition operation. The code works as follows:

1. The `add al, 1` instruction adds the value 1 to the al register.

2. The `jnc no_overflow` instruction jumps to the `no_overflow` label if the Overflow flag is not set.
3. If the Overflow flag is set, then the program jumps to the `overflow` label, where the error can be handled.

The Overflow flag is set when the result of an addition operation is too large to fit into the destination operand. In this case, the destination operand is the al register, which is an 8-bit register.

The maximum value that can be stored in an 8-bit register is 127. Therefore, if the result of the addition operation is greater than 127, then overflow will occur.

In summary, the Overflow Flag (OF) is used to detect overflow in signed number arithmetic. It's crucial for ensuring the correctness of arithmetic operations, especially when dealing with signed data types. The XOR operation on carry flags is one method used by the processor to detect this overflow condition.

=====

## **Zero Flag:**

=====

**Zero Flag:** Set when the result of an arithmetic operation is zero. For example, subtracting two equal values will set the Zero flag.

## ZF:

This is zero flag. Whenever the output is 0 this flag is 1.

If ZF = 1 ; output is zero.

If ZF = 0 ; output is non zero.

---

## *Sign Flag:*

---

**Sign Flag:** Set when the result of an arithmetic operation is negative. If the most significant bit (MSB) of the destination operand is set, indicating a negative number, the Sign flag is set.

The **Sign flag (SF)** is a status flag that is set to the value of the most significant bit (MSB) of the result of an arithmetic operation.

- If the MSB is 1, then the SF flag is set to 1, which indicates that the result is negative.
- If the MSB is 0, then the SF flag is set to 0, which indicates that the result is positive.

The person is also saying that the status flags, including the SF flag, are controlled by the **ALU (arithmetic logic unit)** and not by the user. This means that the user cannot directly set or clear the status flags.

The person is correct in saying that the status flags can be used to check for errors in arithmetic operations. For example, if the **OF (overflow flag)** is set after an addition operation, then this indicates that the result of the operation is too large to fit into the destination operand. This can be used to detect overflow errors.

Another example is the **ZF (zero flag)**. If the ZF is set after an arithmetic operation, then this indicates that the result of the operation is zero. This can be used to check for zero division errors.

The **status flags** are a very important part of the CPU, and they are used by many different instructions. By understanding how the status flags work, you can better understand how the CPU performs arithmetic operations and how to check for errors.

The sign flag can be used to check for errors in arithmetic operations. For example, if the sign flag is set after an addition operation, then this indicates that the result of the operation is negative.

However, if the operands of the addition operation were both positive, then the result should also be positive. **This means that an error has occurred.**

The sign flag can also be used to determine the sign of a number. For example, if the sign flag is set, then the number is negative. If the sign flag is not set, then the number is positive.

-----

The code you provided shows how the sign flag is affected by a subtraction operation. The first instruction, `mov eax, 4`, moves the value 4 into the eax register.

The second instruction, `sub eax, 5`, subtracts 5 from the value in the eax register. Since 5 is greater than 4, the result of the subtraction operation is negative. This is reflected in the value of the sign flag, which is set to 1.

The next part of the code shows the hexadecimal values of the bl register when a negative result is generated. The first instruction, `mov bl, 1`, moves the value 1 into the bl register.

The second instruction, `sub bl, 2`, subtracts 2 from the value in the bl register. Since 2 is greater than 1, the result of the subtraction operation is negative.

This is reflected in the hexadecimal value of the bl register, which is FFh, which is the hexadecimal representation of -1.

```
mov eax, 4  
sub eax, 5 ; EAX = -1, SF = 1
```

```
mov bl, 1  
sub bl, 2 ; BL = FFh (-1), SF = 1
```

The sign flag is a useful tool for checking for errors in arithmetic operations and determining the sign of a number.

=====

## *Control Flags:*

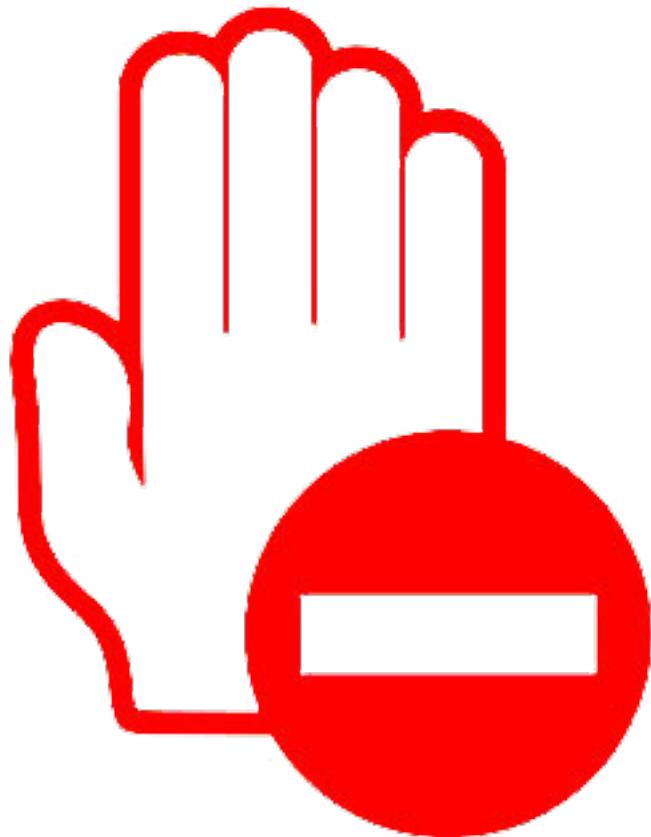
---

The control flags are three special flags that control the operation of the CPU. They are:

**Trap flag (TF):** When the TF flag is set to 1, the CPU enters single-step mode. This means that the CPU will execute one instruction and then stop. This is useful for debugging programs, as it allows you to step through a program one instruction at a time.



**Interrupt flag (IF):** When the IF flag is set to 1, the CPU will allow interrupts. Interrupts are signals from external devices that need the CPU's attention. When an interrupt occurs, the CPU will temporarily stop what it is doing and handle the interrupt. After handling the interrupt, the CPU will return to the program it was executing.



ComputerHope.com

**Direction flag (DF):** The DF flag controls the direction of string operations. When the DF flag is set to 0, the CPU will increment the string pointer after each operation. When the DF flag is set to 1, the CPU will decrement the string pointer after each operation. The image you sent shows the TF,

IF, and DF flags in the flag register of the 8086 microprocessor. The flags are set to 0 by default, which means that the CPU is not in single-step mode, interrupts are disabled, and string operations will increment the string pointer.



Here are some examples of how the control flags can be used:

- **Debugging:** You can use the TF flag to step through a program one instruction at a time. This can be useful for finding bugs in your program. For example, if you are having trouble understanding how the DF flag works, you can set the TF flag and step through the string operations one instruction at a time.
- **Disabling interrupts:** You can use the IF flag to disable interrupts while you are debugging your program. This will prevent interrupts from interfering with your debugging process.

- **Controlling string operations:** You can use the DF flag to control the direction of string operations. For example, if you want to reverse a string, you can set the DF flag to 1.
- 

The **TF flag**, when set to 1, puts the microprocessor in single-step mode, which means that the microprocessor will execute one instruction and then stop. This is useful for debugging programs.

The **IF flag**, when set to 1, enables interrupts. Interrupts are signals from external devices that need the microprocessor's attention. When an interrupt occurs, the microprocessor will temporarily stop what it is doing and handle the interrupt. After handling the interrupt, the microprocessor will return to the program it was executing.

The **DF flag** controls the direction of string operations. When the DF flag is set to 0, the microprocessor will increment the string pointer after each operation. When the DF flag is set to 1, the microprocessor will decrement the string pointer after each operation.

```
;AddSubTest.asm - Arithmetic and Flags Demonstration
.386
.model flat,stdcall
.stack 4096

ExitProcess proto,dwExitCode:dword

.data
    Rval SDWORD ?
    Xval SDWORD 26
    Yval SDWORD 30
    Zval SDWORD 40

.code

main PROC
    ; INC and DEC
    mov ax, 1000h
    inc ax          ; Increment: AX = 1001h
    dec ax          ; Decrement: AX = 1000h

    ; Expression: Rval = -Xval + (Yval - Zval)
    mov eax, Xval      ; Load Xval into EAX
    neg eax          ; Negate EAX (convert to negative)
    mov ebx, Yval      ; Load Yval into EBX
    sub ebx, Zval      ; Subtract Zval from Yval
    add eax, ebx      ; Add the negated Xval and (Yval - Zval) to get Rval
    mov Rval, eax      ; Store the result in Rval

    ; Exit process
    call ExitProcess
    dwExitCode: dw 0
    exit
```

```
; Zero Flag example
mov cx, 1
sub cx, 1           ; ZF = 1 because the result is zero

; Sign Flag example
mov cx, 0
sub cx, 1           ; SF = 1 because the result is negative
mov ax, 7FFFh
add ax, 2           ; SF = 1 because the result is negative

; Carry Flag example
mov al, 0FFh
add al, 1           ; CF = 1, AL = 00 (overflow)

; Overflow Flag example
mov al, 127          ; Maximum positive value
add al, 1           ; OF = 1 (overflow)
mov al, -128         ; Minimum negative value
sub al, 1           ; OF = 1 (overflow)

    INVOKE ExitProcess, 0
main ENDP

END main
```

Repetition of the whole code:

```
; AddSubTest.asm - Arithmetic and Flags Demonstration

.386
.model flat,stdcall
.stack 4096

ExitProcess proto,dwExitCode:dword

.data
    Rval SDWORD ?
    Xval SDWORD 26
    Yval SDWORD 30
    Zval SDWORD 40

.code

main PROC
    ; INC and DEC
    mov ax, 1000h
    inc ax          ; Increment: AX = 1001h
    dec ax          ; Decrement: AX = 1000h

    ; Expression: Rval = -Xval + (Yval - Zval)
    mov eax, Xval   ; Load Xval into EAX
    neg eax         ; Negate EAX (convert to negative)
    mov ebx, Yval   ; Load Yval into EBX
    sub ebx, Zval   ; Subtract Zval from Yval
    add eax, ebx    ; Add the negated Xval and (Yval - Zval) to get Rval
    mov Rval, eax   ; Store the result in Rval

    ExitProcess dwExitCode
```

```
; Zero Flag example
mov cx, 1
sub cx, 1           ; ZF = 1 because the result is zero

; Sign Flag example
mov cx, 0
sub cx, 1           ; SF = 1 because the result is negative
mov ax, 7FFFh
add ax, 2           ; SF = 1 because the result is negative

; Carry Flag example
mov al, 0FFh
add al, 1           ; CF = 1, AL = 00 (overflow)

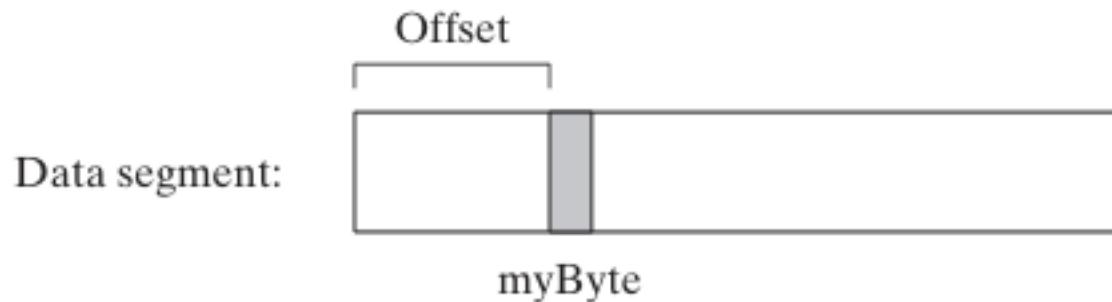
; Overflow Flag example
mov al, 127          ; Maximum positive value
add al, 1           ; OF = 1 (overflow)
mov al, -128         ; Minimum negative value
sub al, 1           ; OF = 1 (overflow)

    INVOKE ExitProcess, 0
main ENDP

END main
```

## *Offset*

A variable named myByte.



The image you sent shows a variable called myByte, an **offset**, and the data segment. The data segment is a section of memory that stores program data, such as variables, constants, and strings. The offset is the distance from the beginning of the data segment to the variable myByte.



For example, if the data segment starts at memory address 1000h and the offset of myByte is 200h, then myByte will be stored at memory address 1200h.

The offset is used by the processor to access the variable myByte. When the processor needs to access myByte, it adds the offset to the starting address of the data segment to get the memory address of myByte.

The data segment is also used by the operating system to manage program data. The operating system keeps track of the starting address and size of the data segment for each program. This allows the

operating system to protect the data of one program from being accessed by another program.

Here is an example of how to access the variable myByte in assembly language:

```
mov ax, data_segment ; Load the starting address of the data segment into the AX register  
add ax, offset_myByte ; Add the offset of myByte to the AX register  
mov al, [ax] ; Load the value of myByte into the AL register
```

The mov instruction is used to move data from one register to another or between a register and memory. The ax register is a general-purpose register that can be used to store any type of data, including integers, floating-point numbers, and addresses. **The al register is a lower 8-bit register of the ax register.**

The first line of code loads the starting address of the data segment into the ax register. The second line of code adds the offset of myByte to the ax register. This gives us the memory address of myByte. The third line of code loads the value of myByte from memory into the al register.

-----

**Is the statement that offset said in the image correct?**

Yes, he is correct. The offset is added to the data segment's starting address to get the memory address of a variable in the data segment. This is how the processor addresses variables in the data segment.

For example, if the data segment's starting address is 1000h and the offset of a variable is 200h, then the memory address of the variable will be 1200h.

The **offset** is a 16-bit value, which means that it can address up to 64KB of memory. This is the maximum size of a data segment in the 8086 microprocessor.

The offset is used in many different instructions, such as the **mov instruction** and the **lea instruction**. The **mov instruction** is used to move data from one register to another or between a register and memory. The **lea instruction** is used to load the effective address of an operand into a register.

The offset is an important concept in assembly language programming, and it is essential to understand how it is used to access variables in the data segment.

---

```
mov ax, data_segment ; Load the starting address of the data segment into the AX register  
add ax, offset_myByte ; Add the offset of myByte to the AX register  
mov al, [ax] ; Load the value of myByte into the AL register
```

The code above is dereferencing the ax register to get the value of the variable myByte, which is located at the address: **data\_segment's address + offset**.

The **mov** instruction in the code loads the starting address of the data segment into the **ax** register. The **add** instruction adds the offset of **myByte** to the **ax** register. This gives us the memory address of **myByte**. The **mov al, [ax]** instruction loads the value of the variable **myByte** from memory into the **al** register.

The **[ax]** notation is used to dereference the **ax** register. This means that the processor will load

the value from the memory address stored in the ax register.

The following is a step-by-step explanation of how the code works:

1. The `mov ax, data_segment` instruction loads the starting address of the data segment into the ax register.
2. The `add ax, offset_myByte` instruction adds the offset of myByte to the ax register. This gives us the memory address of myByte.
3. The `mov al, [ax]` instruction loads the value of the variable myByte from memory into the al register.

After the code has executed, the al register will contain the value of the variable myByte.

=====

## ***How offset works in ASM:***

=====

```
.data  
    bigArray DWORD 500 DUP(?)  
    pArray    DWORD bigArray  
  
    mov esi, pArray  
    ;At this point, ESI contains the address of the beginning of the bigArray
```

The line **pArray DWORD bigArray** in assembly language is defining a doubleword (DWORD) variable named **pArray** and initializing it with the address of the **bigArray** array. Essentially, it creates a pointer variable (**pArray**) that points to the beginning of the **bigArray** array.

**bigArray** contains 500 doubleword(DWORD) elements.

The **OFFSET** operator can also be used to access variables in memory. To access a variable in memory using the **OFFSET** operator, you simply add the offset of the variable to the starting address of the data segment.

For example, the following code accesses the variable **bVal** in memory:

```
.data  
    bVal BYTE ?  
    mov esi, OFFSET bVal  
    ; At this point, ESI contains the address of the variable bVal.  
    mov al, [esi]  
    ; At this point, AL contains the value of the variable bVal.
```

Dereferencing is done ie. accessing data from that address stored in esi (If you don't get this, you should go through the C programming code from my github).

---

```
.data
    bVal BYTE ?
    wVal WORD ?
    dVal DWORD ?
    dVal2 DWORD ?

.code
    mov esi, OFFSET bVal ; ESI = 00404000h
    mov esi, OFFSET wVal ; ESI = 00404001h
    mov esi, OFFSET dVal ; ESI = 00404003h
    mov esi, OFFSET dVal2 ; ESI = 00404007h
```

After the code has executed, the ESI register will contain the offset of the variable that was specified in the MOV instruction.

For example, after the first MOV instruction has executed, the ESI register will contain the offset of the variable bVal. This offset is 00404000h, because that is where the variable bVal is located in memory.

After the second MOV instruction has executed, the ESI register will contain the offset of the variable wVal. This offset is 00404001h, because that is where the variable wVal is located in memory.

And so on.

The OFFSET operator is a useful tool for accessing variables in memory. It is also useful for initializing pointers.

```
.data  
    myArray WORD 1,2,3,4,5  
  
.code  
    mov esi, OFFSET myArray + 4
```

The first line of code in the data segment declares an array called myArray that contains five 16-bit words. The second line of code in the code segment moves the offset of myArray plus 4 to the ESI register.

The OFFSET operator returns the offset of a variable or label. The offset is the distance from the beginning of the data segment to the variable or label. In this case, the offset of myArray is 0000h, because myArray is the first variable in the data segment.

Adding 4 to the offset of myArray gives us the offset of the third integer in the array. This is because the first integer in the array is at offset 0000h, the second integer is at offset 0002h, and the third integer is at offset 0004h.

The ESI register is a general-purpose register that can be used to store any type of data, including addresses. By moving the offset of the third integer in myArray to the ESI register, we are essentially creating a pointer to the third integer in the array.

The ESI register can now be used to access the third integer in the array. For example, the following code would load the value of the third integer in myArray into the AL register:

```
mov al, [esi]
```

## *Calculating offsets for arrays*

Array Type	Array Element Size	Offset
Byte array	1 byte	+1
Word array	2 bytes	+2
Doubleword array	4 bytes	+4
Quadword array	8 bytes	+8

### BYTE Arrays:

If you are working with a BYTE array, where each element is 1 byte in size, you can access the next element by adding +1 to the current element's address.

```
; Assuming byteArray starts at address 00400000h
mov esi, OFFSET byteArray ; Load the starting address of byteArray
add esi, 1 ; Move to the second element (1 byte offset)
mov al, [esi] ; Dereference/Load the value of the second element into AL
```

### WORD Arrays:

For WORD arrays, where each element is 2 bytes (16 bits) in size, you should add +2 to move to the next element's address.

```
; Assuming wordArray starts at address 00400000h
mov esi, OFFSET wordArray ; Load the starting address of wordArray
add esi, 2 ; Move to the third element (2 bytes offset)
mov ax, [esi] ; Load the value of the third element into AX
```

### DWORD Arrays:

DWORD arrays have elements that are 4 bytes (32 bits) in size. To access the next element, add +4 to the current element's address.

```
; Assuming dwordArray starts at address 00400000h
mov esi, OFFSET dwordArray ; Load the starting address of dwordArray
add esi, 4 ; Move to the fourth element (4 bytes offset)
mov eax, [esi] ; Load the value of the fourth element into EAX
```

### QWORD Arrays:

If you are dealing with QWORD arrays, where each element is 8 bytes (64 bits) in size, you should add +8 to move to the address of the next element.

```
; Assuming qwordArray starts at address 00400000h
mov esi, OFFSET qwordArray ; Load the starting address of qwordArray
add esi, 8 ; Move to the fifth element (8 bytes offset)
mov rax, [esi] ; Load the value of the fifth element into RAX
```

## *Align Directive and PTR Byte*

Used to align a variable or label on a specific memory boundary, such as a byte, word, doubleword, or paragraph boundary.

The syntax for the ALIGN directive is as follows: ALIGN bound. For example, **ALIGN 4** aligns the variable or label on a 4-byte boundary.

## Using PTR with Data Types:

When moving data between memory and registers in assembly language, the PTR operator is used in combination with standard assembler data types.

These data types include BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TBYTE. The PTR operator is used to specify the size of the data being moved and helps ensure proper alignment.

## Moving Smaller Values into Larger Destinations:

In some cases, you may need to move smaller values from memory into a larger destination operand, such as moving a word into a doubleword. This can be achieved using the DWORD PTR operator. For example:

```
.data  
wordList WORD 5678h, 1234h  
  
.code  
mov eax, DWORD PTR wordList ; EAX = 12345678h
```

In the example above, the DWORD PTR operator is used to move two words from the wordList array into the lower and upper halves of the EAX register, effectively combining them into a doubleword.

The ALIGN directive and the PTR operator are important tools in assembly language programming to control memory alignment and ensure that data is properly interpreted based on its size when performing data transfers and operations.

---

## ***ALIGN directive***

---

The ALIGN directive is used to align a variable on a byte, word, doubleword, or paragraph boundary. The syntax is:

**ALIGN bound** where **bound** is 1, 2, 4, or 16.

The ALIGN directive is used to improve the performance of certain instructions. For example, the **FPU (floating-point unit)** can process data more quickly if it is aligned on a **doubleword boundary**.

The **ALIGN directive** and the **PTR operator** serve different purposes in assembly language programming, and they are not the same thing. Let's clarify the differences between them and provide examples of each:

The ALIGN directive is used to align a variable or label on a specific memory boundary. It ensures that the **memory address of the variable is a multiple of the specified alignment boundary**.

This can be useful for **optimizing memory access** and ensuring proper data alignment for certain data types, especially when working with **structured data** or **memory-mapped devices**.

Example of ALIGN directive:

```
.data  
myData BYTE 10      ; Declare a byte-sized variable  
ALIGN 4             ; Align the next variable on a 4-byte boundary  
myDouble DWORD 20   ; Declare a doubleword-sized variable
```

In this example, the ALIGN directive is used to align myDouble on a 4-byte boundary. This ensures that myDouble starts at an address that is a multiple of 4.

=====

## ***BYTE PTR Operator***

=====

The **BYTE PTR operator** is used to access a single byte of a variable. The **PTR operator** is used in combination with standard assembler data types to specify the size of the data being accessed or manipulated. It is used when moving data between memory and registers to ensure that the correct number of bytes are read or written.

The following code shows how to use the BYTE PTR operator to access a single byte of the variable myDouble:

myDouble is a doubleword(4 bytes), so the first byte is 78h.

```
.data  
myDouble DWORD 12345678h  
  
.code  
mov bl, BYTE PTR myDouble ; BL = 78h
```

This code will move the value of the least significant byte of the variable myDouble into the register BL.

Memory Layout of the myDouble Variable. The following image shows the memory layout of the myDouble variable:

Doubleword	Word	Byte	Offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

The least significant byte of the variable myDouble is at offset 0000h. The most significant byte of the variable myDouble is at offset 0003h.

## ***Moving Smaller Values into Larger Destinations***

We can use the BYTE PTR operator to move smaller values into larger destination operands. For example, the following code moves two words from memory to the register EAX:

```
.data  
wordList WORD 5678h, 1234h
```

```
.code  
mov eax, DWORD PTR wordList
```

This code will move the two words from the wordList array into the lower half and upper half of the register EAX, respectively.

```
.data  
myValue DWORD 12345678h ; Declare a doubleword-sized variable  
.code  
mov eax, DWORD PTR myValue ; Load the entire DWORD into EAX  
mov al, BYTE PTR myValue ; Load only the lowest byte into AL
```

**NB:** eax is 32 bit register.

In the first mov instruction, the PTR operator is used to specify that the entire DWORD stored in myValue should be loaded into the EAX register.

In the second `mov` instruction, the `PTR` operator is used to specify that only the lowest byte (LSB) of `myValue` should be loaded into the `AL` register.

Differences:

1. **Purpose:** `ALIGN` is used for memory alignment, ensuring that variables are properly positioned in memory for efficient access. `PTR` is used to specify the size of data when performing data transfers or operations.
2. **Usage:** `ALIGN` is a directive that appears in the data section to align variables. `PTR` is an operator used in instructions to specify the size of data during data manipulation.

In summary, `ALIGN` is used to control memory alignment, while `PTR` is used to specify data size in instructions. They serve different purposes and are used in different contexts in assembly language programming.

## *Type, LengthOF, SizeOF, Label*

The `TYPE` operator returns the size, in bytes, of a single element of a variable.

For example, the `TYPE` of a byte is 1, the `TYPE` of a word is 2, the `TYPE` of a doubleword is 4, and the `TYPE` of a quadword is 8. Here are examples of each:

```
.data  
    var1 BYTE ?  
    var2 WORD ?  
    var3 DWORD ?  
    var4 QWORD ?
```

;The following table shows the value of each TYPE expression.

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

---

## **LENGTHOF Operator:**

---

The LENGTHOF operator counts the number of elements in an array.

It also takes into account nested DUP operators that multiply the counters.

If you declare an array that spans multiple program lines, LENGTHOF only regards the data from the first line as part of the array.

first line as part of the array.

```
.data
    byte1 BYTE 10,20,30
    array1 WORD 30 DUP(?)
    array2 WORD 5 DUP(3 DUP(?))
    array3 DWORD 1,2,3,4
    digitStr BYTE "12345678",0
```

; The following table lists the values returned by each LENGTHOF expression:

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30
LENGTHOF array2	5 * (3 * 3)
LENGTHOF array3	4
LENGTHOF digitStr	9

```
.data
    byte1 BYTE 10,20,30
    array1 WORD 30 DUP(?)
    array2 WORD 5 DUP(3 DUP(?))

.code
    mov ax, LENGTHOF byte1 ; AX = 3
    mov ax, LENGTHOF array1 ; AX = 30
    mov ax, LENGTHOF array2 ; AX = 15
```

## *SizeOF Operator:*

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE. For example:

```
.data
    intArray WORD 32 DUP(0)

.code
    mov eax, SIZEOF intArray ; EAX = 32 * 2 = 64
```

`IntArray` is indeed an array of 32 words. Each word is 16 bits or **2 bytes** in size. Therefore, the correct calculation for `SIZEOF intArray` should be:

```
SIZEOF intArray = 32 * 2 = 64
```

So, the correct value for `EAX` should be 64.

=====

## *Label Directive:*

=====

The `LABEL` directive lets you insert a label and give it a size attribute without allocating any storage. It's commonly used to provide an alternative name and size attribute for the variable declared next in the data segment.

```
.data
    val16 LABEL WORD
    val32 DWORD 12345678h

.code
    mov ax, val16 ; AX = 5678h
    mov dx, [val16+2] ; DX = 1234h
```

In this example, `val16` serves as an alias for the same storage location as `val32`. The `LABEL`

directive itself allocates no storage and is used to provide an alternate name and size for the variable.

```
.data
```

```
val16 LABEL WORD  
val32 DWORD 12345678h
```

```
mov ax, val16 ; AX = 5678h  
mov dx, [val16+2] ; DX = 1234h
```

```
LongValue LABEL DWORD  
val1 WORD 5678h  
val2 WORD 1234h
```

```
mov eax, LongValue ; EAX = 12345678h
```

---

The LABEL directive allows you to create a label and assign it a size attribute without actually allocating any storage. This can be useful for a few different things:

- Creating aliases for variables: You can use a LABEL directive to create a different name for a variable, with a different size attribute. This can be useful for making your code more readable and maintainable. For example, the following code creates a label named val16 that is an alias for the variable val32:

```
.data  
    val16 LABEL WORD  
    val32 DWORD 12345678h  
  
.code  
    mov ax, val16 ; AX = 5678h
```

In this example, the `val16` label is used to access the first two bytes of the `val32` variable.

- Constructing larger integers from smaller integers: You can also use a `LABEL` directive to construct a larger integer from two smaller integers. For example, the following code creates a label named `LongValue` that represents a 32-bit integer constructed from the two 16-bit variables `val1` and `val2`:

```
.data  
    LongValue LABEL DWORD  
    val1 WORD 5678h  
    val2 WORD 1234h  
  
.code  
    mov eax, LongValue ; EAX = 12345678h
```

In this example, the `LongValue` label is used to access the 32-bit integer that is constructed from

the val1 and val2 variables.

The LABEL directive does not actually allocate any storage. It simply creates a label with a specific size attribute. This can be useful for creating aliases for variables, constructing larger integers from smaller integers, and other purposes.

---

The LongValue label is not actually storing the values of val1 and val2. Instead, it's acting as a pointer to the memory location where val1 and val2 are stored. Here's what's happening step by step:

1. In the .data section, you declare LongValue as a LABEL DWORD, which effectively gives a name (LongValue) to a memory location that can hold a doubleword (32 bits or 4 bytes) of data.
2. You declare val1 and val2 as two 16-bit words with specific values (5678h and 1234h, respectively). These values are stored in memory in the order they appear.
3. In the .code section, you use the mov instruction to move the value at the memory location referred to by LongValue into the EAX register. This instruction effectively loads a doubleword (32 bits) from memory into EAX.

So, what's happening is that LongValue is acting as a reference or memory address, not a variable that stores the values directly. It's common in assembly language to use labels to represent memory addresses, and by using the label as an operand in an instruction, you can access the data stored at that memory location.

In this case, when you move from LongValue to EAX, you are effectively loading the 32 bits of data starting from the memory location referred to by LongValue, which includes the values of val1 and val2 stored consecutively in memory.

---

The critical concept to understand is memory addressing and how the LongValue label acts as a memory reference. Here's a detailed breakdown:

**Memory Allocation:** In the .data section, you declare LongValue as a LABEL DWORD. This essentially reserves a 32-bit (4-byte) block of memory without assigning any specific value to it. The memory layout looks like this:

Memory:

[ LongValue ] [ val1 ] [ val2 ]

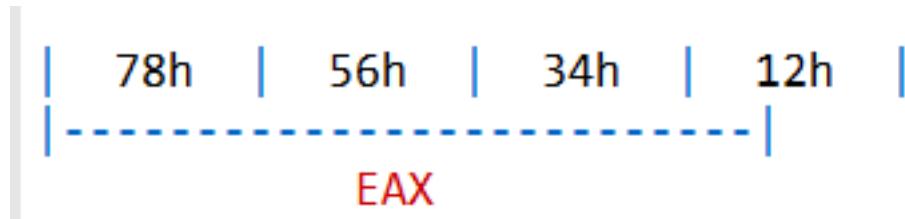
However, at this point, the values of val1 and val2 are stored in their respective memory locations.

**Memory Contents:** val1 is a 16-bit (2-byte) word with the value 5678h, and val2 is another 16-bit word with the value 1234h. These values are stored in memory in little-endian order, which means the least significant byte comes first:

Memory:

[ LongValue ] [ 78h ] [ 56h ] [ 34h ] [ 12h ]

**Using mov Instruction:** In the .code section, you use the mov instruction to move data from memory to a register. When you `mov eax, LongValue`, it means you're loading a 32-bit (4-byte) value into the EAX register, starting from the memory location referred to by LongValue. In essence, you're loading the consecutive bytes from that memory location into EAX.



So, EAX gets loaded with 12345678h, which is the concatenation of the values at the memory locations referred to by LongValue, val1, and val2.

So, assembler will check and see longvalue memory locatoin pointer, has nothing and continue to load the val1 and val2 that follow as the next instructions??

Yes, that's correct. In assembly language, the assembler doesn't "know" or store values for labels like LongValue.

Instead, it treats labels as memory addresses or pointers to specific locations in memory. When you use a label in an instruction, the assembler generates machine code that loads or manipulates data from the memory location pointed to by that label.

In the case of `mov eax, LongValue`, the assembler generates machine code that tells the CPU to:

1. Go to the memory location pointed to by the label LongValue.
2. Load the data from that memory location into the EAX register.

Since LongValue doesn't have a specific value assigned to it in the data section, the assembler doesn't "see" any data there. Instead, it continues to load the data from the memory locations that follow in memory, which in this case are val1 and val2.

So, when you `mov eax, LongValue`, you effectively load the data from val1 and val2 into the EAX register because they are the consecutive memory locations following LongValue.

In summary, the LongValue label acts as a reference to a memory location, and when you move it into EAX, you effectively load the 32 bits (4 bytes) of data starting from that memory location, which includes the values of val1 and val2. This demonstrates how labels can be used to access specific memory locations in assembly language programming.

Let's go through the answers to the section review questions:

**(True/False): The OFFSET operator always returns a 16-bit value.**

False. The OFFSET operator returns the offset or address of a variable or label, which can be either 16-bit or 32-bit, depending on the architecture and context.

**(True/False): The PTR operator returns the 32-bit address of a variable.**

True. The PTR operator is used to specify the size of an operand, and when used in conjunction with a label or variable, it often implies a 32-bit address.

**(True/False): The TYPE operator returns a value of 4 for doubleword operands.**

True. In many assembly languages, the TYPE operator returns a value that corresponds to the size of a data type in bytes. Doubleword operands are typically 4 bytes in size.

**(True/False): The LENGTHOF operator returns the number of bytes in an operand.**

False. The LENGTHOF operator returns the number of elements in an array or the length of a data

structure in terms of elements, not bytes.

**(True/False): The SIZEOF operator returns the number of bytes in an operand.**

True. The SIZEOF operator returns the size of an operand in bytes, including any padding or alignment bytes that may be present.

---

***IGNORE THESE QUESTIONS I HAVE EXPLAINED EVERYTHING ABOVE:***

---

**How the heck do the val1 and val2 values just fly inside LongValue, without something like a register??**

The values of val1 and val2 don't literally fly inside LongValue. The LongValue label is simply an alias/pointer, for the memory location that contains the 32-bit integer that is constructed from the val1 and val2 variables.

When you move the value of LongValue into the eax register, the CPU will read the 32-bit integer from the memory location that LongValue refers to.

In other words, the LongValue label is just a convenient way to refer to the memory location that contains the 32-bit integer. It does not actually store the integer itself.

Here is a diagram that shows how it works:

## Memory

0x0000: 5678

0x0002: 1234

LongValue (label)

eax (register)

When you move the value of LongValue into the eax register, the CPU will read the 32-bit integer from the memory location that LongValue refers to. In this case, the CPU will read the value 0x12345678 from the memory locations 0x0000 and 0x0002.

---

So, why did we just define the label, define val1, define val2, then poof, LongValue contains val1+val2 ?? I don't get what Label does?? Does it concatenate or what?

The LABEL directive does not concatenate anything. It simply creates a label with a specific size attribute.

In the example we were discussing, the LongValue label is used to represent the 32-bit integer that is constructed from the two 16-bit variables val1 and val2.

This is possible because the assembler knows the size of the val1 and val2 variables, and it knows

that they are contiguous in memory.

The assembler can therefore use this information to calculate the address of the 32-bit integer that is constructed from the two 16-bit variables.

When you move the value of LongValue into the eax register, the CPU will read the 32-bit integer from the memory location that LongValue refers to. In this case, the CPU will read the value 0x12345678 from the memory locations 0x0000 and 0x0002.

Here is a more detailed explanation of what happens when you assemble and link the code:

1. The assembler creates a symbol table. The symbol table is a list of all the labels and variables in the code, along with their addresses.
2. The assembler generates machine code for the code. The machine code contains instructions for the CPU to load the values of the variables val1 and val2 into registers, and then add the two values together.
3. The linker combines the machine code for the code with the machine code for any libraries that the code depends on.
4. The linker creates an executable file. The executable file contains the machine code for the code, as well as the symbol table.

When you run the executable file, the CPU will load the machine code into memory. The CPU will then use the symbol table to resolve the addresses of all the labels and variables in the code.

Once the CPU has resolved the addresses of all the labels and variables, it can start executing the machine code. The machine code will load the values of the variables val1 and val2 into registers,

and then add the two values together.

The result of the addition will be stored in a register. The CPU will then use the MOV instruction to move the value of the register to the memory location that the LongValue label refers to.

So, to answer your question, the LABEL directive does not concatenate anything. It simply creates a label with a specific size attribute. The assembler then uses this information to calculate the address of the memory location that the label refers to.

-----

Yes, your understanding is correct. The assembler will treat the LongValue label as a memory address, and when you use the `mov eax, LongValue` instruction, the assembler will generate machine code that tells the CPU to load the data from the memory location pointed to by the LongValue label into the EAX register.

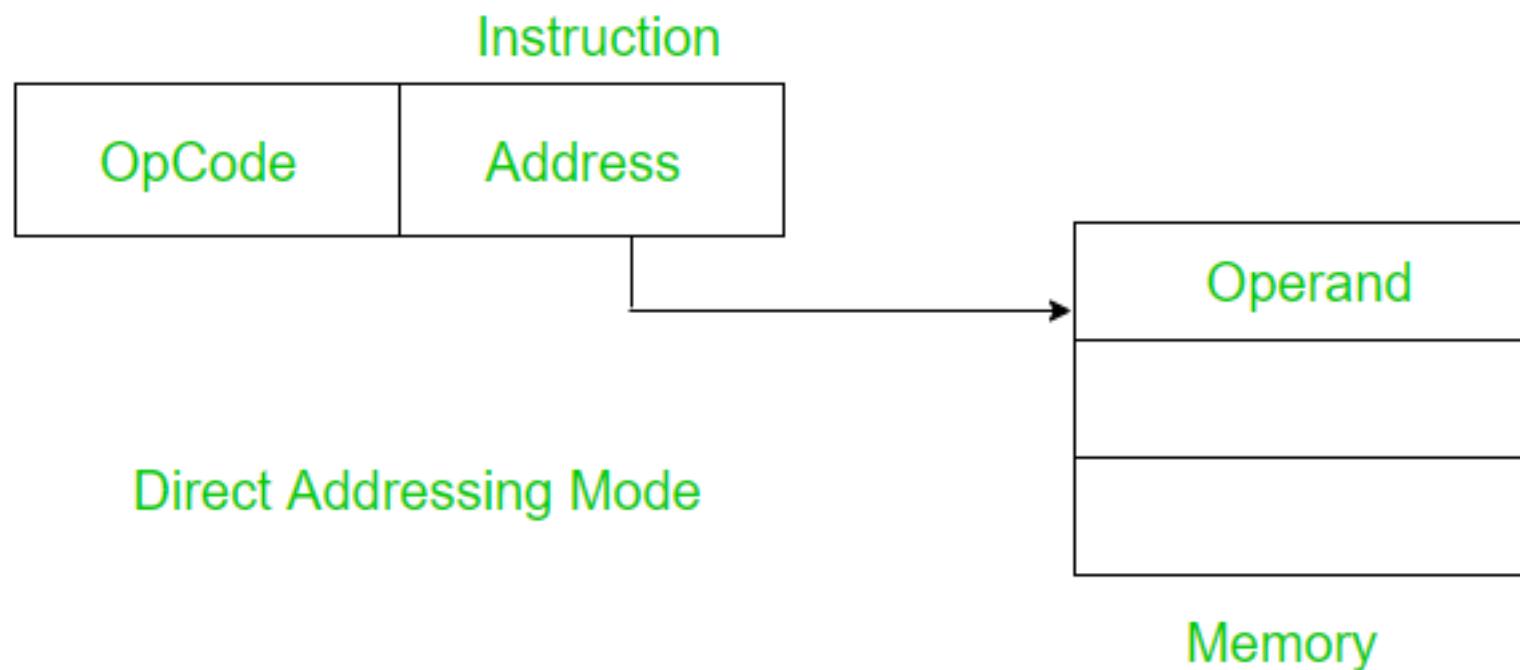
Since the LongValue label does not have a specific value assigned to it in the data section, the assembler will continue to load the data from the memory locations that follow in memory, which in this case are the val1 and val2 variables.

So, when you execute the `mov eax, LongValue` instruction, the EAX register will contain the value `0x12345678`, which is the 32-bit integer that is constructed from the two 16-bit variables val1 and val2.

Labels can be used to access specific memory locations in assembly language programming. This can be useful for a variety of purposes, such as accessing data structures, passing arguments to functions, and returning values from functions.

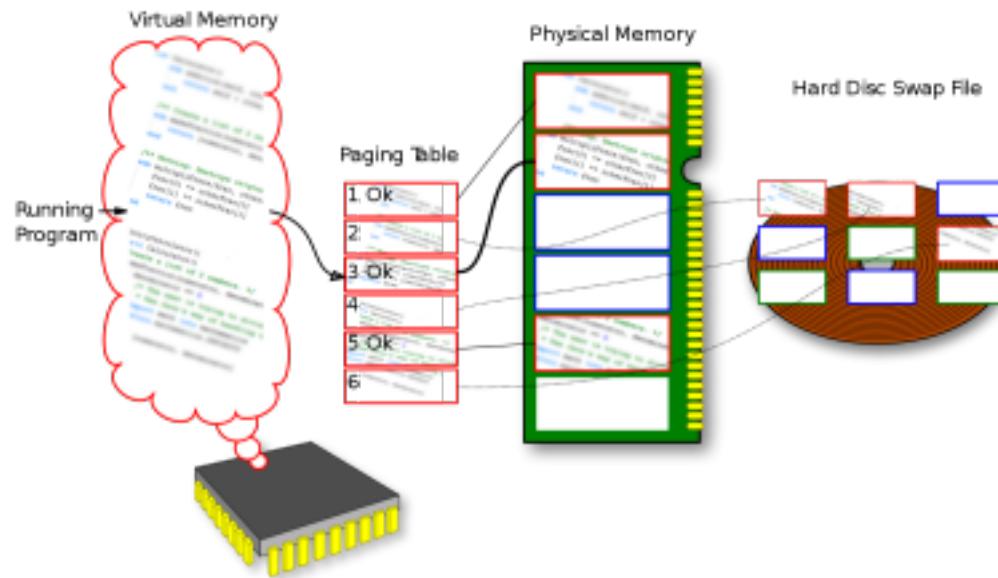
## **Direct Addressing**

**Direct addressing** is a memory addressing mode in which the operand of an instruction is the address of the data to be operated on. This means that the CPU can access the data directly, without having to perform any additional calculations.



Direct addressing is the simplest and most efficient addressing mode, but it has some limitations. One limitation is that the **address of the data must be known** at compile time. This means that direct

addressing cannot be used to access data that is stored dynamically, such as data that is read from a file.



Another limitation of direct addressing is that it can **only be used to access data that is within the address space of the CPU**. This means that direct addressing cannot be used to access data that is stored in external memory, such as memory on a hard disk drive.



Despite its limitations, direct addressing is a widely used addressing mode. It is often used to access data that is stored in static memory, such as global variables and constants.

Here are some examples of direct addressing in assembly language:

```
;Load the content of memory address 100 to register R1.  
LOAD R1, 100
```

```
;Load the content of register R2 to register R1.  
LOAD R1, R2
```

```
;Store the content of register R1 to memory address 200.  
STORE R1, 200
```

Direct addressing can also be used in high-level programming languages. For example, the following C code uses direct addressing to access an element of an array:

```
int main() {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    ;Load the element at index 5 of the array to the register EAX.
    EAX = array[5];

    ;Return the value in register EAX.
    return EAX;
}
```

The second line loads the element at index 5 of the array to the register EAX.

Only a **single memory reference** is required to access data. So no additional calculations are required to work out the effective address.

Disadvantage: The direct addressing mode will provide very limited space for the address.

Direct addressing is a powerful and versatile addressing mode, but it is important to be aware of its limitations.

# *Indirect Addressing*

---

## *Indirect Addressing*

---

Indirect addressing is a memory addressing mode in which the **operand of an instruction is the address of the data to be operated on**, but the address of the data is stored in a memory location or register.

This means that the **CPU must first dereference the pointer to the data** before it can access it.

Indirect addressing is more complex than direct addressing, but it has several advantages.

One advantage is that it allows the address of the data to be unknown at compile time.

This means that indirect addressing can be used to access data that is stored dynamically, such as data that is read from a file.

Another advantage of indirect addressing is that it allows the same code to be used to access different data items. This is useful for implementing functions such as loops and arrays.

Here are some examples of indirect addressing in assembly language:

```
; Load the content of memory address stored at memory address 100 to the register R1.  
LOAD R1, @100  
  
; Load the content of the memory address stored at register R2 to register R1.  
LOAD R1, @R2  
  
; Store the content of register R1 to the memory address stored at register R3.  
STORE R1, @R3
```

```
int main() {  
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
    int *pointer = &array[10];  
  
    ;Load the element at the address pointed to by the pointer to the register EAX.  
    EAX = *pointer;  
  
    ;Return the value in register EAX.  
    return EAX;  
}
```

Indirect addressing is a powerful and versatile addressing mode, but it is important to be aware of its limitations.

One limitation is that indirect addressing can cause the code to be slower than direct addressing, because the CPU has to perform an extra step to dereference the pointer.

Another limitation of indirect addressing is that it can lead to errors if the pointer is not initialized properly or if it points to an invalid memory location.

```
int *pointer = &array[5]; is it necessary to write that 5 in bracket? why not &array ??
```

It is not necessary to write the 5 in brackets when declaring the pointer pointer. The following two declarations are equivalent:

```
int *pointer = &array;  
int *pointer = &array[5];
```

The reason why you can omit the 5 in brackets is because arrays are implicitly converted to pointers to their first element. This means that the expression `&array` is equivalent to the expression `&array[0]`.

However, there are some cases where it is good practice to explicitly specify the array index. For example, if you are declaring a pointer to the last element of an array, it is more clear to write:

```
int *pointer = &array[9];
```

This makes it clear that the pointer is pointing to the last element of the array, and not to the

first element.

Another case where it is good practice to explicitly specify the array index is if you are using the pointer in a function or library that expects the pointer to point to a specific element of the array.

In general, it is up to you to decide whether or not to explicitly specify the array index when declaring a pointer. However, it is important to be aware of the implications of omitting the array index.

---

Another example:

```
.data  
    byteVal BYTE 10h  
.code  
    mov esi, OFFSET byteVal  
    mov al, [esi]           ;al = 10h
```

The first line declares a byte variable named **byteVal** and initializes it with the value **0x10**.

The second line moves the offset of the variable **byteVal** into the register **esi**.

The offset of a variable is the address of the variable relative to the beginning of the data segment.

In other words, the offset of a variable is the number of bytes from the beginning of the data segment to the first byte of the variable.

In the example you provided, the variable `byteVal` is declared in the data segment.

This means that the offset of the variable `byteVal` is the address of the variable `byteVal` relative to the beginning of the data segment.

The following assembly language instruction moves the offset of the variable `byteVal` into the register `esi`:

```
mov ESI, OFFSET byteVal
```

After this instruction is executed, the register `esi` will contain the offset of the variable `byteVal`.

The offset of a variable can be used to load or store the value of the variable. For example, the following assembly language instruction loads the value of the variable `byteVal` into the register `al`:

```
mov al, [ESI]
```

The offset of a variable is also used by the compiler to generate machine code for function calls. When a function is called, the compiler generates machine code to push the arguments to the function onto the stack. The compiler also generates machine code to call the function.

When the function returns, the compiler generates machine code to pop the arguments to the function from the stack. The compiler also generates machine code to return from the function.

The compiler uses the offset of the function arguments to generate machine code to push the arguments onto the stack. The compiler also uses the offset of the function return value to generate machine code to pop the return value from the stack.

In summary, the offset of a variable is the address of the variable relative to the beginning of the data segment. The offset of a variable can be used to load or store the value of the variable. The offset of a variable is also used by the compiler to generate machine code for function calls.

-----

In this code, we have a variable `byteVal` declared in the `.data` section with the value `10h` (which is 16 in decimal).

The `OFFSET` keyword is used to calculate the memory address of `byteVal` and store that address in the `esi` register. Here's how the math works:

1. `byteVal` is declared as a byte variable, which means it occupies one byte in memory.
2. When you use `OFFSET byteVal`, the assembler calculates the memory address where `byteVal` is located. **Since `byteVal` is the first data item declared in the `.data` section, it's at the beginning of the data segment.**
3. In x86 assembly, memory addresses start at `0x0` for the beginning of the data segment. Therefore, the memory address of `byteVal` is `0x0`.

4. The `OFFSET` keyword essentially says, "**Give me the memory address of `byteVal` relative to the start of the data segment.**" Since `byteVal` is at the start of the data segment, the relative address is `0x0`.

5. The `mov esi, OFFSET byteVal` instruction places the value `0x0` (the memory address of `byteVal`) into the `esi` register.

Now, with `esi` containing `0x0`, the next instruction `mov al, [esi]` reads the byte value at memory address `0x0`, which is the value of `byteVal`, and stores it in the `al` register. So, `al` is set to **`10h`** (or **16** in decimal).

In summary, the `OFFSET` keyword calculates the relative memory address of `byteVal` within the data segment, which is `0x0`, and that address is loaded into the `esi` register. This allows subsequent instructions to access the value of `byteVal` in memory.

```
.data
arrayD DWORD 10000h, 20000h, 30000h ; Define an array of doublewords

.code
mov esi, OFFSET arrayD      ; Load the memory address of the array into ESI

; Access the first number
mov eax, [esi]            ; Load the first doubleword into EAX
; ESI points to the first doubleword

; Access the second number
add esi, 4                ; Move ESI to the next doubleword (increment by 4 bytes)
mov eax, [esi]            ; Load the second doubleword into EAX
; ESI points to the second doubleword

; Access the third number
add esi, 4                ; Move ESI to the next doubleword (increment by 4 bytes)
mov eax, [esi]            ; Load the third doubleword into EAX
; ESI points to the third doubleword
```

This code defines an array of three doublewords (arrayD) and then uses the OFFSET keyword to obtain the memory address of the array.

It then accesses each doubleword within the array sequentially by updating the esi register to point to the next doubleword using the add esi, 4 instruction before loading its value into the eax register.

register.

Suppose arrayD is located at offset 10200h.

Offset	Value
10200	10000h
10204	20000h
10208	30000h

← [esi]  
← [esi] + 4  
← [esi] + 8

## ***Indexed Addressing***

---

### ***Indexed Addressing***

---

Indexed addressing is a memory addressing mode in which the **address of the operand is calculated by adding a constant to the contents of an index register**.

The index register is a special register in the CPU that is used to store a value that is used to

offset the address of the operand.

Indexed addressing is useful for accessing elements of an array.

For example, the following assembly language instruction loads the element at index 5 of the array array to the register EAX:

**LOAD R1, 5(R2)**

The actual location of an operand which is stored in memory is the **effective address**, which can be calculated using several addressing modes.

This instruction calculates the **effective address** of the operand by adding the constant 5 to the contents of the index register R2. The value in the register R2 is the offset of the array array.

Indexed addressing can also be used to access other data structures, such as linked lists and trees.

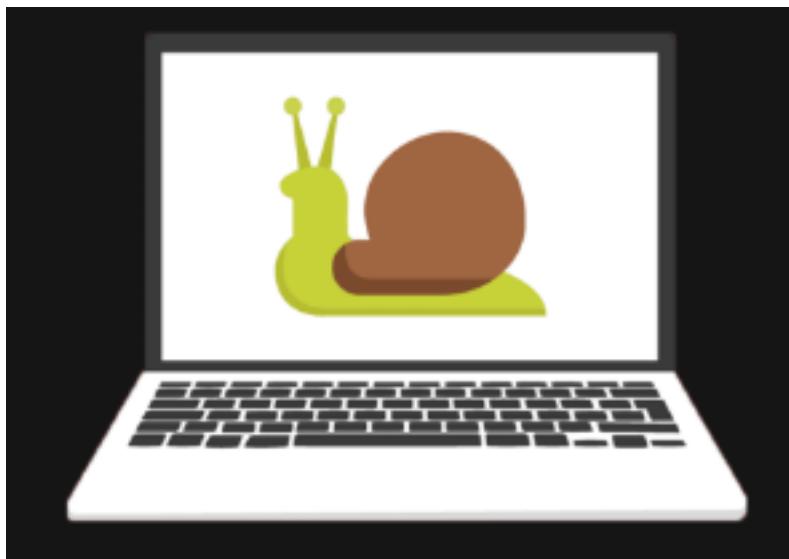
For example, the following assembly language instruction loads the next element in a linked list to the register EAX:

**LOAD R1, [EAX + 4]**

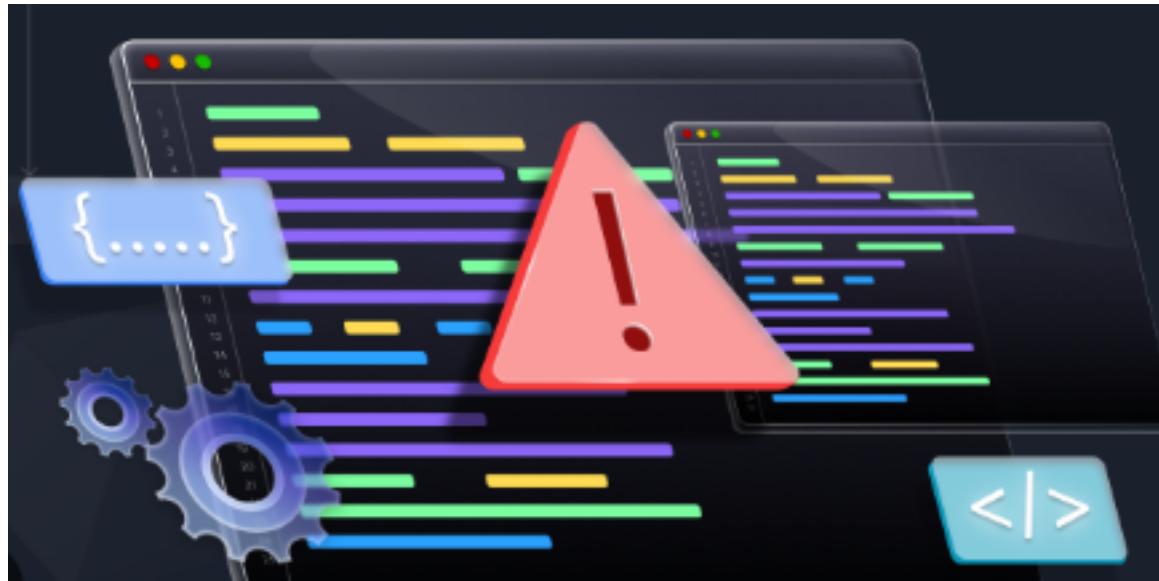
The value in the register EAX is the address of the current element in the linked list. So, this instruction calculates the effective address of the operand by adding the constant 4 to the contents of the register EAX.

Indexed addressing is a powerful and versatile addressing mode, but it is important to be aware of its limitations.

One limitation is that indexed addressing can cause the **code to be slower** than direct addressing, because the CPU has to perform an extra step to calculate the effective address of the operand.



Another limitation of indexed addressing is that it can **lead to errors** if the index register is not initialized properly or if it contains an invalid value



## *Indexed Operands*

Indexed operands are a way to access memory locations using a register and a constant. The register is called the index register, and the constant is called the displacement. The effective address of the operand is calculated by adding the displacement to the value of the index register.

Indexed operands are useful for accessing elements of an array, because they allow you to access any element of the array without having to know the address of the element in advance. For example, the following assembly language instruction loads the element at index 5 of the array array to the register EAX:

```
;Load the element at index 5 of the array `array` to the register `EAX`.
LOAD R1, 5(R2)

;Load the next element in a linked list to the register `EAX`.
LOAD R1, [EAX + 4]

;Store the value of the register `EAX` to the memory location
;at the offset 100 from the base address stored in the register `R4`.
STORE R1, 100(R4)
```

The following C code uses indexed operands to access an element of an array:

```
int main() {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int index = 5;

    ;Load the element at index `index` of the array `array` to the register `EAX`.
    int element = array[index];

    ;Return the value in the register `EAX`.
    return element;
}
```

## ***Indexed Operands with Scale Factors Part 2***

Indexed operands in assembly language allow you to add a constant offset to a register to calculate an effective memory address. This addressing mode is particularly useful for working with arrays.

Here, we'll delve into the concept of indexed operands with scale factors, which provides flexibility when accessing array elements of different sizes.

**Basic Indexed Operand Forms:** Indexed operands can take two notational forms, both utilizing a register and potentially a constant:

```
constant[reg]  
[constant + reg]
```

The first form combines a variable name with a register, and the assembler translates the variable's name into a constant representing its offset. Here are examples of both notational forms:

```
.data  
arrayB BYTE 10h, 20h, 30h
```

```
.code  
mov esi, 0  
mov al, arrayB[esi] ; AL = 10h  
mov al, [arrayB + esi]
```

The code you provided is intended to access and retrieve the value of the first element in an array called `arrayB`, which is defined in the `.data` section as an array of bytes with the values `10h`, `20h`, and `30h`. Let's break down what each line does:

**.data:** This section defines the data for the program. Here, you've defined an array named `arrayB` of type `BYTE`, containing three values: `10h`, `20h`, and `30h`.

**.code:** This section contains the executable code for your program.

**mov esi, 0:** This instruction initializes the `esi` register to zero. The `esi` register is often used as an index or pointer in assembly language.

**mov al, arrayB[esi]:** Here, you're using indexed addressing to access the value of the first element of the `arrayB`. The `[esi]` part is an index operation that says, "**Access the element in `arrayB` that's at the memory location specified by the value in `esi`.**" Since you previously set `esi` to zero, this effectively retrieves the value at the beginning of the `arrayB`, which is `10h`.

**mov al, [arrayB + esi]:** This line does the same thing as the previous line. It retrieves the value at the memory location specified by `arrayB + esi`.

So, in summary, these instructions initialize `esi` to zero and then use it as an index to access and load the value `10h` from the first element of the `arrayB` into the `al` register. After this code is executed, `al` will contain the value `10h`.

-----

**Initializing the Index Register:** When working with indexed operands, it's crucial to initialize the index register to zero before accessing the first element in the array. This allows you to start at

the beginning of the array.

```
.data  
    arrayB BYTE 10h, 20h, 30h  
  
.code  
    mov esi, 0          ; Initialize ESI to zero  
    mov al, arrayB[esi] ; AL = 10h (first element)
```

**Adding Displacements:** The second type of indexed addressing involves combining a register with a constant offset. Here, the index register holds the base address of an array or structure, and the constant specifies offsets to various array elements. Consider this example with a 16-bit word array:

```
.data  
    arrayW WORD 1000h, 2000h, 3000h  
  
.code  
    mov esi, OFFSET arrayW  
    mov ax, [esi]        ; AX = 1000h (first element)  
    mov ax, [esi + 2]    ; AX = 2000h (second element)  
    mov ax, [esi + 4]    ; AX = 3000h (third element)
```

## *Scale Factors in Indexed Operands*

To work with indexed operands more flexibly, especially when the size of array elements varies, you can use scale factors. A scale factor is the size of each array component (e.g., word = 2, doubleword = 4).

Here's an example with a doubleword array, where we multiply the subscript (3) by the scale factor (4) for doublewords to access the element containing 400h:

```
.data  
    arrayD DWORD 100h, 200h, 300h, 400h  
  
.code  
    mov esi, 3 * TYPE arrayD ; Calculate the offset for arrayD[3]  
    mov eax, [arrayD + esi] ; EAX = 400h
```

$3 * 4\text{bytes} = 12$

$0 + 12(\text{esi}) = 12$

$\text{eax} = 12 = 400\text{h}$ 's location

- 
1. You're correct that 3 multiplied by 4 (bytes) equals 12. This multiplication takes into account the size of each array element.
  2. Setting esi to 12 means that you're trying to access the element in the array that is 12 bytes

from the start of the array. However, it's important to note that in assembly language, the array elements are typically indexed starting from zero. So, `arrayD[0]` is the first element, `arrayD[1]` is the second element, and so on.

3. When you set `esi` to 12, you're effectively trying to access `arrayD[3]`, not `arrayD[0]`. If you want to access the element containing `400h`, you should set `esi` to 0, not 12, because `arrayD[0]` is the first element in the array.

So, with `esi` set to 0, you'll access the first element of `arrayD`, which indeed contains `100h`, not `400h`. If you want to access `arrayD[3]` (the element with `400h`), you would set `esi` to 12.

**With `esi` set to 12,** you are correctly accessing the fourth element of `arrayD`, which contains `400h`. The code loads the value `400h` into the `eax` register. Thank you for pointing out the correct setting of `esi`, and I appreciate your patience.

## ***Immediate Addressing***

Immediate addressing mode involves specifying a constant value as an operand directly within an instruction.

It's typically used for providing immediate data to instructions like add, subtract, or load.

In the immediate addressing mode, we specify the operand in the instruction itself.

```
.data
    constant_value DWORD 42
    direct_value DWORD 10

.code
main PROC
    ; Immediate Addressing
    mov eax, 100      ; Load the immediate value 100 into EAX
    add eax, constant_value ; Add the constant value to EAX
    ; EAX now contains the result of 100 + 42 = 142

    ; Direct Addressing
    mov ebx, [direct_value] ; Load the value at the memory location pointed to by direct_value into EBX
    ; EBX now contains the value stored at the memory location pointed to by direct_value

    ; Exit the program
    invoke ExitProcess, 0
main ENDP
END main
```

In this code:

Immediate addressing is demonstrated with `mov eax, 100`. Here, the immediate value 100 is directly loaded into the EAX register.

Direct addressing is demonstrated with `mov ebx, [direct_value]`. Here, the value at the memory location pointed to by the `direct_value` variable is loaded into the EBX register.

These two instructions showcase the difference between immediate addressing (where the value is directly specified in the instruction) and direct addressing (where the value is loaded from a

memory location specified by a variable).

Differentiating the 5 addressing modes we've discussed:

```
.data
    array DWORD 10, 20, 30, 40 ; An array for indexed addressing
    value DWORD 100           ; A constant value for immediate addressing
    ptr DWORD offset array   ; A pointer for indirect addressing
    stackVal DWORD ?        ; A variable for stack addressing

.code
main PROC
    ; Direct Addressing
    mov eax, [array]    ; Load the value at the memory location pointed to by array into EAX

    ; Indirect Addressing
    mov ebx, [ptr]      ; Load the value at the memory location pointed to by ptr into EBX

    ; Indexed Addressing
    mov ecx, [array + 4] ; Load the value at the memory location (array + 4) into ECX
    ; This demonstrates indexed addressing by accessing the second element of the array

    ; Immediate Addressing
    mov edx, value       ; Load the immediate value 100 into EDX

    ; Stack Addressing
    push 200            ; Push the immediate value 200 onto the stack
    pop stackVal        ; Pop the value from the stack into stackVal

    ; Exit the program
    invoke ExitProcess, 0
main ENDP
END main
```

In this code:

**Direct addressing** is demonstrated with `mov eax, [array]`. It loads the value at the memory location pointed to by array into EAX.

**Indirect addressing** is demonstrated with `mov ebx, [ptr]`. It loads the value at the memory location pointed to by ptr into EBX.

**Indexed addressing** is demonstrated with `mov ecx, [array + 4]`. It accesses the second element of the array by adding an offset of 4 bytes to array.

**Immediate addressing** is demonstrated with `mov edx, value`. It loads the immediate value 100 into EDX.

**Stack addressing** is demonstrated with `push 200` and `pop stackVal`. It pushes the immediate value 200 onto the stack and then pops it into the stackVal variable.

## ***Stack Addressing***

---

## ***Stack Addressing***

=====

Stack addressing is an essential concept in computer architecture and assembly language programming, especially in stack-oriented architectures like the x86.

It involves using a special region of memory known as the stack to manage data and control program flow. Let's dive into the basics of stack addressing:

The stack is a region of memory that is used for temporary storage and organization of data. It is a Last-In, First-Out (LIFO) data structure, meaning that the last item pushed onto the stack is the first one to be removed (popped) from it.

**Stack Pointer (SP):** The stack is managed using a special register called the stack pointer (SP). The SP points to the current top of the stack. When data is pushed onto the stack, the SP is decremented, and when data is popped from the stack, the SP is incremented.

-----

## ***Stack Operations***

**Push:** The push operation places data onto the stack. It involves decrementing the SP and storing the data at the memory location pointed to by the SP.

**Pop:** The pop operation retrieves data from the stack. It involves accessing the data at the memory location pointed to by the SP, incrementing the SP, and using the retrieved data.

In assembly language programming, you can use instructions like **PUSH** and **POP** to work with the stack. Here's a simple example in x86 assembly:

```
PUSH EAX ; Push the value in EAX onto the stack  
POP EBX ; Pop the top value from the stack into EBX
```

In the above code, PUSH EAX pushes the value in the EAX register onto the stack, and POP EBX pops the top value from the stack and stores it in the EBX register.

---

### ***Stack for Function Calls***

One of the most common uses of the stack in assembly language is managing function calls. When a function is called, its return address, parameters, and local variables are often pushed onto the stack. When the function returns, these values are popped from the stack.

Here's a simplified example of a function call in assembly:

```
; Call a function, Inside the function, it may push and pop values onto the stack  
CALL MyFunction  
; Return from the function, which pops the return address from the stack  
RET
```

The CALL instruction pushes the return address onto the stack, allowing the program to return to the right location after the function completes.

---

## ***CALL MyFunction:***

The CALL instruction is used to call a function, in this case, a function named MyFunction. When a function is called, the following typically happens:

The return address, which is the memory address of the instruction right after the CALL instruction, is pushed onto the stack.

Execution of the program jumps to the first instruction of the MyFunction function.

This allows the program to return to the correct location in the code after MyFunction finishes executing.

## ***Inside the function:***

Inside the MyFunction function, there may be instructions that push values onto the stack and instructions that pop values from the stack.

These operations are used for various purposes:

**Pushing values onto the stack:** This can be done to save registers, pass function parameters, or store local variables.

**Popping values from the stack:** This is done to retrieve previously pushed values, such as parameters or saved registers, for use within the function.

## **RET:**

The RET (return) instruction is used to exit a function and return control to the calling code.

When RET is executed, it does the following:

- Pops the top value from the stack, which is the return address that was pushed onto the stack when the function was called.
- Jumps to the memory address stored in the popped return address, which is the location right after the original CALL instruction. This allows the program to continue executing from where it left off before the function call.

In summary, the code snippet you provided demonstrates the typical flow of calling a function using the CALL instruction, where the return address is pushed onto the stack, and the RET instruction is used to pop the return address from the stack and resume execution in the calling code.

Inside the function, the stack can also be used to manage data and variables, and these operations depend on the specific requirements of the function.

---

```
;Push the argument to the function `add()` onto the stack.
```

```
PUSH EAX
```

```
;Call the function `add()`.
```

```
CALL add
```

```
;The return value of the function `add()` is now on the top of the stack.
```

```
;Pop the return value off the stack and store it in the register `EAX`.
```

```
POP EAX
```

Stack addressing is a powerful and versatile addressing mode, but it can be a bit confusing to understand at first. Here are some tips for understanding stack addressing:

- The stack is a last-in-first-out (LIFO) data structure. This means that the last operand pushed onto the stack is the first operand popped off the stack.
- The stack pointer is a register that points to the top of the stack.
- When a function is called, the arguments to the function are pushed onto the stack.
- When a function returns, the return value is pushed onto the stack.
- To access an operand on the stack, you must first pop the operand off the stack.

Stack addressing is a very important concept in assembly language programming, because it is used to implement function calls and returns. It is also used to perform operations on multiple operands.

Here is an example of how stack addressing is used to perform an operation on multiple operands:

; Push the first operand onto the stack.

PUSH EAX

; Push the second operand onto the stack.

PUSH EBX

; Add the two operands on the stack and store the result in the register `EAX`.

ADD EAX, [ESP] ; ESP is the stack pointer

; Pop the second operand off the stack.

POP EBX

; Pop the first operand off the stack.

POP EAX

**PUSH EAX:** The PUSH instruction pushes the value in the EAX register onto the stack. This operation effectively saves the value of EAX onto the stack.

**PUSH EBX:** The PUSH instruction pushes the value in the EBX register onto the stack. This operation saves the value of EBX onto the stack.

**ADD EAX, [ESP]:** Here, we're adding the two values on the stack. The value at the top of the stack

(pointed to by ESP) is added to the value in the EAX register. This operation is essentially adding the two values that were pushed onto the stack.

**POP EBX:** The POP instruction pops (removes) the top value from the stack and stores it in the EBX register. This operation retrieves the value of the second operand from the stack.

**POP EAX:** Finally, another POP instruction is used to pop the top value from the stack and store it in the EAX register. This operation retrieves the result of the addition from the stack.

In summary, this code demonstrates a somewhat unconventional way of performing addition using the stack. It pushes the operands onto the stack, adds them together by manipulating the values in registers, and then retrieves the result from the stack. This approach can be used for various purposes, but it's not the most common way to perform arithmetic operations in assembly language.

The code you provided assumes that the EAX and EBX registers already contain values before the push operations. In assembly language, it's the responsibility of the programmer to load values into registers before performing operations on those values.

```
.data
value1 DWORD 10      ; Define a 32-bit integer with the value 10
value2 DWORD 20      ; Define another 32-bit integer with the value 20

.code
main PROC
    ; Push the first value onto the stack
    PUSH DWORD PTR [value1]

    ; Push the second value onto the stack
    PUSH DWORD PTR [value2]

    ; Call a function to add the two values
    CALL add_values

    ; The result is now in EAX; you can use it or print it
    ; For demonstration purposes, we'll print the result
    MOV EAX, DWORD PTR [ESP]    ; Load the result from the stack to EAX
    CALL print_result

    ; Clean up the stack
    ADD ESP, 8     ; Remove the two values (4 bytes each)

    ; Exit the program
    MOV EAX, 1      ; Exit code 1
    INT 20h        ; DOS system call for program exit
main ENDP
```

```
; Function to add two values
add_values PROC
    POP EBX      ; Pop the second value into EBX
    POP EAX      ; Pop the first value into EAX
    ADD EAX, EBX ; Add the values, result in EAX
    PUSH EAX     ; Push the result back onto the stack
    RET
add_values ENDP

; Function to print the result (placeholder)
print_result PROC
    ; (Code for printing the value goes here - platform-dependent)
    RET
print_result ENDP
```

Certainly, here's a comprehensive explanation of the MASM assembly code:

### ***Data and code sections:***

In the .data section, we define two 32-bit integers: value1 and value2. These variables are allocated memory to store integer values. value1 is initialized with 10, and value2 is initialized with 20. Code Section:

In the .code section, we begin defining the main procedure with main PROC. This marks the entry

point of our program. Pushing Values onto the Stack:

We use the PUSH instruction to push the values of value1 and value2 onto the stack. These values are pushed as double words (32-bit integers) because we use DWORD PTR. Function

### ***Call:***

Next, we call a function named add\_values using the CALL instruction. This function is responsible for adding the two values we pushed onto the stack. Retrieving Result:

After the add\_values function call, the result of the addition is left on top of the stack. We retrieve this result by using MOV to move it from the stack to the EAX register. Printing Result (Placeholder):

We have a placeholder print\_result function call. In a complete program, this is where you would typically include code to print the result. The actual code for printing depends on the specific platform or environment you are working with. Cleaning Up the

### ***Stack:***

We use ADD ESP, 8 to clean up the stack by adjusting the stack pointer (ESP). This removes the two values we pushed earlier (each being 4 bytes, so 8 bytes in total).

### ***Exiting the Program:***

Finally, we set the exit code to 1 using MOV EAX, 1 and use the INT 20h instruction to exit the program. This is a DOS system call for program exit. add\_values Function:

### ***Within the add\_values function:***

We use POP to pop the values from the stack back into registers. First, we pop the second value (value2) into EBX, and then we pop the first value (value1) into EAX. We add the values in EAX and EBX and store the result in EAX. Finally, we push the result back onto the stack to make it available to the calling code.

### ***RET (Return):***

The RET instruction is used to return from a function. It transfers control back to the calling code, and the program continues executing from where the function was called.

In summary, this assembly program demonstrates how to perform a basic addition operation using the stack. It defines data, pushes values onto the stack, calls a function to perform the addition, retrieves and potentially prints the result, cleans up the stack, and exits the program.

```
#include <stdio.h>

int addValues(int a, int b) {
    return a + b;
}

int main() {
    int value1 = 10;
    int value2 = 20;

    int result = addValues(value1, value2);

    printf("Result: %d\n", result);

    return 0;
}
```

That's the C equivalent.

In this C code:

We define a function `addValues` that takes two integers as input, adds them, and returns the result. This function corresponds to the `add_values` function in the assembly code.

In the `main` function, we declare two integers `value1` and `value2` and assign them the values 10 and

20, respectively. These correspond to the value1 and value2 variables defined in the assembly code.

We call the addValues function to add value1 and value2, and store the result in the result variable.

Finally, we print the result using printf.

This C code accomplishes the same addition operation as the MASM assembly code, but it uses C's higher-level syntax and functions for the same purpose.

## ***Pointers In ASM***

A pointer in assembly language is a variable that stores the memory address of another variable or data structure.

Pointers are a powerful tool for manipulating arrays and data structures because they allow you to access and modify data at runtime by working with memory addresses.

### ***Pointer Size:***

The size of a pointer in assembly language depends on the processor's current mode, which can be 32-bit or 64-bit. In the following example, we'll focus on 32-bit code.

```
.data  
arrayB BYTE 10H, 20, 30h, 40h  
ptrB DWORD arrayB
```

In this code snippet, we have an array called arrayB, which contains four bytes of data. The ptrB variable is declared as a doubleword (32 bits) and stores the memory offset of arrayB. Optionally, you can use the OFFSET operator to make the relationship between the pointer and the array clearer:

```
.data  
arrayB BYTE 10H, 20, 30h, 40h  
ptrB DWORD OFFSET arrayB
```

## Near Pointers

In 32-bit assembly code, we often use near pointers, which are stored in doubleword variables. For example:

```
arrayB BYTE 10h, 20h, 30h, 40h  
arrayW WORD 1000h, 2000h, 3000h  
ptrB DWORD arrayB  
ptrW DWORD arrayW
```

Here, we have two arrays: arrayB of bytes and arrayW of words. We declare pointers ptrB and ptrW as doublewords to store the memory offsets of these arrays. Optionally, you can use the OFFSET operator

for clarity:

```
arrayB BYTE 10h, 20h, 30h, 40h
arrayW WORD 1000h, 2000h, 3000h
ptrB DWORD OFFSET arrayB
ptrW DWORD OFFSET arrayW
```

## Understanding Pointers

In high-level languages, the details of pointers are often abstracted because they can vary between different machine architectures.

However, in assembly language, we deal with pointers at a more physical level, allowing us to work directly with memory addresses. This approach helps demystify pointers and provides more control over memory management.

Overall, pointers in assembly language are crucial for dynamic memory manipulation and efficient data access. They allow you to interact with memory locations directly, which is essential for low-level programming tasks.

## *Summary:*

The notes you provided are a bit unclear and disorganized, but they can be summarized as follows:

- A **pointer** is a variable that contains the address of another variable.
- The **PTR keyword** tells the assembler that the variable `myPointer` is a pointer.
- Pointers are useful for manipulating arrays and data structures because the address they hold can

be modified at runtime.

- In 32-bit mode, pointers are stored in doubleword variables.
- You can use the OFFSET operator to make the relationship between a pointer and the variable it points to clearer.

Pointers are useful for a variety of reasons, including:

- They allow you to manipulate arrays and data structures in a more efficient way.
- They allow you to dynamically allocate memory at runtime.
- They allow you to pass arguments to functions by reference.
- They allow you to implement function pointers.

```
; Declare a pointer to a byte variable.
```

```
myPointer BYTE PTR
```

```
; Store the address of the byte variable `myByte` in the pointer `myPointer`.
```

```
MOV myPointer, OFFSET myByte
```

```
; Load the value of the byte variable pointed to by `myPointer` into the register `EAX`.
```

```
MOV EAX, [myPointer]
```

```
; Increment the value of the byte variable pointed to by `myPointer`.
```

```
INC [myPointer]
```

The first line declares a pointer to a byte variable. The BYTE PTR keyword tells the assembler that the variable myPointer is a pointer to a byte variable.

The second line stores the address of the byte variable `myByte` in the pointer `myPointer`. The `OFFSET` operator returns the address of the variable `myByte`.

The third line loads the value of the byte variable pointed to by `myPointer` into the register `EAX`. To do this, the assembler dereferences the pointer `myPointer` using the `[]` operator.

The fourth line increments the value of the byte variable pointed to by `myPointer`. To do this, the assembler dereferences the pointer `myPointer` using the `[]` operator and then increments the value of the byte variable at that address.

## ***TYPEDEF operator to create pointer types***

The `TYPEDEF` operator allows you to create a user-defined type that has all the status of a built-in type when defining variables. This is useful for creating pointer types, as it makes the code more readable and maintainable.

For example, the following declaration creates a new data type called `PBYTE` that is a pointer to bytes:

```
PBYTE TYPEDEF PTR BYTE
```

This declaration would usually be placed near the beginning of a program, before the data segment.

Then, variables could be defined using the PBYTE type:

```
PBYTE TYPEDEF PTR BYTE

.data
    arrayB BYTE 10h,20h,30h,40h
    ptr1 PBYTE ? ; uninitialized
    ptr2 PBYTE arrayB ; points to an array
```

### *Advantages of using the TYPEDEF operator to create pointer types:*

It makes the code more readable and maintainable, as it is clear what type of data the pointer is pointing to.

It can help to prevent errors, as the compiler will check that the pointer is only used to access the type of data that it is pointing to.

It can make the code more portable, as the same TYPEDEF declaration can be used on different platforms.

The following program demonstrates how to use the TYPEDEF operator to create pointer types:

```
.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

; Create user-defined types.
PBYTE TYPEDEF PTR BYTE ; pointer to bytes
WORD TYPEDEF PTR WORD ; pointer to words
DWORD TYPEDEF PTR DWORD ; pointer to doublewords

.data
    arrayB BYTE 10h,20h,30h,40h
    arrayW WORD 1,2,3
    arrayD DWORD 4,5,6

    ; Create some pointer variables.
    ptr1 PBYTE arrayB
    ptr2 WORD arrayW
    ptr3 DWORD arrayD

.code
main PROC

    ; Use the pointers to access data.
    mov esi, ptr1
    mov al, [esi] ; AL = 10h
    mov esi, ptr2
    mov ax, [esi] ; AX = 1
    mov esi, ptr3
    mov eax, [esi] ; EAX = 4

    invoke ExitProcess, 0

main ENDP
END main
```

The TYPEDEF operator is a powerful tool that can be used to create pointer types. This can make the code more readable, maintainable, portable, and error-prone.

The JMP and LOOP instructions in assembly language are used to implement conditional and unconditional transfers of control, respectively.

The JMP instruction causes an unconditional transfer to a destination, identified by a code label.

The LOOP instruction repeats a block of statements a specific number of times, using the ECX register as a counter.

Here is an example of how to use the JMP and LOOP instructions to implement a simple loop:

```
loop_start:  
    ;loop body  
    ...  
    ;check the loop condition  
    cmp ecx, 0  
    je loop_end  
  
    dec ecx ;decrement the loop counter  
    jmp loop_start ;jump to the beginning of the loop  
  
loop_end  
    ;loop's exit code
```

This loop will execute as long as the ECX register is not equal to zero. When ECX reaches zero, the loop will exit.

The LOOP instruction is a more concise and efficient way to implement a loop than using a JMP instruction.

The LOOP instruction automatically decrements the ECX register and performs the jump to the loop label, so you don't have to explicitly write these instructions yourself.

Here is an example of how to use the LOOP instruction to implement the same loop as above:

```
loop_start:  
; loop body  
...  
  
; loop condition  
cmp ecx, 0  
je loop_end  
  
; loop  
loop loop_start  
  
loop_end:  
; loop exit code
```

This loop is equivalent to the previous loop, but it is more concise and efficient.

You can also use the LOOP instruction to implement more complex loops, such as loops that iterate over an array or loops that exit based on a different condition.

```
.data
    count DWORD ?

.code
    mov ecx, 100 ; set loop count
    top:
        mov count, ecx ; save the count
        ; modify ECX
        mov ecx, count ; restore loop count
    loop top
    ; loop exit code
```

The loop count is initialized to 100 by moving the value 100 into the ECX register.

The loop counter is saved in the count variable by moving the value of the ECX register to the count variable.

The ECX register is modified. The loop counter is restored by moving the value of the count variable to the ECX register.

The LOOP instruction is executed, which decrements the ECX register and jumps to the top label if the ECX register is not equal to zero.

The loop exit code is executed if the ECX register is equal to zero. The following is a more detailed explanation of each step:

**top:**

This label marks the beginning of the loop.

**loop top**

This instruction decrements the ECX register and jumps to the top label if the ECX register is not equal to zero. This causes the loop to repeat until the ECX register is equal to zero.

=====

## ***Nested Loops***

=====

```
.data
    count DWORD ?
.code
    ; set outer loop count
    mov ecx, 100

L1:
    ; save outer loop count
    mov count, ecx

    ; set inner loop count
    mov ecx, 20

L2:
    ; loop body
    ...
    ; decrement inner loop counter
    dec ecx
    ; jump to beginning of inner loop if inner loop counter is not zero
    jne L2
    ; restore outer loop count
    mov ecx, count
    ; decrement outer loop counter
    dec ecx
    ; jump to beginning of outer loop if outer loop counter is not zero
    jne L1
```

The nested loop you provided is an example of a combination of an outer loop and an inner loop, a common construct in programming. Let me explain what this nested loop does:

### ***Outer Loop (L1):***

- The outer loop is controlled by the ecx register and is initialized with the value 100.
- Inside the outer loop:
  - The current value of ecx (the outer loop counter) is saved into the count variable.
- The inner loop is initialized by setting ecx to 20.

### ***Inner Loop (L2):***

- The inner loop is also controlled by the ecx register and is initialized with the value 20. Inside the inner loop:
  - There is a placeholder comment for the loop body, where actual operations or instructions would be performed.
  - dec ecx is used to decrement the inner loop counter.
  - jne L2 is a conditional jump instruction. It checks if the inner loop counter (ecx) is not equal to zero. If it's not zero, the program jumps back to the beginning of the inner loop (L2) to continue iterating.

### ***Outer Loop Continuation:***

- After the inner loop completes (when ecx becomes zero), the program restores the original value of ecx from the count variable.
- dec ecx is used to decrement the outer loop counter.
- jne L1 is another conditional jump instruction. It checks if the outer loop counter (ecx) is not equal to zero. If it's not zero, the program jumps back to the beginning of the outer loop (L1) to

continue iterating.

In summary, this nested loop structure is designed to execute the inner loop 20 times for each iteration of the outer loop, resulting in a total of  $100 * 20 = 2000$  iterations in total.

The actual operations or instructions within the loop body are not provided in the code snippet, but they would be executed repeatedly as part of the loop's functionality.

```
;NB: Don't indent assembly code like me, I just want clarity
.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.data
    intarray DWORD 10000h,20000h,30000h,40000h

.code
main PROC
    ; 1: EDI = address of intarray
    mov edi, OFFSET intarray
    ; 3: sum = 0
    mov eax, 0
    ; 2: initialize loop counter
    mov ecx, LENGTHOF intarray
L1:
    ; 5: add an integer
    add eax, [edi]
    ; 6: point to next element
    add edi, TYPE intarray
    ; 7: repeat until ECX = 0
    loop L1
    ; Exit with success code
    invoke ExitProcess, 0
main ENDP
END main
```

The code defines the intarray array, which contains four 16-bit integers.

This section of the code defines the main() function, which is the entry point for the program.

```
.code  
main PROC
```

Move the address of the intarray array into the EDI register. The EDI register will be used as an indexed operand to access the array elements.

Move the value 0 into the EAX register. The EAX register will be used to accumulate the sum of the array elements.

Move the length of the intarray array into the ECX register. The ECX register will be used as the loop counter.

L1: This label marks the beginning of the loop.

Add the value at the current address in the EDI register to the EAX register. The EDI register contains the address of the current array element.

Increment the EDI register by the size of an array element. This points the EDI register to the next array element.

Jump back to the L1 label if the ECX register is not equal to zero. This causes the loop to repeat until the ECX register is zero.

Invoke the ExitProcess function, which exits the program with a success code.

```
main ENDP  
END main
```

These directives mark the end of the `main()` function and the program.

## ***Copying Strings***

Using a loop that copies a string, represented as an array of bytes with a null terminator value.

Indexed addressing works well for this type of operation because the same index register references both strings.

The target string must have enough available space to receive the copied characters, including the null byte at the end:

```
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO ,dwExitCode:DWORD

.data
    source BYTE      "This is the source string",0
    target BYTE      SIZEOF source DUP(0)

.code
main PROC
    mov esi, 0          ;index register
    mov ecx, SIZEOF source ;loop counter

L1:
    mov al, source[esi]   ;get a character from source
    mov target[esi], al    ;store it in the target
    inc esi                ;move to the next character

invoke ExitProcess, 0
main ENDP
END main
```

When you move a 32-bit constant to a 64-bit register, the upper 32 bits (bits 32-63) of the destination are cleared (equal to zero):

destination are cleared (equal to zero):

```
mov rax,0FFFFFFFh ; rax = 00000000FFFFFF
```

When you move a 16-bit constant or an 8-bit constant into a 64-bit register, the upper bits are also cleared:

```
mov rax,06666h ; clears bits 16-63  
mov rax,055h ; clears bits 8-63
```

When you move memory operands into 64-bit registers, however, the results are mixed. For example, moving a 32-bit memory operand into EAX (the lower half of RAX) causes the upper 32 bits in RAX to be cleared:

```
.data  
    myDword DWORD 80000000h  
.code  
    mov rax, 0xFFFFFFFFFFFFFFFh  
    mov eax, myDword ;RAX = 0000000080000000
```

But when you move an 8-bit or a 16-bit memory operand into the lower bits of RAX, the highest bits in the destination register are not affected:

The following assembly code example shows how moving an 8-bit or 16-bit memory operand into the lower bits of RAX does not affect the highest bits in the destination register:

```
.data
    myByte BYTE 55h
    myWord WORD 6666h

.code
    ; Move the 16-bit value from myWord into the lower bits of RAX.
    mov ax, myWord

    ; Display the value of RAX in hexadecimal format.
    ; Output: 0066 (only the lower 16 bits are displayed)
    mov edx, 1 ; standard output
    mov eax, RAX
    syscall

    ; Move the 8-bit value from myByte into the lower bits of RAX.
    mov al, myByte

    ; Display the value of RAX in hexadecimal format.
    ; Output: 55 (only the lower 8 bits are displayed)
    mov edx, 1 ; standard output
    mov eax, RAX
    syscall
```

This code is performing the following operations:

**Data Section**, It defines two variables: myByte as a single-byte variable (BYTE) initialized with the value 55h (hexadecimal 55). myWord as a two-byte variable (WORD) initialized with the value 6666h (hexadecimal 6666). It begins the code section where the actual operations are performed.

**Moving myWord into RAX**: It moves the 16-bit value stored in myWord into the lower 16 bits of the RAX register using `mov ax, myWord`. This operation copies the value 6666h into the lower 16 bits of RAX.

**Displaying RAX in Hexadecimal Format**: It prepares for displaying the value of RAX in hexadecimal format. `mov edx, 1` sets up the standard output file descriptor for `syscall` (assuming a Unix-like environment).

`mov eax, RAX` copies the value from RAX to EAX, ensuring that the upper 32 bits are zeros. `syscall` is used to display the value in EAX, which contains 6666h. The output is "0066" in hexadecimal format because only the lower 16 bits are displayed.

**Moving myByte into RAX**: It moves the 8-bit value stored in myByte into the lower 8 bits of the RAX register using `mov al, myByte`. This operation copies the value 55h into the lower 8 bits of RAX.

**Displaying RAX in Hexadecimal Format Again**: It repeats the preparation for displaying the value of RAX in hexadecimal format. `mov eax, RAX` copies the value from RAX to EAX, ensuring that the upper 32 bits are zeros.

`syscall` is used to display the value in EAX, which contains 55h. The output is "55" in hexadecimal format because only the lower 8 bits are displayed.

In summary, this code demonstrates how to move 16-bit and 8-bit values from memory into the RAX register and then display the lower bits of RAX in hexadecimal format using system calls.

The **MOVSXD instruction (move with sign-extension)** permits the source operand to be a 32-bit register or memory operand. The following instructions cause RAX to equal FFFFFFFFFFFFFFh:

```
mov ebx,0FFFFFFFh  
movsxd rax,ebx
```

The code is moving the 16-bit value from the variable myWord into the lower 16 bits of the RAX register. RAX is a 64-bit register, so the highest bits of the register are not affected.

In other words, the code is copying the lower 16 bits of myWord into RAX, and the upper 48 bits of RAX remain unchanged.

This is done by using the MOV instruction in assembly language. The MOV instruction copies the specified number of bits from the source operand to the destination operand.

In this case, the source operand is myWord, which is a 16-bit variable, and the destination operand is RAX, which is a 64-bit register. Therefore, only the lower 16 bits of myWord are copied into RAX.

This operation is often used to prepare data for further processing. For example, the lower 16 bits of a 32-bit value may be copied into RAX so that it can be used in a mathematical operation. Or, the lower 16 bits of a 64-bit value may be copied into RAX so that it can be stored in a 32-bit register.

---

The MOVSXD instruction in assembly language is used to move a signed doubleword (32 bits) to a signed quadword (64 bits) with sign extension. This means that if the sign bit of the source operand

is set, then the upper 32 bits of the destination operand are also set.

In the image you sent, the MOVSXD instruction is being used to move the 32-bit value from the EBX register to the RAX register. RAX is a 64-bit register, so the upper 32 bits of the register will be set to the sign bit of EBX.

This operation is often used to prepare data for further processing. For example, a 32-bit value may be sign extended to a 64-bit value so that it can be used in a mathematical operation that expects a 64-bit operand. Or, a 32-bit value may be sign extended to a 64-bit value so that it can be stored in a 64-bit register.

In the specific example of the code you sent, the purpose of the operation is not clear. It is possible that the code is preparing the value of EBX for further processing, or it is possible that the code is simply copying the value of EBX to another location in memory.

Here is an example of how the MOVSXD instruction can be used to prepare data for further processing:

; EBX contains a 32-bit signed value.

; RAX is a 64-bit register.

movsx rax, ebx ; Sign extend the 32-bit value in EBX to 64 bits.

; RAX now contains a 64-bit signed value.

This code can then be used to perform a mathematical operation on the value in RAX, such as:

```
; Add 10 to the value in RAX.  
add rax, 10
```

The **ADD instruction** expects a 64-bit operand, so the sign extended value in RAX is used in the operation.

The **MOVSXD instruction** is a powerful tool that can be used to prepare data for further processing in a variety of ways.

---

The OFFSET operator generates a 64-bit address, which must be held by a 64-bit register or variable. In the following example, we use the RSI register:

```
.data  
    myArray WORD 10,20,30,40  
.code  
    mov rsi,OFFSET myArray
```

In the data section, you define an array named `myArray` of type WORD (16-bit words) and initialize it with four values: 10, 20, 30, and 40. Code Section:

The code section contains the actual assembly language instructions. Using the OFFSET Operator:

The line `mov rsi, OFFSET myArray` is where the action happens. `OFFSET myArray` is an operator that calculates the memory address (64-bit in this case) of the `myArray` variable.

`mov rsi, OFFSET myArray` moves this calculated memory address into the 64-bit RSI register.

After this instruction, the RSI register holds the memory address where the `myArray` variable is located in memory.

In summary, this code calculates the memory address of the `myArray` variable using the `OFFSET` operator and stores it in the 64-bit RSI register.

This can be useful when you want to work with the memory address of a variable or an array in a 64-bit environment, allowing you to manipulate data at that address or access elements of the array.

---

Certainly, let's break down the key concepts in the provided text:

### ***64-Bit Mode and Registers:***

In 64-bit mode, the RCX register is commonly used as the loop counter with the `LOOP` instruction.

Programming in 64-bit mode is often more straightforward when using 64-bit integer variables and registers.

ASCII strings consist of bytes, making indirect or indexed addressing common when working with them.

### ***64-Bit Version of SumArray:***

```
; Summing an Array (SumArray_64.asm)
ExitProcess PROTO

.data
intarray QWORD 1000000000000h,2000000000000h,3000000000000h,4000000000000h

.code
main PROC
    mov rdi,OFFSET intarray    ; RDI = address of intarray
    mov rcx,LENGTHOF intarray ; initialize loop counter
    mov rax,0                  ; sum = 0

L1: ; mark beginning of loop
    add rax,[rdi]              ; add a quadword
    add rdi,TYPE intarray      ; point to next element
    loop L1                   ; repeat until RCX = 0

    mov ecx,0                  ; ExitProcess return value
    call ExitProcess
main ENDP
END main
```

The code is an example of a program called "SumArray" that calculates the sum of an array of 64-bit integers.

It uses the QWORD directive to define an array of quadwords (64-bit integers).

Registers are updated to use 64-bit counterparts, like RDI, RCX, and RAX.

The program iterates through the array, adding each 64-bit integer to the RAX register.

The loop continues until the RCX loop counter reaches zero. The program then sets ECX to zero (for ExitProcess) and calls the ExitProcess function to exit.

### ***Addition and Subtraction in 64-Bit Mode:***

```
; Addition and Subtraction in 64-Bit Mode (AddSubtract_64.asm)

.data

.code
main PROC
    ; Adding 1 to a 32-bit number in RAX
    mov rax,0FFFFFFFh      ; fill the lower 32 bits
    add rax,1                ; RAX = 10000000h

    ; Adding 16-bit values in AX and BX
    mov rax,0FFFFh          ; RAX = 000000000000FFFF
    mov bx,1                  ; BX = 0001
    add ax,bx                ; RAX = 0000000000000000

    ; Subtracting 1 from zero in EAX
    mov rax,0                ; RAX = 0000000000000000
    mov ebx,1                  ; EBX = 00000001
    sub eax,ebx              ; RAX = 00000000FFFFFF

    ; Subtracting 1 from zero in AX
    mov rax,0                ; RAX = 0000000000000000
    mov bx,1                  ; BX = 0001
    sub ax,bx                ; RAX = 000000000000FFFF

    ; Exit the program
    mov ecx,0                  ; ExitProcess return value
    call ExitProcess
main ENDP
END main
```

The ADD, SUB, INC, and DEC instructions in 64-bit mode affect CPU status flags similarly to 32-bit mode.

In an example, adding 1 to a 32-bit number in RAX causes bit 32 to receive a 1, resulting in RAX becoming 100000000h.

When using partial register operands, be aware that the remainder of the register is not modified. For example, adding 16-bit values in AX and BX does not affect the upper bits in RAX.

The same principle applies to subtraction. Subtracting 1 from zero in EAX or AX modifies the lower bits of RAX while leaving the upper bits unchanged.

These concepts outline the behavior of instructions and registers in 64-bit mode, particularly with regard to handling integers and loops.

### ***Use of 64-bit Registers with Indirect Operands:***

In 64-bit mode, when working with indirect operands, you must use a 64-bit general-purpose register to hold the memory address. Additionally, it's essential to use the PTR operator to specify the size of the target operand. Here are some examples:

dec BYTE PTR [rdi]	; Decrement an 8-bit target
inc WORD PTR [rbx]	; Increment a 16-bit target
inc QWORD PTR [rsi]	; Increment a 64-bit target

dec BYTE PTR [rdi]:

- **dec**: This instruction decrements (subtracts 1 from) the operand.
- **BYTE PTR**: It specifies that the target operand is a byte (8 bits) in size.
- **[rdi]**: This indicates that the memory location pointed to by the 64-bit register rdi will be decremented.
- **Explanation**: This instruction decreases the value at the memory location pointed to by rdi by 1 byte. It's useful for operations involving 8-bit data, like decrementing a byte in memory.

#### **inc WORD PTR [rbx]:**

- **inc**: This instruction increments (adds 1 to) the operand.
- **WORD PTR**: It specifies that the target operand is a word (16 bits) in size.
- **[rbx]**: This indicates that the memory location pointed to by the 64-bit register rbx will be incremented.
- **Explanation**: This instruction increases the value at the memory location pointed to by rbx by 1 word (16 bits). It's used for operations involving 16-bit data, like incrementing a 16-bit value in memory.

#### **inc QWORD PTR [rsi]:**

- **inc**: This instruction increments (adds 1 to) the operand.
- **QWORD PTR**: It specifies that the target operand is a quadword (64 bits) in size.
- **[rsi]**: This indicates that the memory location pointed to by the 64-bit register rsi will be incremented.
- **Explanation**: This instruction increases the value at the memory location pointed to by rsi by 1 quadword (64 bits). It's suitable for operations involving 64-bit data, such as incrementing a 64-bit integer in memory.

In summary, these instructions allow you to manipulate data of different sizes in memory, whether

it's an 8-bit, 16-bit, or 64-bit target. The specific size is indicated by the BYTE PTR, WORD PTR, or QWORD PTR specifiers, and the operation (increment or decrement) is performed accordingly on the target operand at the specified memory location.

### ***Scale Factors in Indexed Operands:***

In 64-bit mode, you can also use scale factors in indexed operands, similar to 32-bit mode. If you're working with an array of 64-bit integers, you can use a scale factor of 8 to correctly calculate offsets. Here's an example:

```
.data  
array QWORD 1,2,3,4  
  
.code  
    mov esi,3          ; Subscript  
    mov eax, array[rsi*8] ; EAX = 4
```

In this example, we're accessing a 64-bit integer in the array using a scale factor of 8 to account for the size of each element.

### ***64-Bit Pointers:***

In 64-bit mode, pointer variables hold 64-bit offsets. Here's an example where the variable ptrB holds the offset of the arrayB:

```
.data  
    arrayB BYTE 10h,20h,30h,40h  
    ptrB QWORD arrayB
```

Optionally, you can declare ptrB with the OFFSET operator to make the relationship between the pointer and the array clearer:

```
.data  
    arrayB BYTE 10h,20h,30h,40h  
    ptrB QWORD OFFSET arrayB
```

---

In this chapter, we've covered various essential concepts related to data transfers, addressing, arithmetic, and loops in assembly language programming. Here's a concise summary of the key points:

#### **MOV Instruction:**

- The MOV instruction is used for transferring data from a source operand to a destination operand.

#### **Zero and Sign Extension:**

- MOVZX extends a smaller operand to a larger one by zero-filling.
- MOVSX extends a smaller operand to a larger one by sign extension.

#### **XCHG Instruction:**

- XCHG swaps the contents of two operands, with at least one of them being a register.

## **Operand Types:**

- Direct operand represents the address of a variable.
- Direct-offset operand adds a displacement to a variable's name, generating a new offset.
- Indirect operand uses a register containing the address of data.
- Indexed operand combines a constant with an indirect operand for effective memory access.

## **Arithmetic Instructions:**

- INC adds 1 to an operand.
- DEC subtracts 1 from an operand.
- ADD adds a source operand to a destination operand.
- SUB subtracts a source operand from a destination operand.
- NEG reverses the sign of an operand.

## **Status Flags:**

- Various CPU status flags are affected by arithmetic operations, including Sign, Carry, Parity, Auxiliary Carry, Zero, and Overflow flags.

## **Operators:**

- OFFSET operator returns the distance of a variable from the beginning of its segment.
- PTR operator overrides a variable's declared size.
- TYPE operator returns the size of a variable or array element.
- LENGTHOF operator returns the number of elements in an array.
- SIZEOF operator returns the number of bytes used by an array initializer.
- TYPEDEF operator creates user-defined types.

## **Loops:**

- JMP instruction unconditionally branches to another location.

- LOOP instruction is used in counting-type loops, with ECX as the counter in 32-bit mode and RCX in 64-bit mode.

#### **64-Bit Mode:**

- In 64-bit mode, a 64-bit general-purpose register must be used when working with indirect operands.
- Scale factors can be used in indexed operands, useful for arrays of 64-bit integers.

=====

#### **QUESTIONS:**

=====

**Question 9: What value will RAX contain after the following instruction executes? `mov rax,44445555h`**

Answer: RAX will contain the hexadecimal value 44445555h.

**Question 10: What value will RAX contain after the following instructions execute?**

```
.data
    dwordVal DWORD 84326732h
.code
    mov rax,0xFFFFFFFF00000000h
    mov rax,dwordVal
```

Answer: RAX will contain the value FFFFFFFF84326732h.

Question 11: What value will EAX contain after the following instructions execute?

```
.data  
    dVal DWORD 12345678h  
.code  
    mov ax,3  
    mov WORD PTR dVal+2,ax  
    mov eax,dVal
```

Answer: EAX will contain the value 00030000h.

Question 12: What will EAX contain after the following instructions execute?

```
.data  
    dVal DWORD ?  
.code  
    mov dVal,12345678h  
    mov ax,WORD PTR dVal+2  
    add ax,3  
    mov WORD PTR dVal,ax  
    mov eax,dVal
```

Answer: EAX will contain the value 00030000h.

**Question 12: Is it possible to set the Overflow flag if you add a positive integer to a negative integer?**

Answer: Yes, it is possible to set the Overflow flag when adding a positive integer to a negative integer if the result is out of range for the destination operand.

**Question 13: Will the Overflow flag be set if you add a negative integer to a negative integer and produce a positive result?**

Answer: No, the Overflow flag will not be set when adding two negative integers that result in a positive value.

**Question 14: Is it possible for the NEG instruction to set the Overflow flag?**

Answer: Yes, the NEG instruction can set the Overflow flag if the operation results in an overflow.

**Question 15: Is it possible for both the Sign and Zero flags to be set at the same time?**

Answer: No, it is not possible for both the Sign and Zero flags to be set simultaneously. If the result is zero, the Zero flag is set, indicating that the sign is not negative, so the Sign flag is not set. If the result is non-zero, the Sign flag is set to indicate whether it's positive or negative.

**Exercise 1: Converting from Big Endian to Little Endian**

```
.data
bigEndian BYTE 12h,34h,56h,78h
littleEndian DWORD 0

.code
mov eax, 0
mov al, [bigEndian]
mov ah, [bigEndian + 1]
mov [littleEndian], ax

mov eax, 0
mov al, [bigEndian + 2]
mov ah, [bigEndian + 3]
mov [littleEndian + 2], ax
```

Indented:

```
.data
    bigEndian BYTE 12h,34h,56h,78h
    littleEndian DWORD 0

.code
    mov eax, 0
    mov al, [bigEndian]
    mov ah, [bigEndian + 1]
    mov [littleEndian], ax

    mov eax, 0
    mov al, [bigEndian + 2]
    mov ah, [bigEndian + 3]
    mov [littleEndian + 2], ax
```

This program copies the value from bigEndian to littleEndian while reversing the order of the bytes.

#### Exercise 2: Exchanging Pairs of Array Values

```
.data
    array DWORD 1, 2, 3, 4, 5, 6, 7, 8
    arraySize DWORD 8

.code
    mov esi, 0
    mov ecx, [arraySize]

exchangeLoop:
    mov eax, [array + esi]          ; Load the current element
    add esi, 4                      ; Move to the next element
    xchg eax, [array + esi]         ; Exchange the values
    loop exchangeLoop               ; Repeat for all pairs
```

This program exchanges every pair of values in the array with an even number of elements.

### Exercise 3: Summing the Gaps between Array Values

```
.data
    array DWORD 0, 2, 5, 9, 10
    arraySize DWORD 5
    sumOfGaps DWORD 0

.code
    mov esi, 0
    mov ecx, [arraySize] - 1      ; One less than the array size

    sumLoop:
        mov eax, [array + esi]    ; Load the current element
        add esi, 4                ; Move to the next element
        sub eax, [array + esi]    ; Calculate the gap
        add [sumOfGaps], eax      ; Add gap to the sum
        loop sumLoop              ; Repeat for all gaps
```

This program calculates the sum of all the gaps between successive array elements in the array variable.

You can assemble and run these programs using an x86 assembler like NASM or MASM in either 32-bit or 64-bit mode as per your preference.

## ***PROCEDURES***

Skip...

To link to an external library, you need to use the linker command line options -l and -L. The -l option tells the linker to link to the specified library, and the -L option tells the linker where to find the library file.

For example, to link to the Irvine32 library, you would use the following command:

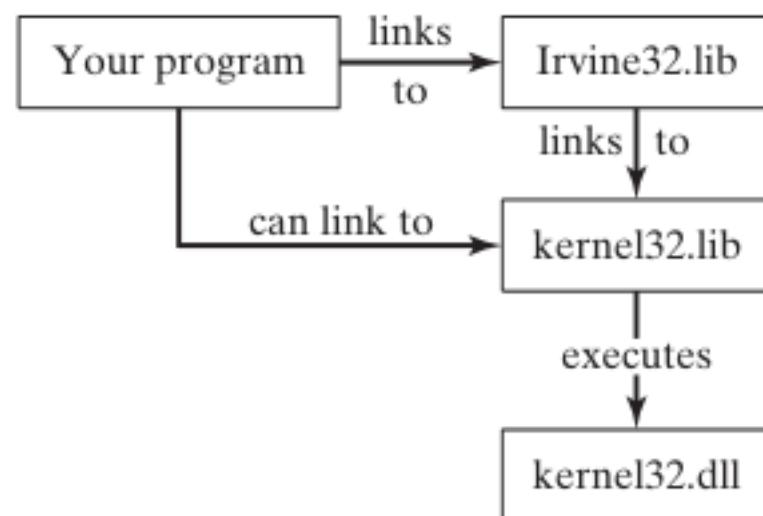
```
link hello.obj -l Irvine32.lib kernel32.lib
```

This will link the object file hello.obj to the Irvine32 library and the kernel32 library.

The kernel32 library is a system library that contains functions for performing basic operations such as reading and writing to the console.

Here is an example of a simple assembly language program that calls a procedure from the Irvine32 library:

FIGURE 5–9 Linking 32-bit programs.



```
; This program displays the string "Hello, world!" on the console.

include Irvine32.inc

.data
msg db "Hello, world!", 0

.code
start:
    mov eax, OFFSET msg
    call WriteString
    mov eax, 1 ; exit program
    int 0x80
```

Yes, the flowchart you sent is a good representation of the linking process for 32-bit programs. It shows how the linker combines the program's object file with the Irvine32.lib and kernel32.lib libraries to produce an executable file.

Here is a brief explanation of each step in the flowchart:

The linker reads the program's object file and creates a symbol table. This table contains the names and addresses of all the symbols in the program.

The linker then searches the Irvine32.lib and kernel32.lib libraries for the symbols that are referenced in the program's object file. If the linker finds a symbol in a library, it copies the machine code for that symbol into the program's executable file.

The linker then updates the program's symbol table to reflect the addresses of the copied symbols. The linker repeats steps 3 and 4 until all of the symbols in the program's object file have been resolved.

Once all of the symbols have been resolved, the linker writes the program's executable file to disk. Note that the flowchart does not show all of the details of the linking process.

For example, the linker also performs tasks such as resolving external references and generating relocation information.

Overall, the flowchart is a good overview of the basic steps involved in linking a 32-bit program.

# **Stack**

The **runtime stack** is a **last-in-first-out (LIFO)** data structure that is used to store function arguments, return values, and local variables.

A **push operation** decrements the **stack pointer (ESP)** by 4 bytes and copies a 32-bit value into the memory location pointed to by ESP.

A **pop operation** removes a 32-bit value from the memory location pointed to by ESP and increments ESP by 4 bytes.

=====

## ***Push Operation:***

=====

The following assembly code shows how to push and pop a 32-bit value onto and off of the stack:

---

; Push the value 10 onto the stack.

push 10

; Pop the value off of the stack and store it in the register EAX.

pop eax

Yes, the stack grows downwards in memory. This means that when you push a value onto the stack, it is stored at a lower memory address than the previous value on the stack.

This is different from the way that a pile of books grows. When you put a book on top of a pile of books, the pile grows upwards.

-----

When we say that a new item pushed onto the stack has a lower memory address, are we implying that the stack starts from the very high memory regions and moves to the very low memory regions?

For instance, if I have 4GB of RAM and a stack of 4096 bytes, does the stack start at the highest memory address, like 4096, and when I push a value like 10 onto the stack, the stack pointer points to 4092, and if I push 11 onto the stack, it goes to 4088?

Then, if I were to pop 10 from the stack, does the stack pointer move back to 4092? Is this the conventional behavior of a stack in memory allocation?

Yes, you're on the right track in understanding how the stack works in memory allocation. When we say that a new item pushed onto the stack has a lower memory address, it implies that the stack

grows from high memory addresses to low memory addresses.

Let me break down your example with your 4GB RAM and a stack size of 4096 bytes to clarify:

1. Initially, the stack pointer points to the top of the stack, which is the highest memory address, let's say 4096 (assuming the stack starts at the highest memory address).
2. You push the value 10 onto the stack. The stack pointer moves down by 4 bytes(for the esp is 32 bits) - 4092 is the new ESP location.
3. Then, you push the value 11 onto the stack. The stack pointer(ESP), moves down further to 4096 - 4 (4088 in your example).
4. If you were to pop the value 10 from the stack, the stack pointer would move back up to 4GB - 5 (4092), effectively "removing" the top element.

So, in summary, as you push items onto the stack, the stack pointer moves to lower memory addresses. When you pop items from the stack, it moves back up.

This is a fundamental concept in stack data structures, and it ensures that the most recently added item is always at the top of the stack.

---

FIGURE 5–3 Pushing integers on the stack.

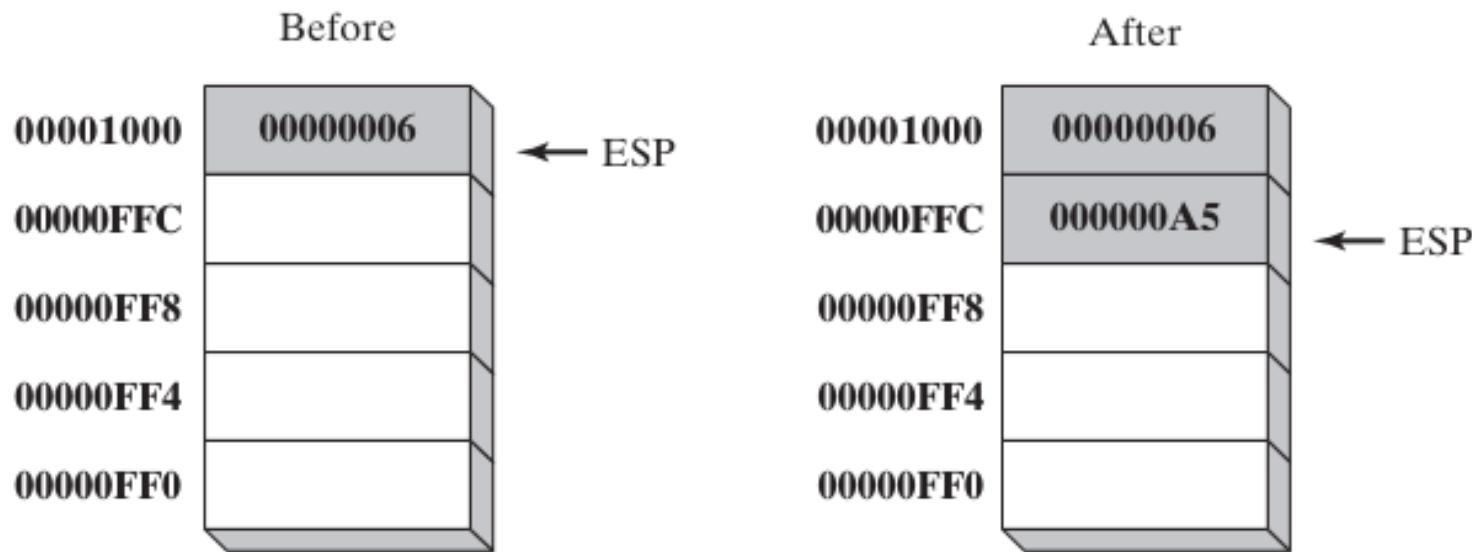
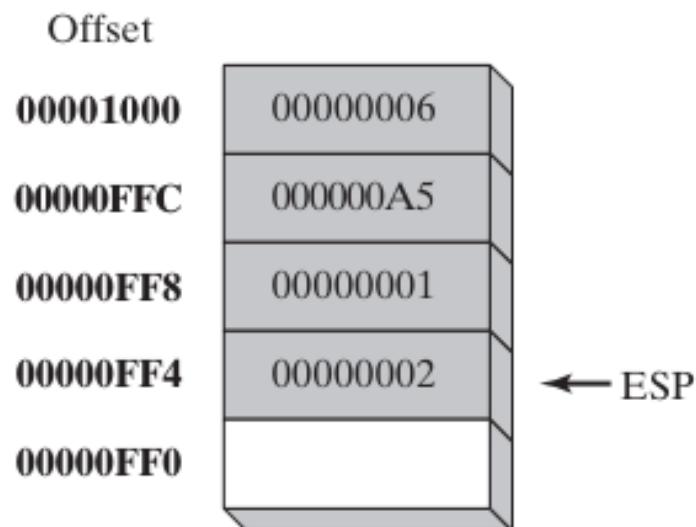


FIGURE 5–4 Stack, after pushing 00000001 and 00000002.



The push operation in a 32-bit system involves two main steps:

- 1. Decrementing the Stack Pointer:** The stack pointer (ESP) is decremented by 4 bytes.
- 2. Copying the Value:** The value you want to push is copied into the memory location pointed to by the updated stack pointer.

This effectively adds the value to the top of the stack.

The stack pointer always points to the last item pushed onto the stack.

The runtime stack typically grows downward in memory, from higher memory addresses to lower memory addresses.

This means that as you push items onto the stack, the stack pointer moves to lower memory addresses.

=====

### ***Pop Operation:***

=====

The pop operation **removes a value from the stack and increments the stack pointer**.

The pop operation in stack management is crucial for removing a value from the stack and restoring the stack pointer to its appropriate location. Here's a breakdown of how the pop operation works:

**Removing a Value:** When you perform a pop operation, the value at the memory location pointed to by

the stack pointer is removed from the stack. This effectively takes the top item off the stack.

**Incrementing the Stack Pointer:** After the value is popped from the stack, the stack pointer (ESP) is incremented by the stack element size. This size is typically 4 bytes in a 32-bit system because it corresponds to the size of a 32-bit integer.

Let's use the example you provided:

Suppose you have a stack with the value 00000002 at the top, and you want to perform a pop operation to remove it.

Before the pop: ESP points to the memory location where 00000002 is stored. After the pop:

The value 00000002 is removed from the stack. ESP is incremented by 4 bytes (the size of a 32-bit integer).

ESP now points to the next-highest location in the stack.

Regarding the use of PUSHFD and POPFD to save and restore flags in assembly language, it's a useful technique.

Flags are often important for controlling program flow and making decisions based on certain conditions.

By pushing the flags onto the stack and later popping them, you can effectively save and restore the flags' state.

However, it's essential to be cautious when using such instructions to ensure that the program's execution path does not skip over the **POPFD instruction**.

**Documentation** and careful programming are critical in this regard.

An alternative method, as you mentioned, is to **push the flags onto the stack and immediately pop them into a variable**.

This provides a more structured and **less error-prone way** to save and restore the flags, as you can clearly see where the flags are being stored and retrieved.

In summary, the pop operation in stack management is used to remove values from the stack and increment the stack pointer accordingly.

Techniques like using PUSHFD and POPFD or pushing flags onto the stack and immediately popping them into variables can be valuable for managing program flags in assembly language.

---

### ***PUSHFD and POPFD Operation:***

---

```
.data
savedFlags DWORD ?

.code
main PROC
    ; Save the flags using PUSHFD
    pushfd

    ; Store the saved flags into the 'savedFlags' variable
    pop savedFlags

    ; Modify some flags (for demonstration purposes)
    cli ; Clear Interrupt Flag
    sti ; Set Interrupt Flag

    ; Restore the original flags using 'savedFlags'
    push savedFlags
    popfd

    ; Your code continues here

    ; Exit the program
    mov eax, 0x4C00 ; Exit code 0
    int 0x21        ; Call DOS function to exit
main ENDP

END main
```

The **PUSHFD instruction** in x86 assembly language knows where to find the flags in the flags register. You don't need to specify anything explicitly when using PUSHFD.

The **PUSHFD instruction** is designed to push the entire contents of the flags register (EFLAGS) onto the stack, and it does so automatically.

It saves all the status flags, such as the Zero Flag (ZF), Carry Flag (CF), Overflow Flag (OF), etc., as well as other control flags like the Direction Flag (DF) and Interrupt Flag (IF).

When you execute PUSHFD, it takes care of pushing the 32-bit value representing the flags onto the stack, and you don't need to specify individual flags.

Later, when you use POPFD, it restores the flags register from the value on the stack. So, you can use PUSHFD and POPFD without specifying the flags explicitly; they handle the flags manipulation for you.

---

In this example:

We declare a DWORD variable named savedFlags in the .data section to store the saved flags.

Inside the main procedure, we first use PUSHFD to push the flags onto the stack.

Then, we immediately POP the saved flags from the stack and store them in the savedFlags variable.

Next, we modify some flags for demonstration purposes. We clear the Interrupt Flag (CLI) and then set it (STI). These changes will be temporary.

To restore the original flags, we push the savedFlags variable onto the stack and then use POPFD to copy the saved flags back into the flags register.

Your code can continue after this point.

Finally, we use the DOS interrupt int 0x21 to exit the program. The value 0x4C00 in eax specifies that the program should exit with code 0.

Please note that using PUSHFD and POPFD to manipulate flags should be done carefully, as it affects the control flow and behavior of your program. Ensure that you save and restore the flags correctly to avoid unintended consequences.

```
; Reversing a String (RevStr.asm)
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD

.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1

.code
main PROC
    ; Initialize registers and counters
    mov ecx, nameSize
    mov esi, 0
```

L1:

```
; Get the next character from the 'aName' string  
movzx eax,aName[esi]  
  
; Push the character onto the stack  
push eax  
  
; Move to the next character  
inc esi  
  
; Continue the loop until all characters are processed  
loop L1  
  
; Pop the characters from the stack (in reverse order)  
; and store them back into the 'aName' array  
  
; Reset counters  
mov ecx,nameSize  
mov esi,0
```

L2:

```
; Get the character from the stack
pop eax

; Store the character in the 'aName' string
mov aName[esi],al

; Move to the next character
inc esi

; Continue the loop until all characters are restored
loop L2

; Exit the program
INVOKE ExitProcess,0

main ENDP
END main
```

Now, let's break down what this code is doing:

The code begins with some assembly directives and the declaration of the ExitProcess function, which is used to exit the program later.

In the .data section, there's a string aName containing "Abraham Lincoln," and nameSize is

calculated to determine the length of the string.

In the .code section, the main procedure starts. It initializes some registers and counters for later use.

The first loop labeled as L1 processes each character in the aName string:

It retrieves a character from the string.

Pushes that character onto the stack.

Moves to the next character in the string.

Repeats until all characters are processed.

After pushing all characters onto the stack, the code proceeds to the second loop labeled as L2, which restores the characters in reverse order:

It pops a character from the stack.

Stores that character back into the aName string. Moves to the next character. Continues until all characters are restored. Finally, the program invokes the ExitProcess function to exit.

This program effectively reverses the characters in the aName string by pushing them onto the stack and then popping them off in reverse order. The stack's Last-In-First-Out (LIFO) behavior ensures that the characters are reversed in the string.

-----

Here's what's happening step by step:

### **.data Section:**

In the .data section, you've defined a null-terminated string named aName with the value "Abraham Lincoln" and terminated with a null character 0. This string represents the name you want to reverse. You've also calculated the length of the string using nameSize.

The expression `($ - aName)` calculates the difference between the current location `$`, which represents the address of the current instruction) and the start of the aName string. This effectively gives you the length of the string. However, you subtract 1 to account for the null terminator, as it's not part of the string length.

### **.code Section:**

In the .code section, the main procedure begins. `mov ecx, nameSize:` This instruction moves the value of nameSize (which represents the length of the string) into the ecx register. The ecx register is often used as a loop counter or for holding counts in assembly language. `mov esi, 0:` This instruction initializes the esi register to 0. The esi register is often used as a pointer or an index when working with strings or arrays. So, in summary, this part of the code sets up the ecx register with the length of the aName string and initializes the esi register to 0, which is used to iterate through the string character by character in subsequent parts of the code.

### **L1:Reversing the String and Pushing onto the Stack**

In this part of the code labeled as L1, the goal is to reverse the characters in the aName string and push them onto the stack in reverse order:

`movzx eax, aName[esi]:` This instruction loads the next character from the aName string into the eax register. `movzx` is used to zero-extend the character from a byte to a 32-bit register. `aName[esi]`

accesses the character at the current position pointed to by esi.

**push eax:** After loading the character into eax, it is pushed onto the stack using the push instruction. This effectively reverses the order of characters since the stack is a Last-In-First-Out (LIFO) data structure.

**inc esi:** This increments the esi register, which acts as an index, to move to the next character in the string.

**loop L1:** The loop instruction decrements the ecx register (loop counter) and checks if it's zero. If ecx is not zero, it jumps back to L1 to process the next character. This loop continues until all characters are pushed onto the stack.

## ***L2: Popping Characters from the Stack and Restoring the String***

In this part of the code labeled as L2, the reversed characters are popped from the stack and stored back into the aName string:

- **pop eax:** The pop instruction retrieves a character from the stack (in reverse order) and stores it in the eax register.
- **mov aName[esi], al:** This instruction takes the character in al (the lower byte of eax) and stores it back into the aName string at the current position pointed to by esi.
- **inc esi:** esi is incremented to move to the next character in the string.
- **loop L2:** The loop continues until all characters have been restored, just like in L1.

Finally, the program exits using INVOKE ExitProcess, 0.

Overall, this code snippet efficiently reverses the characters in the aName string by pushing them onto the stack in L1 and then popping them off in reverse order in L2, effectively reversing the string in place.

---

Certainly, let's focus on the more complex parts of the code and break down the challenging aspects:

**String Processing:** The code is manipulating a string (aName) character by character. This involves understanding how strings are represented in memory and how to access individual characters using indexing (e.g., aName[esi]). The movzx instruction is used to zero-extend the character from a byte to a 32-bit register (eax) for proper handling.

**Looping with ECX:** The code uses the ecx register as a counter to control loops. Understanding how loop counters work and how they decrement with each iteration is essential. The loop instruction is used to check the counter and decide whether to continue looping.

**Stack Manipulation:** Pushing and popping values onto/from the stack can be conceptually challenging. It's important to grasp that the stack is a last-in-first-out (LIFO) data structure. push pushes a value onto the stack, and pop retrieves it. In this code, characters are pushed onto the stack in one loop and popped from it in another loop to reverse their order.

**Register Usage:** The code uses various registers (eax, esi, ecx, al) for different purposes. Keeping track of what each register is used for and how they interact with the data is crucial.

**Procedure Calls:** The program uses the INVOKE macro to call the ExitProcess function to exit. Understanding how to make function calls and pass parameters is fundamental in assembly programming.

**Memory Access and Data Movement:** Manipulating memory and moving data between registers and memory locations requires a solid understanding of assembly concepts. This includes addressing modes, data sizes (byte, word, dword), and operations like `mov` (move) and `movzx` (move with zero extension).

**String Length Calculation:** The calculation of `nameSize` may be challenging for beginners. It determines the length of the string (`aName`) by subtracting the address of the string's beginning from the current location (`$`). This can be a bit abstract and requires knowledge of how memory addresses work.

**Control Flow:** Understanding how the program flows from one section to another, particularly the branching between loops (L1 and L2), is essential.

These are some of the more challenging aspects of the code. Assembly language programming can be intricate because it requires precise management of low-level details. However, with practice and a good grasp of the fundamentals, it becomes easier to work with.

---

Let's review the questions one by one:

**Which register (in 32-bit mode) manages the stack?**

In 32-bit mode, the `ESP` (Extended Stack Pointer) register manages the stack. It points to the top of the stack, and as items are pushed or popped, it is adjusted accordingly.

**How is the runtime stack different from the stack abstract data type?**

The runtime stack, as used in computer programming, is a memory region used for function calls, local variable storage, and managing program flow. It grows and shrinks as functions are called and return. The stack abstract data type, on the other hand, is a concept in computer science that represents a data structure with a specific set of operations (push and pop) and follows the Last-In-First-Out (LIFO) principle. While the runtime stack is implemented using memory in the computer's

architecture, the stack abstract data type is a higher-level concept used in algorithms and data structures.

### **Why is the stack called a LIFO structure?**

The stack is called a LIFO (Last-In-First-Out) structure because the last item (or element) pushed onto the stack is the first one to be popped off. It operates on the principle that the most recently added item is the one that gets accessed and removed first, similar to a stack of physical objects where you would take the top item first.

### **When a 32-bit value is pushed on the stack, what happens to ESP?**

When a 32-bit value is pushed onto the stack, the ESP (Extended Stack Pointer) register is decremented by 4 bytes (the size of a 32-bit value). This adjustment of ESP ensures that it points to the new top of the stack.

### **(True/False): Local variables in procedures are created on the stack.**

True. In many programming languages and on most computer architectures, local variables within functions or procedures are typically allocated on the stack. This allows for efficient memory management and scoping within function calls.

### **(True/False): The PUSH instruction cannot have an immediate operand.**

False. The PUSH instruction can indeed have an immediate operand. You can push an immediate value onto the stack using PUSH in assembly language. For example, PUSH 42 would push the value 42 onto the stack.

-----

Let's go through the assembly language instructions and directives you've mentioned that haven't been discussed yet:

1. **ENDP:** This directive marks the end of a procedure. It is used to define the end of a named procedure block. Typically, you see it paired with PROC to define the beginning and end of a procedure.
2. **POP:** The POP instruction is used to remove (pop) the top value from the stack and store it in the specified destination operand. For example, POP RAX would pop the top value from the stack and store it in the RAX register.
3. **POPA and POPAD:** These instructions are used to pop all general-purpose registers in a specific order (from DI to AX for POPA and from EDI to EAX for POPAD) from the stack.
4. **POPFD:** This instruction pops the values of the EFLAGS register (the processor flags) from the stack. It is used to restore the state of the flags.
5. **PROC:** The PROC directive is used to define the beginning of a named procedure block. It's often paired with ENDP to enclose the code of a subroutine or function.
6. **PUSH:** The PUSH instruction is used to push a value onto the stack. For example, PUSH RAX pushes the value in the RAX register onto the stack.
7. **PUSHA and PUSHAD:** These instructions are used to push all general-purpose registers in a specific order (from AX to DI for PUSHA and from EAX to EDI for PUSHAD) onto the stack.
8. **PUSHFD:** This instruction pushes the values of the EFLAGS register (the processor flags) onto the stack. It is used to save the current state of the flags.
9. **RET:** The RET instruction is used to return from a procedure or subroutine. It typically pops the return address from the stack and transfers control back to the calling code.

**10. USES:** The USES directive is used to specify which registers a procedure uses or modifies. It helps document which registers are affected by the procedure, making it clear to the programmer.

---

**USES:** The USES directive is used in the context of procedure or subroutine declarations. It specifies which registers a procedure uses or modifies. For example, if a procedure uses the RAX and RBX registers, you can declare it like this:

```
MyProcedure PROC USES RAX, RBX
```

This helps document which registers the procedure interacts with, making it clear to other programmers.

**POPA** and **POPAD**: These instructions are used to pop all general-purpose registers in a specific order from the stack. POPA is for 16-bit registers (DI through AX), while POPAD is for 32-bit registers (EDI through EAX). They are the opposite of **PUSHA** and **PUSHAD**, which push all these registers onto the stack.

**PUSHFD** and **PUSHAD**: These instructions are used to save the state of processor flags and general-purpose registers on the stack.

PUSHFD pushes the EFLAGS register (processor flags) onto the stack. It's used to save the current state of flags. PUSHAD pushes all general-purpose registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI) onto the stack. It's used to save the values of these registers. POPFD: This instruction is used to pop the values of the EFLAGS register (processor flags) from the stack. It's used to restore the state of flags.



```
477 .model flat, stdcall
478 .stack 4096
479
480 .data
481     message db "Hello, World!", 0
482
483 .code
484     MyProcedure PROC USES EAX, EBX
485         ; Your code here that uses EAX and EBX registers
486         mov eax, 42
487         mov ebx, 24
488         add eax, ebx
489         ; Rest of your code
490         ret
491     MyProcedure ENDP
492
493     main PROC
494         ; Push all general-purpose registers onto the stack
495         pushad
496         ; Call MyProcedure
497         call MyProcedure
498         ; Pop all general-purpose registers from the stack
499         popad
500         ; Push processor flags onto the stack
501         pushfd
502         ; Modify some flags (for demonstration purposes)
503         cld ; Clear direction flag
504         ; Pop processor flags from the stack
505         popfd
506         ; Display a message
```

```
507     mov eax, 4          ; syscall number for sys_write (Linux)
508     mov ebx, 1          ; file descriptor 1 (stdout)
509     mov ecx, offset message ; pointer to the message
510     mov edx, 13         ; message length
511     int 0x80            ; invoke syscall
512
513     ; Exit program
514     mov eax, 1          ; syscall number for sys_exit (Linux)
515     int 0x80            ; invoke syscall
516
      main ENDP
END main
```

## *Defining and Using Procedures*

Procedures are subroutines that can be used to divide programs into separate tasks. This makes the program easier to understand, implement, and test.

In assembly language, a procedure is a block of code that is identified by a name. The procedure can be called from anywhere in the program.



Procedures can be used to encapsulate data and code. This makes the program more modular and reusable.

Assembly language programmers must impose their own formal structure on programs, as it does not have the formal structure found in object-oriented languages.



---

## ***PROC Directive***

---

The PROC directive is used to define a procedure in assembly language. A procedure is a named block of code that can be called from anywhere in the program.

To define a procedure, you use the PROC directive followed by the name of the procedure. The procedure then ends with the **ENDP directive**.

For example, the following code defines a procedure called SumOf:

```
SumOf PROC
add
eax,ebx
add
eax,ecx
ret
SumOf ENDP
```

This procedure calculates the sum of three 32-bit integers and returns the result in the EAX register.

## ***Calling a Procedure***

To call a procedure, you simply use the name of the procedure. For example, the following code calls the SumOf procedure:

```
mov eax, 1
mov ebx, 2
mov ecx, 3

call SumOf

; eax now contains the sum of 1, 2, and 3, which is 6
```

### ***Returning from a Procedure***

To return from a procedure, you use the RET instruction. This instruction causes the CPU to return to the location from where the procedure was called.

The PROC directive is a powerful tool that allows you to define procedures in assembly language. Procedures can be used to divide programs into separate tasks, which makes the program easier to understand, implement, and test.

=====

### ***Call and RET Instructions***

=====

The CALL instruction calls a procedure by pushing the return address on the stack and copying the called procedure's address into the instruction pointer.

When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer.

The following example shows how to call and return from a procedure:

```
.code
main PROC
    ; Call the MyProcedure
    call MyProcedure

    ; Continue executing after the call

    ; Exit the program
    exit

MyProcedure PROC
    ; Your procedure code here

    ; Return to the caller
    ret

MyProcedure ENDP

main ENDP
END main
```

Now, let's break down and explain the example:

**.code Section:** This section contains the main program and the MyProcedure procedure.

**main PROC:** This is the starting point of the program.

**call MyProcedure:** The CALL instruction is used to call the MyProcedure procedure. When this instruction is executed, it does the following: Pushes the return address onto the stack. This return address is the address of the instruction immediately following the CALL instruction in the main program. Copies the address of the MyProcedure into the instruction pointer (EIP). After the CALL instruction, you can include any code you want to execute after calling the procedure.

**MyProcedure PROC:** This is the beginning of the MyProcedure procedure. Inside MyProcedure, you can include any code specific to this procedure. This is where you define what the procedure does.

**ret:** The RET instruction is used to return from the procedure. When this instruction is executed, it does the following: Pops the return address from the stack into the instruction pointer (EIP). This effectively "jumps back" to the instruction immediately after the CALL in the main program, continuing execution from there.

**MyProcedure ENDP:** This marks the end of the MyProcedure procedure.

**main ENDP:** This marks the end of the main program.

**END main:** This indicates the end of the assembly program.

In summary, the CALL instruction is used to call a procedure, and the RET instruction is used to return from the procedure to the caller. The return address is stored on the stack during the CALL and retrieved from the stack during the RET, allowing for structured program flow and function calls in assembly language.

=====

**Nested Procedure Calls**

=====

A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns.

The following example shows a nested procedure call:

```
.code
main PROC
    ; Call the OuterProcedure
    call OuterProcedure
    ; Continue executing after the call to OuterProcedure
    ; Exit the program
    exit

OuterProcedure PROC
    ; Some code in the OuterProcedure
    ; Call the InnerProcedure
    call InnerProcedure
    ; Continue executing in OuterProcedure after the call to InnerProcedure
    ; Return from OuterProcedure
    ret

OuterProcedure ENDP
InnerProcedure PROC
    ; Some code in the InnerProcedure
    ; Return from InnerProcedure
    ret

InnerProcedure ENDP
main ENDP
END main
```

Now, let's break down and explain the example of nested procedure calls:

**.code Section:** This section contains the main program and two procedures, OuterProcedure and InnerProcedure.

**main PROC:** This is the starting point of the program.

**call OuterProcedure:** The main program calls OuterProcedure using the CALL instruction. When this happens, control is transferred to OuterProcedure.

**OuterProcedure PROC:** Inside OuterProcedure, there is some code specific to this procedure. After executing this code, it calls InnerProcedure using the CALL instruction.

**call InnerProcedure:** The CALL instruction within OuterProcedure invokes InnerProcedure. Control is transferred to InnerProcedure, and the execution continues there.

**InnerProcedure PROC:** Inside InnerProcedure, there is some code specific to this procedure. After executing its code, InnerProcedure returns using the RET instruction.

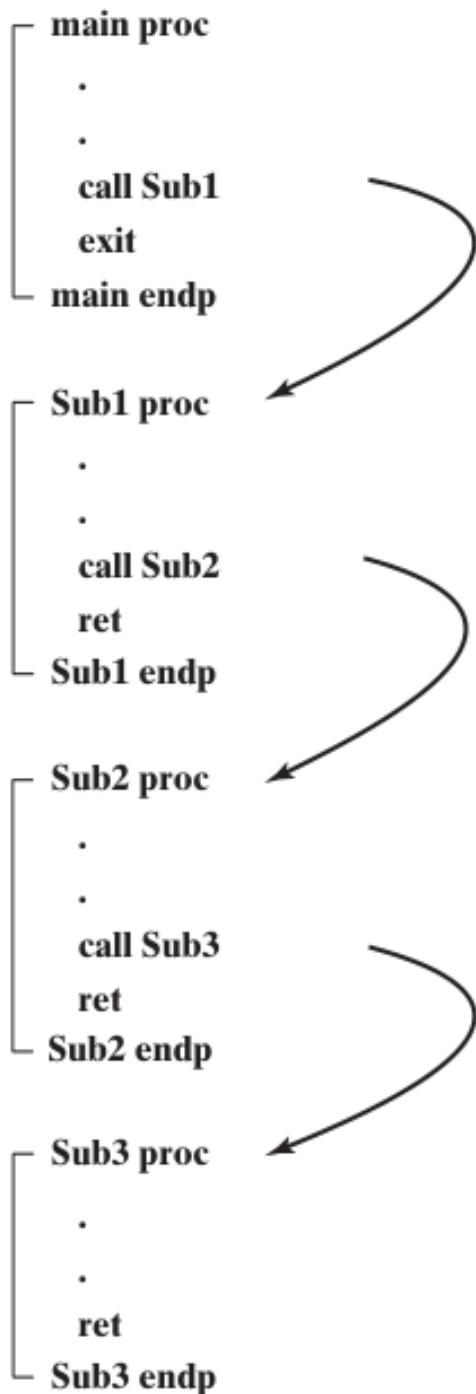
**ret (InnerProcedure):** The RET instruction in InnerProcedure causes control to return to the point immediately after the CALL instruction in OuterProcedure.

**ret (OuterProcedure):** Finally, when OuterProcedure completes its execution, it returns using the RET instruction. This brings control back to the instruction immediately after the CALL instruction in the main program.

The main program can then continue executing after the call to OuterProcedure.

In summary, this example demonstrates nested procedure calls. The main program calls OuterProcedure, which in turn calls InnerProcedure. Each procedure has its own code and can perform specific tasks.

The use of CALL and RET instructions facilitates structured program flow, allowing you to call procedures within procedures and manage control flow effectively.



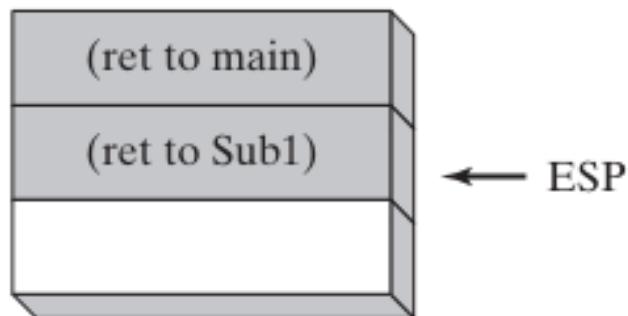
The main procedure calls the Sub1 procedure. Sub1 calls Sub2, and Sub2 calls Sub3. Sub3 then returns to Sub2, Sub2 returns to Sub1, and Sub1 returns to main.

Here is a more detailed explanation of the flow:

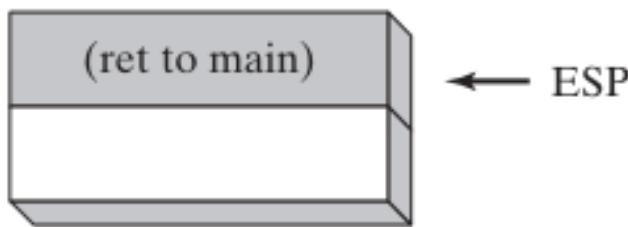
- The main procedure calls Sub1 by executing the CALL Sub1 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub1.
- The CALL instruction then loads the address of Sub1 into the instruction pointer.
- The Sub1 procedure begins executing. Sub1 calls Sub2 by executing the CALL Sub2 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub2.
- The CALL instruction then loads the address of Sub2 into the instruction pointer.
- The Sub2 procedure begins executing.
- Sub2 calls Sub3 by executing the CALL Sub3 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub3.
- The CALL instruction then loads the address of Sub3 into the instruction pointer.
- The Sub3 procedure begins executing. Sub3 returns by executing the RET instruction.
- The RET instruction pops the return address for Sub2 from the stack into the instruction pointer.
- Sub2 begins executing again.
- Sub2 returns by executing the RET instruction.
- The RET instruction pops the return address for Sub1 from the stack into the instruction pointer.
- Sub1 begins executing again.

- Sub1 returns by executing the RET instruction.
  - The RET instruction pops the return address for main from the stack into the instruction pointer.
  - The main procedure begins executing again.
  - The stack is used to keep track of the return addresses for the nested procedure calls.
  - When a procedure calls another procedure, it pushes its return address onto the stack.
  - When a procedure returns, it pops its return address from the stack.
  - This ensures that the procedures return to the correct location in the program.
-

After the return, ESP points to the next-highest stack entry. When the RET instruction at the end of **Sub2** is about to execute, the stack appears as follows:



Finally, when **Sub1** returns, stack[ESP] is popped into the instruction pointer, and execution resumes in **main**:



Clearly, the stack proves itself a useful device for remembering information, including nested procedure calls. Stack structures, in general, are used in situations where programs must retrace their steps in a specific order.

Here is a more detailed explanation of the image:

- The ret to main entry contains the address of the instruction following the CALL Sub1 instruction in the main procedure.

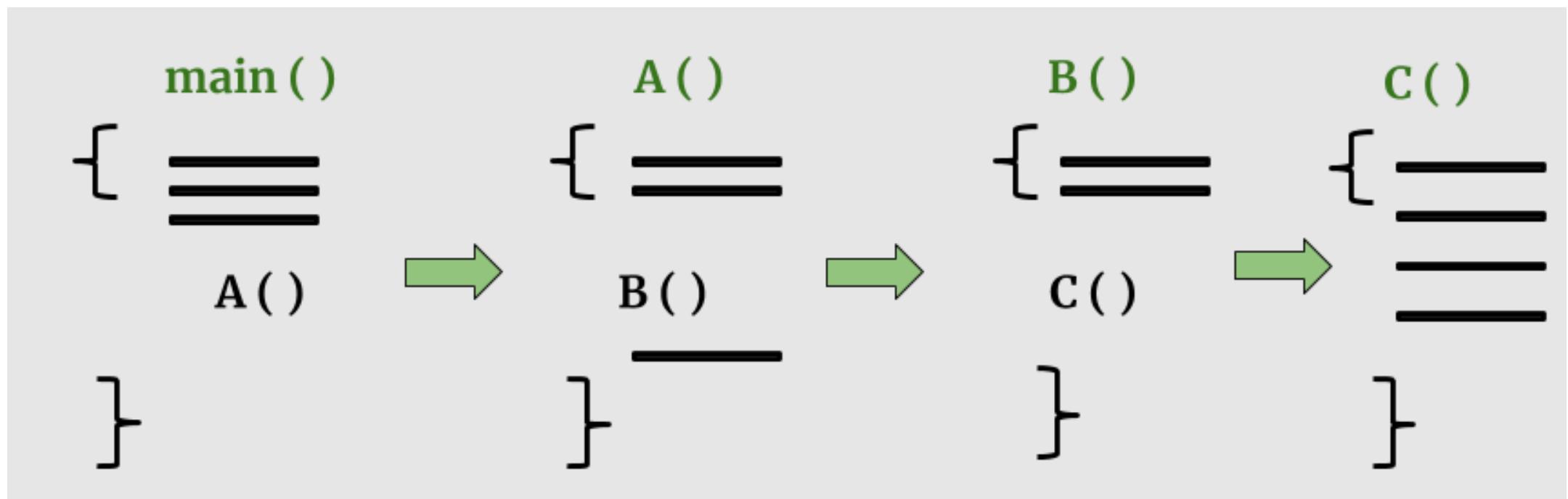
- This is the return address for the main procedure.
  - The ret to Sub1 entry contains the address of the instruction following the CALL Sub2 instruction in the Sub1 procedure.
  - This is the return address for the Sub1 procedure.
  - The ret to Sub2 entry contains the address of the instruction following the CALL Sub3 instruction in the Sub2 procedure.
  - This is the return address for the Sub2 procedure.
  - The stack is used to keep track of the return addresses for the nested procedure calls.
  - When a procedure calls another procedure, it pushes its return address onto the stack.
  - When a procedure returns, it pops its return address from the stack.
  - This ensures that the procedures return to the correct location in the program.
  - In the image, Sub3 has just returned.
  - The RET instruction in Sub3 popped the return address for Sub2 from the stack and loaded it into the instruction pointer.
  - This caused Sub2 to begin executing again.
  - The next instruction to be executed is the instruction after the CALL Sub3 instruction in the Sub2 procedure.
  - After Sub2 finishes executing, it will return to Sub1 by executing the RET instruction.
  - The RET instruction in Sub2 will pop the return address for Sub1 from the stack and load it into the instruction pointer.
  - This will cause Sub1 to begin executing again.
  - The same process will continue until all of the procedures have returned.
  - The last procedure to return will be the main procedure.
- 

So, this nesting is basically, putting the address of the procedure that called me onto the stack, executing my code, then returning causes a pop of the address of the procedure that called me? And I will only press return if I don't call another procedure within me?

Yes, your understanding of nested procedure calls is correct. When a procedure is called, the address of the instruction immediately following the CALL instruction is pushed onto the stack as the return address.

This allows the called procedure to execute its code and, when it's ready to return, it uses the RET instruction. The RET instruction pops the return address from the stack and transfers control back to the instruction with that address, effectively "returning" to the calling procedure.

Here's a step-by-step explanation of the process:



The calling procedure (let's call it Procedure A) encounters a CALL instruction to call another procedure (Procedure B).

Before transferring control to Procedure B, Procedure A pushes its own return address (the address of the instruction immediately following the CALL) onto the stack.

Procedure B is executed. When Procedure B is ready to return, it uses the RET instruction.

The RET instruction pops the return address (which is the address of the instruction following the CALL in Procedure A) from the stack.

Control is transferred back to Procedure A, continuing from where it left off.

Your statement, "**I will only press return if I don't call another procedure within me,**" is also accurate.

When a procedure calls another procedure within it (nested calls), it won't use RET until all nested calls are completed.

Each RET instruction corresponds to a single CALL instruction, and they must match.

If you call another procedure within the current one, you need to ensure that all inner procedure calls are resolved before the outer procedure returns.

This maintains the proper stack structure and allows for correct program execution.

---

```
ArraySum PROC
    ; Input:
    ;   ecx = array size
    ;   esi = pointer to the start of the array
    ; Output:
    ;   eax = sum of array elements

    xor eax, eax          ; Initialize eax to 0 (sum)
L1:
    add eax, [esi]        ; Add the current element to the sum
    add esi, 4             ; Move to the next element (assuming 32-bit integers)
    loop L1               ; Repeat for the remaining elements
    ret
```

```
ArraySum ENDP
```

The code snippet you provided appears to be part of a procedure named ArraySum, which is designed to calculate the sum of an array of 32-bit integers.

This code is not directly related to nested procedures but rather demonstrates the concept of using procedures for code modularity and reusability.

**loop L1:** This instruction is likely inside a loop that iterates through an array. It decrements the ecx register (loop counter) and checks if it's zero. If ecx is not zero, it jumps back to the L1 label, effectively looping through the array elements.

**pop ecx:** After the loop completes, it pops the original value of ecx from the stack. This is often done to restore the loop counter to its original value, allowing the caller to access it.

**pop esi:** Similarly, it pops the esi register from the stack. This might be used to restore the pointer or index used to access the array elements.

**ret:** This is the return instruction, indicating that the procedure is finished. The sum of the array is expected to be in the eax register when the procedure returns.

In this rewritten version, the ArraySum procedure takes ecx as the array size and esi as the pointer to the start of the array. It initializes eax to 0 for the sum and uses a loop to iterate through the array, adding each element to the sum. This makes the procedure more flexible and suitable for various array sizes and starting addresses.

```
; Testing the ArraySum procedure (TestArraySum.asm)
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
    array DWORD 10000h, 20000h, 30000h, 40000h, 50000h ; An array of 32-bit integers
    theSum DWORD ? ; Variable to store the sum

.code
main PROC
    mov esi, OFFSET array ; ESI points to the array
    mov ecx, LENGTHOF array ; ECX = number of elements in the array
    call ArraySum ; Call the ArraySum procedure to calculate the sum
    mov theSum, eax ; Store the returned sum in theSum variable
    INVOKE ExitProcess, 0 ; Exit the program

main ENDP

;-----
; ArraySum
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
;             ECX = number of elements in the array
; Returns:  EAX = sum of the array elements
;-----
ArraySum PROC
    push esi ; Save ESI (pointer to the array)
    push ecx ; Save ECX (number of elements)
    mov eax, 0 ; Initialize the sum to zero
```

L1:

```
add eax, [esi] ; Add each integer to the sum  
add esi, TYPE DWORD ; Move to the next integer (assuming 32-bit integers)  
loop L1 ; Repeat for the remaining elements
```

```
pop ecx ; Restore ECX  
pop esi ; Restore ESI  
ret ; Return with the sum in EAX
```

ArraySum ENDP

This program tests the ArraySum procedure, which calculates the sum of an array of 32-bit integers. Let's break it down step by step:

- **.386 and .model:** These directives specify the target processor (80386) and the memory model (flat), respectively.
- **.stack 4096:** This directive reserves 4096 bytes of stack memory for the program.
- **ExitProcess PROTO, dwExitCode:DWORD:** This line declares the ExitProcess function prototype, which is used later to exit the program with an exit code.
- **array DWORD 10000h, 20000h, 30000h, 40000h, 50000h:** This defines an array of 32-bit integers named array. It contains five elements with the values 0x10000, 0x20000, 0x30000, 0x40000, and 0x50000.
- **theSum DWORD ?:** This declares a variable named theSum to store the sum of the array. The ? indicates that it's uninitialized.

- **main PROC:** This marks the beginning of the main procedure. `mov esi, OFFSET array`: This sets esi to the memory address (offset) of the array variable, effectively pointing to the beginning of the array.
- **mov ecx, LENGTHOF array:** This loads ecx with the number of elements in the array, which is 5 in this case.
- **call ArraySum:** This calls the ArraySum procedure to calculate the sum of the array elements.
- **mov theSum, eax:** After ArraySum returns, the sum is stored in eax, and this line moves it to the theSum variable.
- **INVOKE ExitProcess, 0:** This invokes the ExitProcess function with an exit code of 0, terminating the program.

Here's a detailed explanation of the ArraySum procedure:

- **push esi and push ecx:** These instructions push the values of esi (pointer to the array) and ecx (number of elements) onto the stack to preserve their values.
- **mov eax, 0:** This initializes eax to zero, which will be used to accumulate the sum of the array.
- **L1:::** This label marks the beginning of a loop.
- **add eax, [esi]:** This instruction adds the value at the memory location pointed to by esi (the current element of the array) to eax, effectively accumulating the sum.
- **add esi, TYPE DWORD:** This increments esi by the size of a DWORD (32 bits) to move to the next element in the array.

- **loop L1:** This instruction decrements ecx (the loop counter) and checks if it's zero. If ecx is not zero, it jumps back to the L1 label, repeating the addition for the remaining elements of the array.
- **pop ecx and pop esi:** These instructions restore the original values of ecx and esi that were pushed onto the stack at the beginning of the procedure.
- **ret:** This instruction indicates the end of the procedure and returns with the sum of the array elements in eax.

In summary, the ArraySum procedure is called from the main program to calculate the sum of the array. It uses a loop to iterate through the elements, accumulating the sum in eax, and then returns the result.

-----

Let's review the concepts and questions related to procedures and the RET instruction:

**(True/False): The PROC directive begins a procedure, and the ENDP directive ends a procedure.**

True. The PROC directive is used to begin the definition of a procedure, and the ENDP directive marks the end of that procedure.

**(True/False): It is possible to define a procedure inside an existing procedure.**

True. In many assembly languages, including x86 assembly, it is possible to define a procedure inside another procedure. These are called nested procedures. However, it's important to maintain proper stack and register management when doing so.

**What would happen if the RET instruction was omitted from a procedure?**

If the RET instruction is omitted from a procedure, the program would not properly return control to the calling code. This can lead to unpredictable behavior, memory leaks, or crashes, as the stack would not be properly adjusted, and the program counter would not be updated.

**How are the words Receives and Returns used in the suggested procedure documentation?**

In procedure documentation, the word "Receives" is often used to describe the input parameters or arguments that the procedure expects. These are the values or data that are passed to the procedure when it's called. The word "Returns" is used to describe what the procedure provides as output. This typically includes the result of the procedure's computation, often in a specific register like EAX.

**(True/False): The CALL instruction pushes the offset of the CALL instruction on the stack.**

False. The CALL instruction does not push the offset of the CALL instruction on the stack. It pushes the return address, which is the address of the instruction immediately following the CALL instruction.

**(True/False): The CALL instruction pushes the offset of the instruction following the CALL on the stack.**

True. The CALL instruction pushes the offset of the instruction immediately following the CALL on the stack. This is necessary so that the program knows where to continue execution when the called procedure returns. In summary, procedures are defined using the PROC and ENDP directives, and the RET instruction is used to return from a procedure. Proper stack management is essential to ensure correct program flow. The terms "Receives" and "Returns" are used in procedure documentation to describe input and output, respectively. Finally, the CALL instruction pushes the return address on the stack, allowing for structured program flow and function calls.

(True/False): A link library consists of assembly language source code.

False. A link library does not consist of assembly language source code. Instead, it contains precompiled machine code procedures (subroutines) that can be used by programs. Use the PROTO directive to declare a procedure named MyProc in an external link library.

To declare a procedure named MyProc in an external link library, you would typically use the PROTO directive like this:

```
MyProc PROTO
```

This declares the existence of a procedure named MyProc without specifying its parameters or return type. The linker will resolve the actual details when linking to the external library.

Write a CALL statement that calls a procedure named MyProc in an external link library. To call a procedure named MyProc in an external link library, you would use a CALL statement like this:

```
call MyProc
```

The linker will ensure that the correct machine code for MyProc is linked into your program when you build it. What is the name of the 32-bit link library supplied with this book?

The name of the 32-bit link library supplied with the book is **Irvine32.lib**. This library contains procedures that link to the MS-Windows API for input and output operations and other functionality.

## **What type of file is kernel32.dll?**

kernel32.dll is a dynamic link library (DLL) file. It contains executable functions that perform various system-level tasks, including character-based input and output. These DLLs are an essential part of the Windows operating system and can be linked to from assembly language programs to access their functionality. These answers provide a better understanding of how external libraries are used in assembly language programming, especially in the context of the book's examples.

## ***Irvine32 Library***

**ReadInt:** Reads a 32-bit signed decimal integer from the keyboard, terminated by the Enter key.

### **READINT**

**.ReadKey:** Reads a single character from the keyboard's input buffer without waiting for input. Useful for detecting key presses.



**ReadString:** Reads a string from the keyboard, terminated by the Enter key.

# READSTRING() FUNCTION



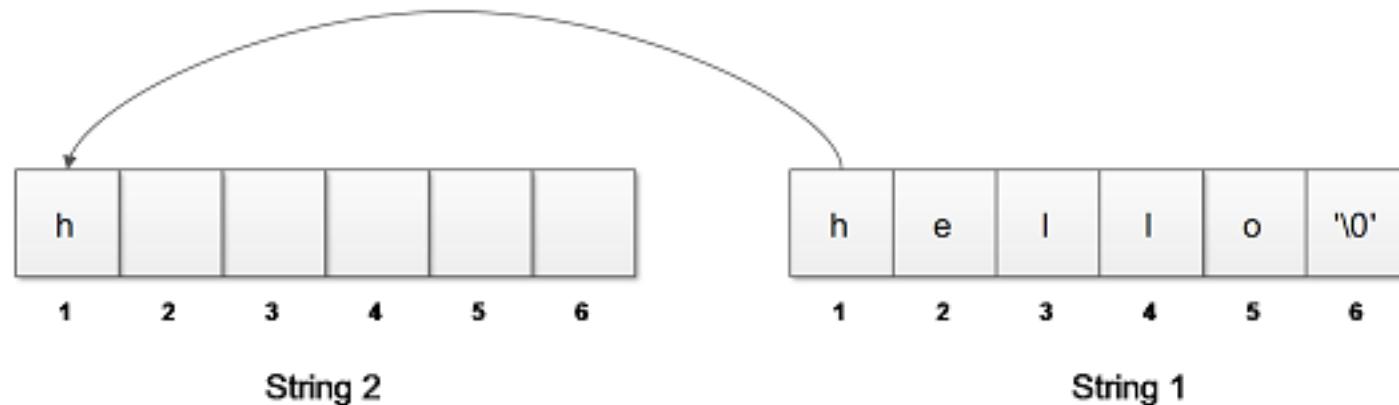
**SetTextColor:** Sets the foreground and background colors for all subsequent text output to the console.



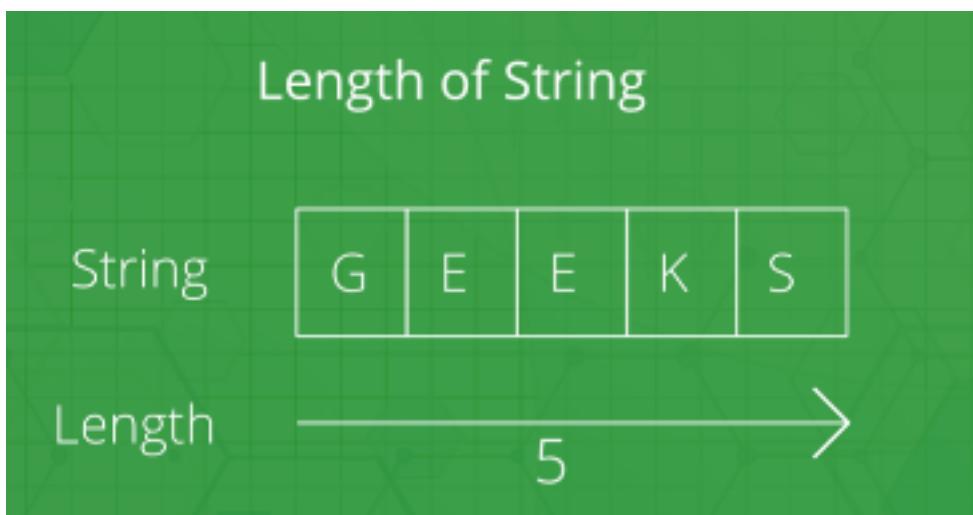
**Str\_compare:** Compares two strings and returns a result indicating their relationship.

## String Comparison Function **strcmp()**

**Str\_copy:** Copies a source string to a destination string.



**Str\_length:** Returns the length of a string in EAX (in characters).



**Str\_trim:** Removes unwanted characters (specified by a given character set) from a string.

“ Geeks for geeks 

3 spaces in front

2 spaces in end

**trim()**

“Geeks for geeks”

**Str\_uppercase:** Converts a string to uppercase letters.

Hello Python



HELLO PYTHON

**WaitMsg:** Displays a message and waits for a key to be pressed before continuing.



**WriteBin:** Writes an unsigned 32-bit integer to the console window in ASCII binary format.

sparklyr/sparklyr

#3297 **Error in**  
**writeBin(as.integer(value**  
**con, endian = "big")**



**WriteBinB:** Writes a binary integer to the console window in byte, word, or doubleword format.

```
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000  
0000 0000 0000 0000 0000 0000 0011 0000
```

**WriteChar:** Writes a single character to the console window.

micropython/micropython

## #4694 **UART readchar** **writechar**

**WriteDec:** Writes an unsigned 32-bit integer to the console window in decimal format.

## Example: Displaying an Integer

```
.code
    mov  eax, -1000
    call WriteBin           ; display binary
    call Crlf
    call WriteHex          ; display hexadecimal
    call Crlf
    call WriteInt          ; display signed decimal
    call Crlf
    call WriteDec          ; display unsigned decimal
    call Crlf
```

Sample output

```
1111 1111 1111 1111 1100 0001 1000
FFFFFC18
-1000
4294966296
```

**WriteHex:** Writes a 32-bit integer to the console window in hexadecimal format.

**WriteHexB:** Writes a byte, word, or doubleword integer to the console window in hexadecimal format.

**WriteInt:** Writes a signed 32-bit integer to the console window in decimal format.

**WriteStackFrame:** Writes the current procedure's stack frame information to the console.

# The Stack Frame

**WriteStackFrameName:** Writes the name of the current procedure and its stack frame information to the console.

**WriteString:** Writes a null-terminated string to the console window.

**WriteToFile:** Writes a buffer of data to an output file.

## WRITE TO FILE

**WriteWindowsMsg:** Displays a string containing the most recent error message generated by MS-Windows.

```
RVER:servername      Send message to all sessions on specified server.  
ME:seconds          Time delay to wait for receiver to acknowledge msg.  
                    Display information about actions being performed.  
                    Wait for response from user, useful with /V.  
sage                Message to send. If none specified, prompts for it  
                    or reads from file.  
  
ers\ashis>msg ashis  
message to send; end message  
are you  
  
hello  
where are you  
OK
```

Each of these procedures serves specific purposes, such as input/output operations, string manipulation, and text formatting, and can be extremely helpful for assembly language programmers to simplify common tasks.

---

Overview of Console Window The console window is a text-only window in MS-Windows created when a command prompt is displayed. You can customize its size, font size, and colors.

- Mode Command: You can use the "mode" command in the command prompt to change the number of columns and lines in the console window. For example:

```
mode con cols=40 lines=30
```

---

## *Individual Procedure Descriptions*

=====

Here are descriptions of some of the procedures in the Irvine32 library:

### 1. *CloseFile*

- Closes a file that was previously created or opened using a file handle. Pass the file handle in EAX.

```
mov eax, fileHandle call CloseFile
```

### 2. *Clrscr*

- Clears the console window. Typically called at the beginning and end of a program to clear the screen. If called at other times, you may need to pause using WaitMsg to allow the user to view existing screen contents.

```
call WaitMsg call Clrscr
```

### 3. *CreateOutputFile*

- Creates a new disk file and opens it for writing. Provide the filename offset in EDX. Returns a valid file handle (32-bit integer) in EAX if successful, otherwise INVALID\_HANDLE\_VALUE.

```
mov edx, offset filename call CreateOutputFile
```

#### 4. Crlf

- The Crlf procedure advances the cursor to the beginning of the next line in the console window. It essentially moves to the next line. Sample Call:

```
call Crlf
```

This instruction calls the Crlf procedure. The Crlf procedure stands for "Carriage Return" and "Line Feed." It is used to advance the cursor to the beginning of the next line in the console window. Essentially, it simulates pressing the Enter key, which moves the cursor to the next line as in a typical text editor.

#### 5. Delay

- The Delay procedure pauses the program for a specified number of milliseconds. You need to set EAX to the desired interval before calling it. Sample Call:

```
mov eax, 1000 ; 1 second call Delay
```

This instruction calls the Delay procedure. The Delay procedure is used to pause the program for a specified number of milliseconds. In this case, it's designed to create a 1-second delay since EAX is set to 1000 (milliseconds).

## 6. DumpMem

- The DumpMem procedure writes a range of memory to the console window in hexadecimal format. It requires you to pass the starting address in ESI, the number of units in ECX, and the unit size in EBX.

```
mov esi, OFFSET array ; starting OFFSET
mov ecx, LENGTHOF array ; number of units
mov ebx, TYPE array ; doubleword format
call DumpMem
```

This call displays the content of the array in hexadecimal.

Here's an explanation of the provided assembly code:

**mov esi, OFFSET array:** This instruction loads the memory address (offset) of the array into the ESI register. It's setting up ESI to point to the starting address of the array in memory.

**mov ecx, LENGTHOF array:** This instruction loads the length of the array in terms of the number of elements into the ECX register. It determines how many units of data will be displayed when calling DumpMem.

**mov ebx, TYPE array:** This instruction loads the data type of the array elements into the EBX register. It specifies the format of the data when calling DumpMem. In this case, it indicates that the elements are in doubleword format.

`call DumpMem`: This instruction calls the DumpMem procedure. The DumpMem procedure is responsible for displaying a range of memory in hexadecimal format. It takes the starting address in ESI, the number of units in ECX, and the unit size in EBX (in this case, doubleword format). It will display the contents of the array specified by ESI, ECX, and EBX in hexadecimal format.

In summary, these instructions set up the parameters for the DumpMem procedure to display the contents of the array in doubleword format from its starting address with the specified length.

## 7. *DumpRegs*

- The DumpRegs procedure displays the values of various registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFL), as well as the Carry, Sign, Zero, Overflow, Auxiliary Carry, and Parity flags in hexadecimal.

### `call DumpRegs`

This instruction calls the DumpRegs procedure. The DumpRegs procedure displays the values of various CPU registers and flags in hexadecimal format. It provides a snapshot of the CPU's current state, which can be helpful for debugging and understanding the program's execution.

## 8. *GetCommandTail*

- GetCommandTail copies the program's command line into a null-terminated string and checks if the command line is empty.
- To use it, you must provide the offset of a buffer in EDX where the command line will be stored.

Sample Call:

```
.data
    cmdTail BYTE 129 DUP(0) ; empty buffer
.code
    mov edx, OFFSET cmdTail
    call GetCommandTail ; fills the buffer
```

## 9. *GetMaxXY*

- The GetMaxXY procedure retrieves the size of the console window's buffer. The number of buffer columns is stored in DX, and the number of buffer rows is stored in AX. Sample Call:

```
.data
    rows BYTE ?
    cols BYTE ?
.code
    call GetMaxXY
    mov rows, al
    mov cols, dl
```

## 10. *GetMseconds*

- GetMseconds returns the number of milliseconds elapsed since midnight on the host computer in the

EAX register. It's useful for measuring time between events. Sample Call:

```
.data
    startTime DWORD ?
.code
    call GetMseconds
    mov startTime, eax
L1:
; (loop body)
loop L1
call GetMseconds
sub eax, startTime ; EAX = loop time, in milliseconds
```

This example measures the execution time of a loop.

## 11. *GetTextColor*

- The *GetTextColor* procedure retrieves the current foreground and background colors of the console window. It has no input parameters. The background color is stored in the upper four bits of AL, and the foreground color is stored in the lower four bits. Sample Call:

```
.data  
    color BYTE ?  
.code  
    call GetTextColor  
    mov color, al
```

Here's a breakdown of what this code does:

1. **.data color BYTE ?:** This declares a byte-sized variable named color in the data section. This variable will be used to store the color information retrieved from the console.
2. **.code call GetTextColor:** This line calls the GetTextColor procedure, which retrieves the current text color attributes of the console window. The result is returned in the AL register.
3. **mov color, al:** This instruction moves the value in the AL register (which contains the retrieved color information) into the color variable declared earlier. It stores the current text color in the color variable.

So, after executing these instructions, the color variable will contain the current text color attribute of the console window, allowing you to use or manipulate it in your program as needed.

## 12. Gotoxy

- The Gotoxy procedure positions the cursor at a specific row and column in the console window. You should pass the row (Y-coordinate) in DH and the column (X-coordinate) in DL. Sample Call:

```
mov dh, 10 ; row 10
mov dl, 20 ; column 20
call Gotoxy ; position cursor
```

Here's a breakdown of what this code does:

**mov dh, 10:** This instruction moves the value 10 into the DH (Destination High) register, which represents the row number where you want to position the cursor. In this case, it sets the row to 10.

**mov dl, 20:** This instruction moves the value 20 into the DL (Destination Low) register, which represents the column number where you want to position the cursor. It sets the column to 20.

**call Gotoxy:** This is a procedure call to the Gotoxy procedure. When called, it takes the values in DH (row) and DL (column) and uses them to position the cursor in the console window.

So, after executing these instructions, the cursor will be moved to row 10 and column 20 in the console window. This can be useful for controlling the cursor's position when you want to display text or interact with the user at specific locations on the screen.

### 13. *IsDigit*

- IsDigit checks whether the value in AL is the ASCII code for a valid decimal digit. If AL contains a valid decimal digit, it sets the Zero flag (ZF); otherwise, it clears ZF. Sample Call:

```
mov AL, somechar  
call IsDigit
```

You are trying to determine whether the value in the AL register represents a valid decimal digit by calling the IsDigit procedure.

You move the value of somechar into the AL register using mov AL, somechar. somechar presumably contains an ASCII character.

You call the IsDigit procedure with the current value in the AL register using call IsDigit.

Inside the IsDigit procedure, it checks if the value in AL is a valid ASCII code for a decimal digit. If it is, the procedure sets the Zero flag (ZF) to 1. If it's not a valid digit, the ZF is cleared (set to 0).

After calling IsDigit, you can check the state of the ZF to determine whether the character in AL is a valid decimal digit or not.

If ZF is set (ZF = 1), then somechar represents a valid decimal digit. If ZF is cleared (ZF = 0), then somechar does not represent a valid decimal digit.

This code is useful if you want to validate whether a character is a decimal digit in your program.

## 14. MsgBox

The MsgBox procedure displays a popup message box with an optional caption. Pass it the offset of a string in EDX to appear inside the box, and optionally, pass the offset of a string for the box's

title in EBX. To leave the title blank, set EBX to zero.

```
.data
    caption BYTE "Dialog Title", 0
    HelloMsg BYTE "This is a pop-up message box.", 0dh,0ah, "Click OK to continue...", 0
.code
    mov ebx, OFFSET caption
    mov edx, OFFSET HelloMsg
    call MsgBox
```

You are setting up a message box with a title ("Dialog Title") and a message ("This is a pop-up message box. Click OK to continue..."). Then, you are calling the MsgBox procedure to display this message box.

You define two null-terminated strings in the .data section. caption contains the title of the message box, and HelloMsg contains the message along with a line break (0dh,0ah) and the instruction "Click OK to continue..."

You load the address of the caption string into the EBX register using mov ebx, OFFSET caption. This prepares the address of the title string to be passed as a parameter to the MsgBox procedure.

You load the address of the HelloMsg string into the EDX register using mov edx, OFFSET HelloMsg. This prepares the address of the message string to be passed as a parameter to the MsgBox procedure.

You call the MsgBox procedure using call MsgBox. The procedure displays a graphical popup message box with an OK button. It expects the address of the title in EBX and the address of the message in EDX.

After the user clicks the OK button in the message box, the MsgBox procedure returns, and your program continues executing any subsequent code.

This code demonstrates how to display a simple informational message box with a title and a message, allowing the user to acknowledge the message by clicking the OK button.

## 15. *MsgBoxAsk*

The MsgBoxAsk procedure displays a popup message box with Yes and No buttons. It returns an integer in EAX that indicates which button was selected by the user (IDYES or IDNO).

```
.data
    caption BYTE "Survey Completed", 0
    question BYTE "Thank you for completing the test.", 0dh,0ah, "Receive results?", 0
.code
    mov ebx, OFFSET caption
    mov edx, OFFSET question
    call MsgBoxAsk ; check return value in EAX
```

You are setting up a message box with a title ("Survey Completed") and a question ("Thank you for completing the test. Receive results?"). Then, you are calling the MsgBoxAsk procedure to display this message box.

Here's what's happening step by step:

1. You define two null-terminated strings in the .data section. caption contains the title of the message box, and question contains the message along with a line break (0dh,0ah) and the question "Receive results?"
2. You load the address of the caption string into the EBX register using `mov ebx, OFFSET caption`. This prepares the address of the title string to be passed as a parameter to the `MsgBoxAsk` procedure.
3. You load the address of the question string into the EDX register using `mov edx, OFFSET question`. This prepares the address of the message/question string to be passed as a parameter to the `MsgBoxAsk` procedure.
4. You call the `MsgBoxAsk` procedure using `call MsgBoxAsk`. The procedure displays a graphical popup message box with a Yes and No button. It expects the address of the title in EBX and the address of the message/question in EDX.

After the user interacts with the message box (by clicking either Yes or No), the `MsgBoxAsk` procedure returns an integer value in EAX, which tells you which button was selected. The value will be either `IDYES` (equal to 6) if the user clicked Yes or `IDNO` (equal to 7) if the user clicked No.

To check the return value and take further actions based on the user's choice, you can use conditional statements or other logic in your code, depending on whether EAX contains `IDYES` or `IDNO`.

## ***16. OpenInputFile***

The `OpenInputFile` procedure opens an existing file for input. Pass it the offset of the filename in EDX. If the file is opened successfully, EAX contains a valid file handle; otherwise, EAX equals `INVALID_HANDLE_VALUE`.

```
.data  
    filename BYTE "myfile.txt",0  
.code  
    mov edx, OFFSET filename  
    call OpenInputFile
```

After the call, you can check the value of EAX to determine if the file was opened successfully. If EAX equals INVALID\_HANDLE\_VALUE, the file was not opened successfully.

You are defining a null-terminated string "myfile.txt" in the .data section and then using it as the filename to open an input file using the OpenInputFile procedure.

Here's what's happening step by step:

You define a null-terminated string filename with the content "myfile.txt" in the .data section.

You load the address of the filename string into the EDX register using `mov edx, OFFSET filename`. This prepares the address of the filename to be passed as a parameter to the OpenInputFile procedure.

You call the OpenInputFile procedure using `call OpenInputFile`. The procedure expects the filename (the address of the null-terminated string) in the EDX register. It will attempt to open the file with the given name for input.

After calling OpenInputFile, if the file was successfully opened, the EAX register will contain a valid file handle. If the file could not be opened, EAX will be set to INVALID\_HANDLE\_VALUE (a

predefined constant).

In your code, you have not shown how you are handling the result of opening the file (the value in EAX). Depending on whether the file was opened successfully or not, you would typically check the value in EAX and take appropriate actions, such as reading from the file or displaying an error message.

## 17. *ReadString:*

Use ReadString to read a string from the keyboard until the Enter key is pressed. Pass the offset of a buffer where the input will be stored in EDX and specify the maximum number of characters the user can enter in ECX. It returns the count of the number of characters typed by the user in EAX.

Example:

```
.data
    buffer BYTE 21 DUP(0) ; input buffer
    byteCount DWORD ? ; holds counter
.code
    mov edx, OFFSET buffer ; point to the buffer
    mov ecx, SIZEOF buffer ; specify max characters
    call ReadString ; input the string
    mov byteCount, eax ; number of characters
```

The provided code segment is using the ReadString procedure to read a string from the keyboard and save it in a buffer. Let's break down what each part of the code does:

.data section:

**buffer BYTE 21 DUP(0):** This defines a buffer named buffer that can hold up to 21 bytes (including the null terminator). It's initialized with zeros to ensure it's an empty string.

**byteCount DWORD ?:** This declares a DWORD variable named byteCount to hold the count of characters entered by the user.

.code section:

**mov edx, OFFSET buffer:** This line sets the EDX register to point to the buffer variable, effectively providing the address of the buffer to the ReadString procedure.

**mov ecx, SIZEOF buffer:** Here, the ECX register is loaded with the size of the buffer variable, which is the maximum number of characters that can be read. In this case, it's set to 21.

**call ReadString:** This line calls the ReadString procedure, which reads a string from the keyboard. It stops reading when the Enter key is pressed. The string entered by the user is stored in the buffer, and the number of characters read is returned in EAX.

**mov byteCount, eax:** Finally, this line stores the value of EAX (the number of characters read) into the byteCount variable, so you can later access and use this count in your program.

Where did eax come from?

The **ReadString procedure** reads a string from the keyboard and returns the number of characters read in the **EAX register**. After calling ReadString, the value in EAX represents the number of characters read, and it's stored in the byteCount variable for later use in your program.

In summary, this code segment initializes a buffer, reads a string from the keyboard into the buffer

using ReadString, and stores the count of characters read in the byteCount variable.

## 18. SetTextColor:

Use SetTextColor to set the foreground and background colors for text output. Set the desired color attribute in EAX, combining foreground and background colors. Example (setting text to white on a blue background).

```
mov eax, white + (blue SHL 4) ; white on blue  
call SetTextColor
```

The provided code segment is setting the text color for text output in a console window using the SetTextColor procedure. Let's break down what each part of the code does:

**mov eax, white + (blue SHL 4):** This line sets up the eax register to specify the desired text color attribute. It combines two color constants, white and blue, to achieve the desired color. Here's the breakdown:

**white and blue are color constants** defined in the Irvine32 library. white represents white text, and blue represents blue as the background color.

**(blue SHL 4)** shifts the value of blue four bits to the left. This bitwise shift operation is used to specify the background color. In this case, it's indicating a blue background.

**Adding white and the shifted blue value together** combines the foreground and background colors. So, the result is a combination of white text on a blue background.

**call SetTextColor:** This line calls the SetTextColor procedure with the color attribute specified in eax. The procedure then sets the text color for subsequent text output to the console using the specified color combination.

In summary, this code segment sets the text color to white on a blue background using the SetTextColor procedure, so any text output that follows will be displayed with this color combination in the console window.

## 19. *Str\_length:*

Use Str\_length to find the length of a null-terminated string. Pass the offset of the string in EDX. It returns the string's length in EAX.

```
.data
    buffer BYTE "abcde", 0
    bufLength DWORD ?
.code
    mov edx, OFFSET buffer ; point to string
    call Str_length ; EAX = 5
    mov bufLength, eax ; save length
```

The provided code snippet calculates the length of a null-terminated string stored in memory using the Str\_length procedure and then saves the length in a DWORD variable named bufLength. Here's a breakdown of what each part of the code does:

.data: This section defines a data segment where you declare data variables.

buffer BYTE "abcde", 0: This line declares a null-terminated string named buffer with the value "abcde". The null terminator (0) signifies the end of the string.

bufLength DWORD ?: This line declares an uninitialized DWORD variable named bufLength to store the length of the string.

.code: This section is the code segment where you write the program's instructions.

mov edx, OFFSET buffer: This instruction loads the offset of the buffer string into the edx register. It sets edx to point to the beginning of the string.

call Str\_length: This line calls the Str\_length procedure, passing the address of the buffer string as a parameter in edx. The Str\_length procedure calculates the length of the string and stores it in the eax register.

mov bufLength, eax: This instruction copies the value of eax (which now contains the length of the string) into the bufLength variable. This variable will now hold the length of the "abcde" string, which is 5 characters.

In summary, this code segment calculates and stores the length of the "abcde" string in the bufLength variable, resulting in bufLength being set to 5.

## 20. *WaitMsg*:

Use WaitMsg to display the message "Press any key to continue..." and wait for the user to press a key. Example:

```
call WaitMsg
```

The code snippet `call WaitMsg` is a function call to the `WaitMsg` procedure. Here's what it does:

**call WaitMsg:** This line of code calls the `WaitMsg` procedure, which is part of the `Irvine32` library. When you call `WaitMsg`, it displays the message "Press any key to continue..." on the console window and waits for the user to press any key.

The **purpose of using `WaitMsg`** is to pause the program's execution temporarily and provide a message to the user, prompting them to press a key to continue. This can be useful when you want to give the user time to read the output on the console before it disappears.

So, when this line of code is executed, it will display the "Press any key to continue..." message on the console, and the program will wait until the user presses a key before proceeding further.

## 21. *WriteInt:*

Use `WriteInt` to write a 32-bit signed integer to the console window in decimal format. Pass the integer in `EAX`.

```
mov eax, -12345  
call WriteInt ; displays: "-12345"
```

In this code, you are performing the following steps:

**mov eax, -12345:** This instruction moves the signed 32-bit integer value -12345 into the EAX register. It's a negative integer represented in two's complement form.

**call WriteInt:** Here, you are calling the WriteInt procedure. This procedure takes a 32-bit signed integer in the EAX register and writes it to the console window in decimal format with a leading sign if it's negative. In this case, since you loaded -12345 into EAX, it will display "-12345" on the console.

So, when you run this code, it will display the signed integer -12345 on the console window.

## 22. *WriteString:*

Use WriteString to write a null-terminated string to the console window. Pass the offset of the string in EDX.

```
.data
    greeting BYTE "Hello, Assembly Programmer!", 0
.code
    mov edx, OFFSET greeting
    call WriteString
```

In this part of the code:

**mov edx, OFFSET greeting:** This line sets the EDX register to the offset of the greeting string. In other words, it tells the program where the string is located in memory.

**call WriteString:** Here, you are calling the WriteString procedure. This procedure takes the offset

of a string (in this case, the offset of the greeting string) and writes the content of the string to the console window.

So, the code you provided is essentially displaying the "Hello, Assembly Programmer!" message on the console window using the WriteString procedure.

### 23. *WriteToFile*:

This procedure is used to write the contents of a buffer to an output file.

- Parameters: EAX (file handle), EDX (buffer address), ECX (number of bytes to write).
- Returns the number of bytes written if successful; otherwise, an error code.

```
.data
    fileHandle DWORD ?
    buffer BYTE "This is a test.", 0

.code
    mov eax, fileHandle
    mov edx, OFFSET buffer
    mov ecx, LENGTHOF buffer - 1 ; Length excluding the null terminator
    call WriteToFile
```

### 24. *WriteWindowsMsg*:

This procedure writes a string containing the most recent error generated by your application to the Console window when executing a system function. It is often used to display error messages to the user.

```
.code
    ; Some code that might generate an error
    ; ...
    ; After an error occurs, call WriteWindowsMsg to display the error message
    call WriteWindowsMsg
```

---

### *Implementing the procedures*

---

```
300 ; Library Test #1: Integer I/O (InputLoop.asm)
301 ; Tests the Clrscr, Crlf, DumpMem, ReadInt, SetTextColor,
302 ; WaitMsg, WriteBin, WriteHex, and WriteString procedures.
303
304 include Irvine32.inc
305
306 .data
307 COUNT = 4
308 BlueTextOnGray = blue + (lightGray * 16)
309 DefaultColor = lightGray + (black * 16)
310 arrayD SDWORD 12345678h, 1A4B2000h, 3434h, 7AB9h
311 prompt BYTE "Enter a 32-bit signed integer: ", 0
312
313 .code
314
315 main PROC
316     ; Select blue text on a light gray background
317     mov eax, BlueTextOnGray
318     call SetTextColor
319
320     ; Clear the screen
321     call Clrscr
322
323     ; Display an array using DumpMem.
324     mov esi, OFFSET arrayD      ; starting OFFSET
325     mov ebx, TYPE arrayD       ; doubleword = 4 bytes
326     mov ecx, LENGTHOF arrayD   ; number of units in arrayD
327     call DumpMem               ; display memory
```

```
329 ; Ask the user to input a sequence of signed integers
330 call Crlf ; new line
331 mov ecx, COUNT
332
333 L1:
334 mov edx, OFFSET prompt
335 call WriteString ; prompt the user
336 call ReadInt ; input integer into EAX
337 call Crlf ; new line
338
339 ; Display the integer in decimal, hexadecimal, and binary
340 call WriteInt ; display in signed decimal
341 call Crlf
342 call WriteHex ; display in hexadecimal
343 call Crlf
344 call WriteBin ; display in binary
345 call Crlf
346
347 Loop L1 ; repeat the loop
348
349 ; Return the console window to default colors
350 call WaitMsg ; "Press any key..."
351 mov eax, DefaultColor
352 call SetTextColor
353 call Clrscr
354
355 exit
356 main ENDP
357 END main
```

The program you've provided is a demonstration of using various procedures from the Irvine32 library to perform tasks like clearing the screen, displaying memory content, reading and displaying integers, and generating random numbers. Here's a breakdown of the program:

**Setup and Definitions:** The program starts with comments indicating its purpose and includes the Irvine32.inc library.

It defines constants COUNT, BlueTextOnGray, and DefaultColor.

An array named arrayD is declared with four signed doubleword integers. There's also a prompt message.

**Main Procedure (main PROC):**

- It sets the text color to blue on a light gray background using SetTextColor.
- Clears the screen using Clrscr.
- Displays the content of the arrayD array using the DumpMem procedure.
- It then enters a loop (L1) to ask the user to input signed integers COUNT times.
- For each input, it:
  - Prompts the user to enter an integer using WriteString. Reads the integer input into EAX using ReadInt. Displays the entered integer in decimal, hexadecimal, and binary formats using WriteInt, WriteHex, and WriteBin. After the loop, it returns the console window to default colors using WaitMsg, and clears the screen again.
  - Random Number Generation Procedures (Rand1 and Rand2):
    - Rand1 generates ten pseudo-random unsigned integers in the range 0 to 4,294,967,294 using Random32.
    - Rand2 generates ten pseudo-random signed integers in the range -50 to +49 using RandomRange.

- Main Procedure (continued):
- After defining the random number generation procedures, the program calls Randomize to initialize the random number generator.
- It then calls both Rand1 and Rand2 procedures to generate and display random integers.

Program End:

- The program ends with END main.
  - This program is a comprehensive example of how to use various Irvine32 library procedures to perform tasks related to console input and output, as well as random number generation. It also demonstrates how to format and display data in different formats (decimal, hexadecimal, and binary).
-

```
; Library Test #3: Performance Timing (TestLib3.asm)
; Calculate the elapsed execution time of a nested loop.

include Irvine32.inc

.data
OUTER_LOOP_COUNT = 3
startTime DWORD ?
msg1 BYTE "Please wait...", 0dh, 0ah, 0
msg2 BYTE "Elapsed milliseconds: ", 0

.code

main PROC
    mov edx, OFFSET msg1 ; "Please wait..."
    call WriteString

    ; Save the starting time
    call GetMSeconds
    mov startTime, eax

    ; Start the outer loop
    mov ecx, OUTER_LOOP_COUNT
L1:
    call innerLoop
    loop L1

    ; Calculate the elapsed time
    call GetMSeconds
    sub eax, startTime
```

```
; Calculate the elapsed time
call GetMSeconds
sub eax, startTime

; Display the elapsed time
mov edx, OFFSET msg2 ; "Elapsed milliseconds: "
call WriteString
call WriteDec           ; Write the milliseconds
call Crlf

exit
main ENDP

innerLoop PROC
    push ecx           ; Save current ECX value
    mov ecx, 0FFFFFFh ; Set the loop counter
L1:
    mul eax            ; Use up some cycles
    mul eax
    mul eax
    loop L1           ; Repeat the inner loop
    pop ecx            ; Restore ECX's saved value
    ret
innerLoop ENDP
END main
```

---

Certainly, let's delve into the additional information you provided regarding the Irvine64 library and some of its key procedures:

**WriteHex64:** This procedure is used to display the contents of the RAX register as a 64-bit hexadecimal integer. It's a convenient way to view the value of a 64-bit integer in hexadecimal format. Unlike some other procedures, it doesn't return any value; its purpose is to output the hexadecimal representation to the console.

**WriteHexB:** WriteHexB is a versatile procedure that displays the contents of the RAX register as a hexadecimal integer in different formats, depending on the display size specified in the RBX register. You can pass 1, 2, 4, or 8 in RBX to indicate whether you want to display the value as a 1-byte, 2-byte, 4-byte, or 8-byte hexadecimal integer, respectively. Like WriteHex64, it doesn't return a value but is used for outputting hexadecimal representations.

**WriteString:** This procedure is used to display a null-terminated ASCII string. You pass the 64-bit offset of the string in RDX, and it will output the string to the console. Like the previous procedures, it also doesn't return a value.

The Irvine64 library you mentioned is designed for 64-bit assembly programming. It's worth noting that in 64-bit assembly, the calling conventions and register usage can be different from 32-bit assembly.

For example, some registers like RAX, RCX, RDX, R8, R9, R10, and R11 may not be preserved across function calls, while others like RBX, RBP, RDI, RSI, R12, R13, R14, and R15 are usually preserved. Understanding these conventions is important when writing 64-bit assembly code.

This library, while smaller than its 32-bit counterpart, still provides essential tools for

interactive programming in a 64-bit environment.

As you progress in your 64-bit assembly programming journey, you may find it beneficial to expand this library with your own code to suit your specific needs and applications.

-----

### **Didn't we deal with those earlier on?**

You are correct; some of the procedures mentioned, such as WriteHex64, WriteHexB, and WriteString, are similar to their 32-bit counterparts that we discussed earlier.

However, the key difference here is that these procedures are designed for 64-bit assembly programming, as indicated by the use of the RAX, RBX, RDX, and other 64-bit registers.

In the previous discussions, we primarily focused on 32-bit assembly and the Irvine32 library.

The additional information you provided pertains to the Irvine64 library, which is tailored for 64-bit assembly programming.

While the basic concepts and functionality of these procedures may be similar, the implementation and usage may differ due to the nuances of 64-bit assembly, including register usage and calling conventions.

## Irvine64 Library

To call a subroutine in 64-bit assembly, you need to place input parameters in registers and then use the CALL instruction. For example:

```
mov rax, 12345678h  
call WriteHex64
```

In this example, WriteHex64 is a subroutine being called with rax as an input parameter.

Using the PROTO Directive: To inform the assembler about external procedures (subroutines) that you plan to call but are not defined within your program, you should use the PROTO directive at the top of your program.

This helps the assembler understand the calling conventions and parameter types for these external procedures. For instance:

```
ExitProcess PROTO      ; Located in the Windows API  
WriteHex64 PROTO       ; Located in the Irvine64 library
```

In this code, ExitProcess and WriteHex64 are identified as external procedures, and the PROTO directive provides information about their usage.

These guidelines are essential for proper procedure calls in 64-bit assembly with the Irvine64

library, ensuring that the correct calling conventions are followed and that the assembler can generate the appropriate code for calling external procedures.

-----

The additional content you've mentioned explains the x64 calling convention, which is used in 64-bit programs on the x86-64 architecture, such as Windows API functions and C/C++ programs. Here are the key points from this section:

**Microsoft x64 Calling Convention:** This is a consistent scheme for passing parameters and calling procedures in 64-bit programs. It's followed by C/C++ compilers and the Windows API. Some of its characteristics include:

The **CALL** instruction subtracts 8 from the **RSP (stack pointer) register** because addresses are 64 bits long.

The first four parameters passed to a procedure are placed in the **RCX, RDX, R8** and **R9** registers, in that order.

If only one parameter is passed, it goes in **RCX**, and so on.

The caller's responsibility is to **allocate at least 32 bytes of shadow space** on the runtime stack to optionally save register parameters.

When calling a **subroutine**, the **stack pointer (RSP)** must be aligned on a **16-byte boundary**, considering the 8 bytes pushed by the **CALL** instruction.

**Sample Program:** The provided sample program demonstrates how to call a subroutine (**AddFour**) using the Microsoft x64 calling convention.

```
AddFour PROC
    ; Input parameters: RCX, RDX, R8, R9
    ; Output: RAX contains the sum of the input parameters

    ; Add the input parameters
    mov rax, rcx
    add rax, rdx
    add rax, r8
    add rax, r9

    ret
AddFour ENDP
```

It adds four input parameters placed in RCX, RDX, R8, and R9 and saves the sum in RAX. The program aligns the stack pointer, sets the parameters, and calls the subroutine.

This code defines the "AddFour" subroutine, which takes four input parameters (RCX, RDX, R8, and R9) and calculates their sum, storing the result in RAX. The ret instruction is used to return from the subroutine.

You can include this code in your assembly program to call the "AddFour" subroutine using the Microsoft x64 calling convention, as demonstrated in the previous content.

**PROTO Directive:** The PROTO directive is used to declare external procedures (subroutines) at the top of your program. It helps the assembler understand how to call these procedures correctly, including

their names and parameter types.

**Stack Alignment:** Ensuring that the stack pointer (RSP) is properly aligned on a 16-byte boundary is crucial when working with the x64 calling convention. This alignment is essential to maintain proper stack integrity during procedure calls.

**Usage in Irvine64 Library:** It's worth noting that when you work with the Irvine64 library (used in these examples), you don't need to adhere to the Microsoft x64 calling convention for library procedures. It's primarily required when calling Windows API functions or external C/C++ functions.

These explanations provide a detailed understanding of how the Microsoft x64 calling convention works and how to apply it in your 64-bit assembly programs, particularly when interfacing with external functions and libraries.

-----

Certainly, here's a short assembly code snippet that demonstrates the use of the PROTO directive and stack alignment for a simple subroutine:

```
.data
    result QWORD 0      ; Declare a variable to store the result
.code
    main PROC
        sub rsp, 8      ; Align the stack pointer to a 16-byte boundary
        mov rcx, 5      ; First parameter
        mov rdx, 3      ; Second parameter

        call AddNumbers ; Call the subroutine
        add rsp, 8      ; Restore the stack pointer

        ; The result is now stored in the "result" variable
        ; You can use it as needed
        ; Exit the program
        mov ecx, 0
        call ExitProcess

    AddNumbers PROC
        ; Parameters are already in RCX and RDX
        add rax, rcx     ; Add the first parameter to RAX
        add rax, rdx     ; Add the second parameter to RAX
        mov [result], rax ; Store the result in the "result" variable
        ret
    AddNumbers ENDP

END main
```

## In this code:

We use the PROTO directive for the ExitProcess procedure, which is an external procedure used to exit the program.

We ensure proper stack alignment by subtracting 8 from rsp at the beginning of the main procedure and then adding 8 to rsp to restore it after the subroutine call.

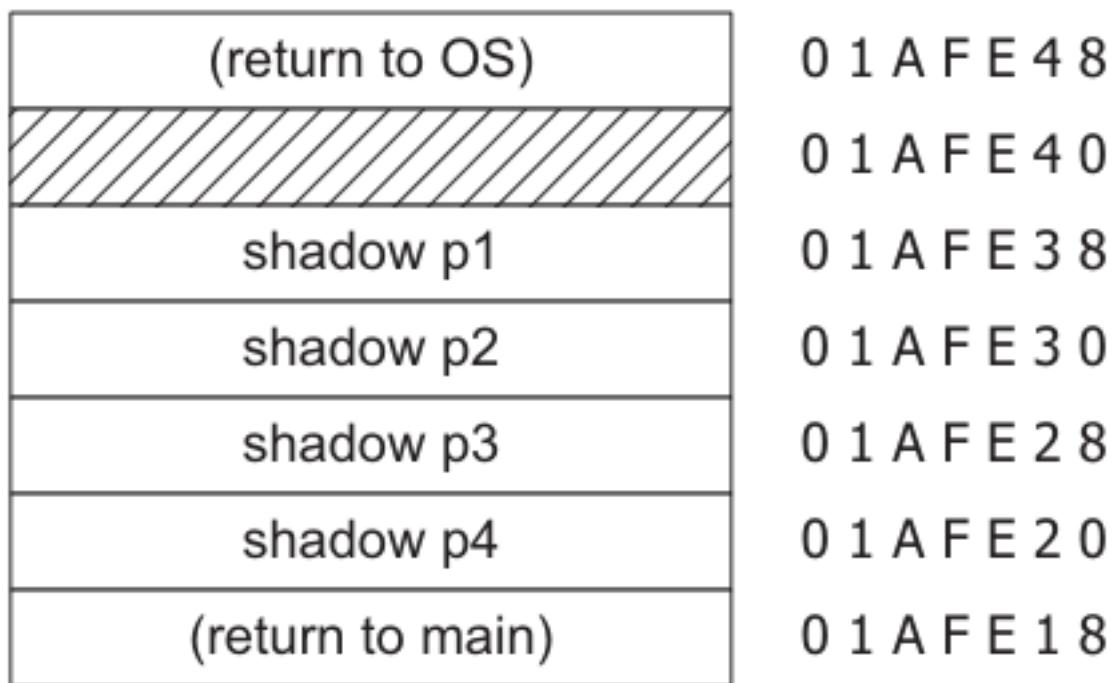
We call a subroutine AddNumbers with two parameters (rcx and rdx), which adds these parameters and stores the result in the result variable.

Finally, we exit the program using ExitProcess.

Please note that this is a simplified example, and in practice, you would use the Microsoft x64 calling convention for more complex scenarios or when calling external functions.

---

FIGURE 5–11 Runtime stack for the CallProc\_64 program.



The runtime stack for the CallProc\_64 program is a diagram that shows how the stack changes as the program executes. The stack is a region of memory that is used to store function calls, return addresses, and local variables.

The diagram shows the following:

The stack pointer (RSP) starts at address 01AFE48 before the program is called. When the OS calls the program, it subtracts 8 from the stack pointer to push the return address onto the stack.

After line 10 of the program executes, the stack pointer is at address 01AFE40, showing that the stack has been properly aligned on a 16-byte boundary.

After line 11 of the program executes, the stack pointer is at address 01AFE20, showing that 32 bytes of shadow space have been reserved on the stack.

Inside the AddFour procedure, the stack pointer is at address 01AFE18, showing that the caller's return address has been pushed onto the stack.

After AddFour returns, the stack pointer is again at address 01AFE20, the same value it had before calling AddFour. When the program reaches the end of the main procedure, it returns to the OS by executing a RET instruction.

If the program had chosen to execute an ExitProcess instruction instead, it would have been responsible for restoring the stack pointer to the way it was when the main procedure began to execute.

***Here is a more detailed explanation of each line in the diagram:***

Before line 10 executed, RSP = 01AFE48.

This tells us that RSP was equal to 01AFE50 before the OS called our program. (The CALL instruction subtracts 8 from the stack pointer.)

The RSP register is the stack pointer register. It contains the address of the top of the stack. The CALL instruction pushes the return address onto the stack, which is the address of the instruction that will be executed after the called function returns.

After line 10 executed, RSP = 01AFE40, showing that the stack was properly aligned on a 16-byte boundary.

The stack must be aligned on a 16-byte boundary for performance reasons. The stack is aligned by pushing a dummy value onto the stack before calling a function and popping the dummy value off the stack after the function returns.

After line 11 executed, RSP = 01AFE20, showing that 32 bytes of shadow space were reserved at addresses 01AFE20 through 01AFE3F.

**Shadow space** is used to store the shadow registers of a function. Shadow registers are used to save the values of the callee-saved registers before a function is called and restore them after the function returns.

Inside the AddFour procedure, RSP = 01AFE18, showing that the caller's return address had been pushed on the stack. When a function is called, the caller's return address is pushed onto the stack.

This is so that the function knows where to return to when it is finished.

After AddFour returned, RSP again was equal to 01AFE20, the same value it had before calling AddFour. When a function returns, the stack pointer is restored to the value it had before the function was called.

This is done by popping the caller's return address off the stack.

Overall, the runtime stack diagram for the CallProc\_64 program shows how the stack is used to store function calls, return addresses, and local variables.

## **Practice Questions**

---

### **QUESTIONS**

---

**Question 1: Which instruction pushes all of the 32-bit general-purpose registers on the stack?**

Answer 1: The instruction that pushes all of the 32-bit general-purpose registers on the stack is PUSHAD.

**Question 2: Which instruction pushes the 32-bit EFLAGS register on the stack?**

Answer 2: The instruction that pushes the 32-bit EFLAGS register on the stack is PUSHFD.

**Question 3: Which instruction pops the stack into the EFLAGS register?**

Answer 3: The instruction that pops the stack into the EFLAGS register is POPFD.

**Question 4: Challenge:** Another assembler (called NASM) permits the PUSH instruction to list multiple specific registers. Why might this approach be better than the PUSHAD instruction in MASM? Here is a

**NASM example: PUSH EAX EBX ECX**

Answer 4: NASM's approach of allowing the PUSH instruction to list multiple specific registers can be better in some cases because it provides more flexibility. It allows you to choose which registers to push onto the stack, whereas PUSHA in MASM pushes all the general-purpose registers. This can save stack space and execution time when you only need to save a subset of registers.

**Question 5: Challenge: Suppose there were no PUSH instruction. Write a sequence of two other instructions that would accomplish the same as push eax.**

Answer 5: If there were no PUSH instruction, you could achieve the same result as PUSH EAX using the following two instructions:

```
sub esp, 4  
mov [esp], eax
```

**Question 6: (True/False): The RET instruction pops the top of the stack into the instruction pointer.**

Answer 6: False. The RET instruction pops the return address from the stack into the instruction pointer (EIP).

**Question 7: (True/False): Nested procedure calls are not permitted by the Microsoft assembler unless the NESTED operator is used in the procedure definition.**

Answer 7: False. Nested procedure calls are permitted without the need for the NESTED operator in the Microsoft assembler.

**Question 8: (True/False): In protected mode, each procedure call uses a minimum of 4 bytes of stack space.**

Answer 8: False. In protected mode, each procedure call doesn't necessarily use a minimum of 4 bytes of stack space. The actual stack space used depends on the number of parameters and local variables.

**Question 9: (True/False): The ESI and EDI registers cannot be used when passing 32-bit parameters to procedures.**

Answer 9: False. The ESI and EDI registers can be used when passing 32-bit parameters to procedures.

**Question 10: (True/False): The ArraySum procedure (Section 5.2.5) receives a pointer to any array of doublewords.**

Answer 10: False. The ArraySum procedure from Section 5.2.5 expects a pointer to an array of doublewords specifically.

**Question 11: (True/False): The USES operator lets you name all registers that are modified within a procedure.**

Answer 11: True. The USES operator lets you specify all registers that are modified within a procedure.

**Question 12: (True/False): The USES operator only generates PUSH instructions, so you must code POP instructions yourself.**

Answer 12: True. The USES operator generates PUSH instructions, so you need to code the corresponding POP instructions yourself.

corresponding POP instructions yourself.

**Question 13: (True/False):** The register list in the USES directive must use commas to separate the register names.

Answer 13: True. The register list in the USES directive must use commas to separate the register names.

**Question 14:** Which statement(s) in the ArraySum procedure (Section 5.2.5) would have to be modified so it could accumulate an array of 16-bit words? Create such a version of ArraySum and test it.

Answer 14: To accumulate an array of 16-bit words, you would need to modify the mov eax, [esi] and add esi, 4 statements to work with 16-bit words, like this:

```
mov ax, [esi]
add esi, 2
```

**Question 15:** What will be the final value in EAX after these instructions execute? push 5 push 6 pop eax pop eax

Answer 15: EAX will equal 5 after these instructions execute. The second pop eax instruction will overwrite the previous value of EAX.

**Question 16:** Which statement is true about what will happen when the example code runs?

```
1: main PROC  
2: push 10  
3: push 20  
4: call Ex2Sub  
5: pop eax  
6: INVOKE ExitProcess,0  
7: main ENDP
```

```
10: Ex2Sub PROC  
11: pop eax  
12: ret  
13: Ex2Sub ENDP
```

Answer 16: a. EAX will equal 10 on line 6.

Question 17: Which statement is true about what will happen when the example code runs?

```
1: main PROC  
2: mov eax,30  
3: push eax  
4: push 40  
5: call Ex3Sub  
6: INVOKE ExitProcess,0  
7: main ENDP
```

```
10: Ex3Sub PROC  
11: pusha  
12: mov eax,80  
13: popa  
14: ret  
15: Ex3Sub ENDP
```

Answer 17: d. The program will halt with a runtime error on Line 11 because there's no matching pop for the pusha instruction.

Question 18: Which statement is true about what will happen when the example code runs?

```
1: main PROC  
2: mov eax,40  
3: push offset Here  
4: jmp Ex4Sub  
5: Here:  
6: mov eax,30  
7: INVOKE ExitProcess,0  
8: main ENDP  
  
10: Ex4Sub PROC  
11: ret  
12: Ex4Sub ENDP
```

Answer 18: b. The program will halt with a runtime error on Line 4 because there's no matching pop for the push instruction.

Question 19: Which statement is true about what will happen when the example code runs?

```
1: main PROC  
2: mov edx,0  
3: mov eax,40  
4: push eax  
5: call Ex5Sub  
6: INVOKE ExitProcess,0  
7: main ENDP
```

```
10: Ex5Sub PROC  
11: pop eax  
12: pop edx  
13: push eax  
14: ret  
15: Ex5Sub ENDP
```

Answer 19: a. EDX will equal 40 on line 6.

Question 20: What values will be written to the array when the following code executes?

```
592 .data  
593 array DWORD 4 DUP(0)  
594  
595 .code
```

In the provided code, you've declared an array named array with four double word (DWORD) elements, and you've initialized each element to 0 using 4 DUP(0). Therefore, when this code is executed, the array will contain the following values:

array[0] will be 0.  
array[1] will be 0.  
array[2] will be 0.  
array[3] will be 0.

So, all elements in the array will have the value 0.

## ***Practice God Level Questions 2***

### ***Exercise 1: Draw Text Colors***

```
598 include Irvine32.inc
599
600 .data
601     colors BYTE 2, 4, 6, 14    ; Colors: Green, Red, Yellow, White
602     message BYTE "Hello, Colors!",0
603
604 .code
605 main PROC
606     mov ecx, 4    ; Number of colors
607     mov esi, 0    ; Index for colors array
608
609 loop_colors:
610     mov eax, colors[esi]
611     call SetTextColor
612     mov edx, OFFSET message
613     call WriteString
614     call Crlf
615
616     inc esi
617     loop loop_colors
618
619     call WaitMsg  ; Wait for a key press
620     call Clrscr   ; Clear the screen
621     call ExitProcess
622 main ENDP
623 END main
```

## *Exercise 2: Linking Array Items*

```
.data
    start DWORD 1
    chars BYTE 'H', 'A', 'C', 'E', 'B', 'D', 'F', 'G'
    links DWORD 0, 4, 5, 6, 2, 3, 7, 0
    outputArray BYTE 8 DUP(?) ; To store the characters in order
.code
main PROC
    mov edi, OFFSET outputArray ; Destination for output characters
    mov esi, start ; Start index
    mov ecx, 8 ; Number of characters to locate
    traverse_links:
        mov al, chars[esi] ; Load character
        mov [edi], al ; Store it in outputArray
        inc edi ; Move to the next position in outputArray
        ; Get the next link index
        mov eax, esi
        mov ebx, 4 ; Size of DWORD (4 bytes)
        mul ebx ; Multiply esi by 4
        mov esi, links[eax] ; Get the next link index

        loop traverse_links ; Repeat for all characters

    ; Display the characters in outputArray
    mov edx, OFFSET outputArray
    call WriteString
    call Crlf

    call WaitMsg ; Wait for a key press
    call ExitProcess
main ENDP
END main
```

*Exercise 3: Simple Addition (1)*

```
662 include Irvine32.inc
663 .data
664     prompt1 BYTE "Enter the first integer: ", 0
665     prompt2 BYTE "Enter the second integer: ", 0
666     resultMsg BYTE "The sum is: ", 0
667     buffer1 DWORD ?
668     buffer2 DWORD ?
669     result DWORD ?
670 .code
671     main PROC
672         call Clrscr
673         mov edx, OFFSET prompt1
674         call WriteString
675         call ReadInt
676         mov buffer1, eax
677
678         mov edx, OFFSET prompt2
679         call WriteString
680         call ReadInt
681         mov buffer2, eax
682
683         ; Add the integers
684         mov eax, buffer1
685         add eax, buffer2
686         mov result, eax
687
688         mov edx, OFFSET resultMsg
689         call WriteString
690         mov eax, result
691         call WriteInt
692
693         call WaitMsg ; Wait for a key press
694         call ExitProcess
695     main ENDP
696 END main
```

*Exercise 4: Simple Addition (2)*

```
700 include Irvine32.inc
701 .data
702     prompt1 BYTE "Enter the first integer: ", 0
703     prompt2 BYTE "Enter the second integer: ", 0
704     resultMsg BYTE "The sum is: ", 0
705     buffer1 DWORD ?
706     buffer2 DWORD ?
707     result DWORD ?
708
709 .code
710     main PROC
711         mov ecx, 3 ; Repeat the process three times
712         loop_repeat:
713             call Clrscr
714
715             mov edx, OFFSET prompt1
716             call WriteString
717             call ReadInt
718             mov buffer1, eax
719
720             mov edx, OFFSET prompt2
721             call WriteString
722             call ReadInt
723             mov buffer2, eax
724
725             ; Add the integers
726             mov eax, buffer1
727             add eax, buffer2
728             mov result, eax
```

```
729
730     mov edx, OFFSET resultMsg
731     call WriteString
732     mov eax, result
733     call WriteInt
734
735     call WaitMsg ; Wait for a key press
736
737     dec ecx
738     jnz loop_repeat ; Repeat the process three times
739
740     call ExitProcess
741 main ENDP
742 END main
```

*Exercise 5: BetterRandomRange Procedure*

```
746 include Irvine32.inc
747
748 .data
749
750 .code
751 BetterRandomRange PROC
752     ; Input: EBX = Lower bound (M)
753     ;         EAX = Upper bound (N)
754     ; Output: EAX = Random number between M and N-1
755
756     sub eax, ebx    ; Calculate the range (N - M)
757     add eax, 1      ; Include the upper bound itself
758     call RandomRange
759     add eax, ebx    ; Offset the result by M (lower bound)
760     ret
761 BetterRandomRange ENDP
762
763 main PROC
764     mov ecx, 50    ; Repeat 50 times
765     loop_repeat:
766         call Clrscr
767
768         mov ebx, -300 ; Lower bound
769         mov eax, 100   ; Upper bound
770         call BetterRandomRange
771
772         ; Display the randomly generated value
773         mov edx, eax
774         call WriteInt
775         call Crlf
```

```
776  
777     call WaitMsg ; Wait for a key press  
778     dec ecx  
779     jnz loop_repeat ; Repeat the process 50 times  
780  
781     call ExitProcess  
782 main ENDP  
783 END main
```

These solutions demonstrate the requested functionality for each exercise. Feel free to modify them, as you do your practice to be a better programmer.

### ***Exercise 1: Draw Text Colors Part 2***

Write a program that displays the same string in four different colors, using a loop. Call the SetTextColor procedure from the book's link library. Any colors may be chosen, but you may find it easiest to change the foreground color.

```
INCLUDE Irvine32.inc

.data
message DB "Colorful Text",0

.code
main PROC
    mov ecx, 4 ; Number of times to display the text in different colors
    mov esi, 1 ; Color index
```

```
colorLoop:  
    call SetTextColor  
    mov edx, OFFSET message  
    call WriteString  
    call Crlf  
  
    inc esi ; Move to the next color  
    loop colorLoop  
  
    exit  
main ENDP  
  
SetTextColor PROC  
    ; Set the text color based on the value in ESI  
    mov eax, esi  
    switch eax  
        case 1  
            call SetTextColorRed  
        case 2  
            call SetTextColorBlue  
        case 3  
            call SetTextColorGreen  
        case 4  
            call SetTextColorYellow  
    endSwitch  
    ret  
SetTextColor ENDP  
  
SetTextColorRed PROC  
    mov eax, Red  
    call SetTextColor  
    ret  
SetTextColorRed ENDP  
  
SetTextColorBlue PROC  
    mov eax, Blue
```

```
call SetTextColor
ret
SetTextColorBlue ENDP

SetTextColorGreen PROC
    mov eax, Green
    call SetTextColor
    ret
SetTextColorGreen ENDP

SetTextColorYellow PROC
    mov eax, Yellow
    call SetTextColor
    ret
SetTextColorYellow ENDP

Yellow = 14
Green = 10
Blue = 9
Red = 12

END main
```

## ***CONDITIONAL PROCESSING***

Skip...

***AND Operation***

The Boolean instructions in assembly language are used to perform logical operations on bits or bytes. These instructions are typically used to manipulate data, make decisions, and control the flow of a program.

## AND

The AND instruction performs a bit-by-bit logical AND operation on two operands. The result is stored in the destination operand. The AND operation returns a 1 if both bits are 1, otherwise it returns a 0.

## OR

The OR instruction performs a bit-by-bit logical OR operation on two operands. The result is stored in the destination operand. The OR operation returns a 1 if either bit is 1, otherwise it returns a 0.

## XOR

The XOR instruction performs a bit-by-bit logical exclusive OR operation on two operands. The result is stored in the destination operand. The XOR operation returns a 1 if only one bit is 1, otherwise it returns a 0.

## NOT

The NOT instruction performs a bit-by-bit logical NOT operation on a single operand. The result is stored in the destination operand. The NOT operation inverts all of the bits in the operand.

## TEST

The TEST instruction performs a bit-by-bit logical AND operation on two operands, but does not store the result. Instead, the TEST instruction sets the CPU flags according to the result of the operation. The TEST instruction is often used to check the value of a register or memory location before making a decision.

Sr.No.	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

Here are some examples of how to use the Boolean instructions in assembly language:

```
01 ; AND two registers
02 mov eax, 10h
03 mov ebx, 5h
04 and eax, ebx ; eax = 4h
05
06 ; OR two registers
07 mov eax, 10h
08 mov ebx, 5h
09 or eax, ebx ; eax = 15h
10
11 ; XOR two registers
12 mov eax, 10h
13 mov ebx, 5h
14 xor eax, ebx ; eax = 11h
15
16 ; NOT a register
17 mov eax, 10h
18 not eax ; eax = 11111110h
19
20 ; TEST two registers
21 mov eax, 10h
22 mov ebx, 5h
23 test eax, ebx ; CF flag is set to 0
24
25 ; TEST a register against a value
26 mov eax, 10h
27 test eax, 0Fh ; CF flag is set to 1
```

**Zero Flag (ZF):**

Set when the result of an operation is zero.

Used for testing equality or non-equality.

Example: Used to skip instructions if the result is zero.

**Carry Flag (CF):**

Set when an operation generates a carry out of the highest bit.

Commonly used in arithmetic operations like addition and subtraction.

Example: Indicates an overflow in addition if set.

**Sign Flag (SF):**

Set when the most significant bit of the result is set.

Indicates whether the result is positive or negative.

Example: Set if the result is negative.

**Overflow Flag (OF):**

Set when an operation generates a result outside the signed range.

Used to detect arithmetic errors.

Example: Set when adding two positive numbers results in a negative value.

**Parity Flag (PF):**

Set when the destination operand has an even number of 1 bits.

Used for data error checking.

Example: Can indicate data corruption if not set when reading from memory.

=====

## **AND Instruction:**

---

Operation: The AND instruction performs a bitwise AND operation between each pair of matching bits in two operands and stores the result in the destination operand.

Syntax: AND destination, source

Operand Combinations:

```
30 AND reg, reg  
31 AND reg, mem  
32 AND reg, imm  
33 AND mem, reg  
34 AND mem, imm
```

**Operand Sizes:** The operands can be 8, 16, 32, or 64 bits, and they must be the same size.

If both bits equal 1, the result bit is 1; otherwise, it is 0.

Example:  $x \text{ AND } y$ , where  $x$  and  $y$  are bits.

<b>x</b>	<b>y</b>	<b><math>x \wedge y</math></b>
0	0	0
0	1	0
1	0	0
1	1	1

**Bit Masking:** The AND instruction is commonly used for bit masking. It allows you to clear specific bits in an operand without affecting other bits. For example, if you want to reset a hardware device by clearing bits 0 and 3 in the AL register without modifying other bits, you can use:

**and AL, 11110110b ; Clear bits 0 and 3, leave others unchanged**

**Flags:** The AND instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags based on the value assigned to the destination operand.

---

Yes, the AND instruction can be used to clear selected bits in an operand while preserving the remaining bits. This is called **masking**.

To mask out a bit, you AND the operand with a mask that has a 0 in the bit position that you want to clear. For example, to mask out the least significant bit of the AL register, you would AND it with the mask 0b11111110.

Here is an example of how to use the AND instruction to mask out the least significant bit of the AL register:

```
; Mask out the least significant bit of the AL register.  
  
mov al, 0b00111011b ; AL = 00111011b  
and al, 0b11111110b ; AL = 00111010b  
  
; The least significant bit of the AL register is now cleared.
```

You can use the same technique to mask out any bit in an operand, regardless of its position. To mask out bit n, simply AND the operand with a mask that has a 0 in the bit position n.

Masking is a very useful technique in assembly language programming. It can be used to perform a variety of tasks, such as:

- Isolating a specific bit or bits in an operand.
- Clearing or setting a specific bit or bits in an operand.
- Checking if a specific bit or bits in an operand are set or cleared
- Performing logical AND operations on multiple operands.

The assembly code below is a simple way to convert a character from lowercase to uppercase. It works by clearing bit 5 of the character, which is the bit that determines whether the character is lowercase or uppercase.

Here is a breakdown of the code:

```
.data
    array BYTE 50 DUP(?)  
  
.code
    mov ecx, LENGTHOF array
    mov esi, OFFSET array
    L1:
    and byte ptr [esi], 11011111b
    inc esi
    loop L1
```

- The .data section declares an array of 50 bytes, each of which can store a single character.
- The .code section contains the actual code to convert the characters in the array to uppercase.
- The mov ecx, LENGTHOF array instruction loads the length of the array into register ecx.
- The mov esi, OFFSET array instruction loads the address of the first element in the array into register esi.
- The L1: label marks the beginning of a loop. The loop will iterate LENGTHOF array times, once for each character in the array.
- The and byte ptr [esi], 11011111b instruction ANDs the byte at the address stored in register esi with the value 11011111b. This clears bit 5 of the byte, which converts the character to uppercase.
- The inc esi instruction increments register esi by 1. This moves the pointer to the next element in the array.
- The loop L1 instruction loops back to the beginning of the loop. This will continue until the loop has iterated LENGTHOF array times.

- After the loop has finished executing, all of the characters in the array will have been converted to uppercase.

## ***BitMapped Sets***

The passage you provided explains bit-mapped sets and how they can be used to represent sets of items efficiently using binary bits. Here's a breakdown of the key concepts discussed:

**Bit-Mapped Sets:** Bit-mapped sets are used to represent sets of items from a limited-sized universal set. Instead of using pointers or references, a bit vector (or bit map) is used to map binary bits to an array of objects. Each bit position in the binary number corresponds to a specific element in the set.

**Checking Set Membership:** To check if a particular element is a member of a set, you can use the **AND instruction**. By AND-ing the bit at the element's position with 1, you can determine membership. For example, `mov eax,SetX` and `eax,10000b` checks if `element[4]` is a member of SetX. If the Zero flag is cleared after the operation, it means the element is a member.

**Set Complement:** The complement of a set can be generated using the **NOT instruction**, which reverses all the bits. This can be useful for operations involving set differences or negation.

**Set Intersection:** The AND instruction is used to generate a bit vector representing the intersection of two sets. By AND-ing the bit vectors of the sets, you get a result that shows **which elements are common to both sets**.

**Set Union:** The **OR instruction** is used to produce a bit map representing the union of two sets. By OR-ing the bit vectors of the sets, you get a result that includes all elements present in either set.

This low-level bit manipulation is important in systems programming, especially when dealing with hardware or memory management. It allows for efficient representation and manipulation of sets without the need for complex data structures.

Certainly, I can provide code examples in MASM (Microsoft Assembler) for 32-bit architecture to demonstrate the concepts you mentioned. Here's the code for checking set membership, set complement, set intersection, and set union:

***Checking Set Membership (Example for element[4] in SetX):***

```
119 .data
120 SetX DWORD 10000000h ; SetX represented as a 32-bit value
121
122 .code
123 main PROC
124     mov eax, SetX      ; Load SetX into EAX
125     and eax, 00000010b ; Check if element[4] is a member
126     cmp eax, 0          ; Compare the result with 0
127     jz not_a_member    ; Jump if Zero flag is set (not a member)
128     ; If Zero flag is not set, element[4] is a member of SetX
129     ; Your code here for member case
130     jmp done
131
132 not_a_member:
133     ; Your code here for not a member case
134
135 done:
136     ; Exit your program
137     invoke ExitProcess, 0
138
139 main ENDP
```

*Set Complement:*

```
145 .data
146 SetX DWORD 10000000h ; SetX represented as a 32-bit value
147
148 .code
149 main PROC
150     mov eax, SetX      ; Load SetX into EAX
151     not eax           ; Complement SetX
152     ; EAX now contains the complement of SetX
153     ; Your code here to work with the complemented set
154     ; Exit your program
155     invoke ExitProcess, 0
156
157 main ENDP
```

*Set Intersection (Example of SetX and SetY):*

```
162 .data
163 SetX DWORD 10000000000000000000000000000000111h
164 SetY DWORD 100000101010000000011101100011h
165
166 .code
167 main PROC
168     mov eax, SetX      ; Load SetX into EAX
169     and eax, SetY      ; Calculate the intersection of SetX and SetY
170     ; EAX now contains the intersection
171     ; Your code here to work with the intersection
172     ; Exit your program
173     invoke ExitProcess, 0
174
175 main ENDP
```

*Set Union (Example of SetX and SetY):*

```
.data
SetX DWORD 10000000000000000000000000000000111h
SetY DWORD 100000101010000000011101100011h

.code
main PROC
    mov eax, SetX      ; Load SetX into EAX
    or eax, SetY       ; Calculate the union of SetX and SetY
    ; EAX now contains the union
    ; Your code here to work with the union
    ; Exit your program
    invoke ExitProcess, 0

main ENDP
```

## ***OR Operation***

---

### ***Or Instruction:***

---

**Operation:** The OR instruction performs a bitwise OR operation between each pair of matching bits in two operands and stores the result in the destination operand.

```
51 OR reg, reg  
52 OR reg, mem  
53 OR reg, imm  
54 OR mem, reg  
55 OR mem, imm
```

**Combinations above:** Same as the AND instruction.

**Operand Sizes:** The operands can be 8, 16, 32, or 64 bits, and they must be the same size.

**Truth Table:** For each matching bit in the two operands: The output bit is 1 when at least one of the input bits is 1.

Example:  $x \text{ OR } y$ , where  $x$  and  $y$  are bits.

<b>x</b>	<b>y</b>	<b><math>x \vee y</math></b>
0	0	0
0	1	1
1	0	1
1	1	1

**Setting Bits:** The OR instruction is useful when you need to set one or more bits in an operand

without affecting other bits. This is a common technique in microcontroller and embedded systems programming to manipulate specific control bits in registers while preserving the rest of the configuration. For example, you can set bit 2 in the AL register with:

```
or AL, 00000100b ; Set bit 2, leave others unchanged
```

or

```
OR AL, 1 << 2
```

The OR instruction performs a bitwise OR operation on its two operands. The << operator shifts the number on its left by the number of bits specified by the number on its right.

In this case, the number 1 is shifted left by 2 bits, which results in the number 4. The OR instruction then ORs the AL register with the number 4, which sets the bit in position 2 of the AL register.

Here is an example of how to use the code above:

```
61 ; Set the bit in position 2 of the control byte in the AL register.  
62  
63 mov al, 0b00111010 ; AL = 00111010  
64 OR AL, 1 << 2 ; AL = 00111110  
65  
66 ; The bit in position 2 of the AL register is now set.
```

You can use the same technique to set any bit in an operand, regardless of its position. To set bit n, simply OR the operand with the number 1 shifted left by n bits.

---

The instruction OR AL, 1 << 2 is a concise way of setting bit 2 (counting from the least significant bit) in the AL register without altering the other bits.

It uses the left shift (<<) operation to create the bitmask 00000100b, and then it performs a bitwise OR operation with the contents of AL.

Here's the breakdown:

**1 << 2** shifts the binary value 00000001 two positions to the left, resulting in 00000100. This creates a bitmask with only bit 2 set to 1. OR AL, 00000100b performs a bitwise OR operation between the contents of AL and the bitmask.

This sets bit 2 in AL to 1 while leaving the other bits unchanged. So, after executing this instruction, bit 2 in the AL register will be set to 1, and the rest of the bits will remain as they were.

---

**Flags:** The OR instruction always clears the Carry and Overflow flags. It modifies the Sign, Zero, and Parity flags based on the value assigned to the destination operand.

These instructions are essential for performing bitwise operations in assembly language, and they are often used for tasks like bit manipulation and flag setting/clearing. If you have any specific questions or would like further examples, feel free to ask!

**SetX:**

**10000000000000000000000000000000111**

**SetY:**

**100000101010000000011101100011**

**Union (SetX OR SetY):**

**100000101010000000011101100111**

It's clear that the OR operation combines the two sets by preserving any bits that are set in either SetX or SetY.

In binary representation:

- SetX has bits set at positions 0, 1, and 31.
- SetY has bits set at positions 0, 5, 9, 14, 18, 23, 26, 30, and 31.

When you perform a bitwise OR between SetX and SetY, the resulting union has bits set at all the positions where at least one of SetX or SetY had a bit set. In this case, the union contains bits set at positions 0, 1, 5, 9, 14, 18, 23, 26, 30, and 31.

This operation can be visualized as a union operation in set theory, where you're combining the elements of two sets while eliminating duplicates.

## ***XOR Operation***

---

### ***XOr Instruction:***

---

The XOR (exclusive OR) instruction performs a boolean exclusive-OR operation between corresponding bits in two operands and stores the result in the destination operand.

The XOR operation follows these rules:

If both bits are the same (both 0 or both 1), the result is 0. If the bits are different (one 0 and one 1), the result is 1. A bit XORED with 0 retains its value, and a bit XORED with 1 toggles (complements) its value.

XOR is reversible, meaning applying it twice to the same operand reverts it to the original value. Here's a truth table for XOR ( $x \oplus y$ ):

## XOR Operation

<u>x</u>	<u>y</u>	<u><math>x \oplus y</math></u>
0	0	0
0	1	1
1	0	1
1	1	0

The XOR instruction performs a bitwise exclusive OR operation on its two operands.

`xor destination_operand, source_operand`

The XOR instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags according to the value assigned to the destination operand.

**Parity checking** is a method used to determine whether a binary number has even or odd parity. It counts the number of 1 bits in the number, and if the count is even, it's considered even parity, and if it's odd, it's considered odd parity.

In x86 processors, the **Parity flag (PF)** is set when the lowest byte (8 bits) of the destination operand of a bitwise or arithmetic operation has even parity. If the lowest byte has odd parity, the PF is cleared.

Here's an example of how to check parity without changing the value of a byte in assembly:

```
199 mov al, 10110101b ; 5 bits = odd parity
200 xor al, 0          ; Parity flag clear (odd)
201
202 mov al, 11001100b ; 4 bits = even parity
203 xor al, 0          ; Parity flag set (even)
```

The XOR instruction can be used to toggle (invert) bits, check the parity of a number, and perform other bitwise operations. Here are some examples of how to use the XOR instruction:

```
207 mov ax, 64C1h ; 0110 0100 1100 0001
208 xor ah, al      ; Parity flag set (even)
```

To calculate parity for 32-bit values, you can XOR all the bytes together, like this:

```
212 B0 XOR B1 XOR B2 XOR B3
```

## ***NOT Operation***

The **NOT instruction** is used to invert or toggle all the bits in an operand. This operation is also known as taking the one's complement of the operand. Here's how it works:

**NOT reg:** This form of the NOT instruction operates on a register. It inverts all the bits in the specified register.

```
215 mov al, 11110000b ; AL = 11110000b (F0h in hexadecimal)
216 not al             ; AL = 00001111b (0Fh in hexadecimal)
```

**NOT mem:** This form of the NOT instruction operates on a memory location. It inverts all the bits in the value stored at that memory location.

```
218 mov byte ptr [ebx], 10101010b ; Store 10101010b at the memory location pointed to by EBX  
219 not byte ptr [ebx]           ; Invert the bits at that memory location
```

In both examples, the NOT instruction flips all the bits. In the first example, it's applied to the AL register, and in the second example, it's applied to a byte stored in memory via the EBX register.

Flags: The NOT instruction does not affect any of the CPU flags. It simply performs the bitwise inversion without changing the status flags like Zero Flag, Sign Flag, etc.

## ***TEST Operation***

The TEST instruction in assembly language, particularly in x86 assembly, is a useful instruction for performing bitwise logical operations without changing the contents of the destination operand.

The test operation in assembly language is used to test the value of a register or memory location against a specific value. It does this by performing a logical AND operation between the two values and setting the flags accordingly.

It's often used to check the status of specific bits within a data value. Let me break down the information you provided and explain it in more detail.

### **What TEST Instruction Does:**

The TEST instruction performs a bitwise AND operation between two operands, setting the Sign (S), Zero (Z), and Parity (P) flags based on the result of the operation.

It is similar to the AND instruction but does not modify the destination operand.

This makes it valuable for checking whether specific bits are set without altering the original data.

### Operand Combinations:

The TEST instruction allows you to use the same operand combinations as the AND instruction.

You can perform bitwise AND operations between registers, memory locations, or immediate values (constants).

### Example: Testing Multiple Bits:

In the example you provided, the goal is to determine whether bit 0 or bit 3 is set in the AL register. The following instruction accomplishes this:

```
222 test al, 00001001b ; test bits 0 and 3
```

Here, al represents the AL register, and 00001001b is a bit mask. The bit mask is used to specify which bits you want to test. In this case, it's checking bits 0 and 3.

**Flag Effects:** The result of the TEST instruction affects the Sign (S), Zero (Z), and Parity (P) flags. The Zero flag (ZF) is particularly useful in determining whether the tested bits are all clear. If ZF is set (1), it means all tested bits are clear if it's clear (0), at least one of the

tested bits is set.

**Flag Modifications:** The TEST instruction always clears the Overflow (OF) and Carry (CF) flags. It modifies the Sign (S), Zero (Z), and Parity (P) flags in the same way as the AND instruction.

In summary, the TEST instruction in assembly language allows you to perform bitwise AND operations between operands without altering the destination operand.

It's a valuable tool for checking the status of specific bits within data, and the result of the operation affects certain processor flags, such as the Zero flag (ZF), which is often used for conditional branching in assembly code.

Example of using a bit mask with the TEST instruction

(The value 00001001 in this example is called a bit mask.)

```
251 Input value: 0 0 1 0 0 1 0 1
252 Test value: 0 0 0 0 1 0 0 1
253 Result: 0 0 0 0 0 0 0 1
254 Zero Flag (ZF) = 0 (Not set)
255
256
257 Input value: 0 0 1 0 0 1 0 0
258 Test value: 0 0 0 0 1 0 0 1
259 Result: 0 0 0 0 0 0 0 0
260 Zero Flag (ZF) = 1 (Set)
```

**Flags:** The TEST instruction always clears the Overflow (OF) and Carry (CF) flags.

It modifies the Sign (SF), Zero (ZF), and Parity (PF) flags in the same way as the AND instruction.

In the first example, the Zero Flag (ZF) is not set ( $ZF = 0$ ) because at least one of the tested bits is set.

In the second example, the Zero Flag (ZF) is set ( $ZF = 1$ ) because all tested bits are clear (resulting in all zeros).

The TEST instruction is often used for bitwise checks like these to determine whether specific bits are set in a data value.

It is a common practice in assembly language programming to use bit masks to isolate and check individual bits within a data value.

The ZF flag is useful for conditional branching in assembly code. If ZF is set, it can indicate that certain conditions are met.

---

The above image:

This shows that the test operation is used to test the value of the register or memory location 0x100 against the value 0x09. The first test results in a zero flag of 0, which means that the two values are not equal. The second test results in a zero flag of 1, which means that the two values are equal.

The test operation works by performing a logical AND operation between the two values. The logical

AND operation returns a 1 bit only if both bits in the corresponding positions are 1 bits. Otherwise, it returns a 0 bit.

The result of the test operation is stored in the zero flag (ZF). If the result of the AND operation is 0, then the zero flag is set to 1. Otherwise, the zero flag is set to 0.

The zero flag can then be used to determine whether the two values are equal. If the zero flag is set to 1, then the two values are equal. Otherwise, the two values are not equal.

The test operation is a powerful tool that can be used to implement a variety of different algorithms and programs. For example, the test operation can be used to:

Test the value of a register or memory location before performing a conditional jump. Test the value of a register or memory location to determine which branch of a decision tree to take.

Test the value of a register or memory location to determine whether a specific condition has been met.

In the example you sent, the test operation is being used to test the value of the register or memory location 0x100 against the value 0x09.

If the two values are equal, then the program will jump to the label "result". If the two values are not equal, then the program will continue executing the next instruction.

Here is a simple example of how to use the test operation in assembly language:

```
263 ; Test the value of the EAX register against the value 10.  
264 ; If the two values are equal, then jump to the label "result".  
265 test eax, 10  
266 je result  
267  
268 ; The two values are not equal, so continue executing the next instruction.
```

## **CMP Instruction**

The CMP (compare) instruction is indeed an essential part of x86 assembly language when it comes to comparing integers, including character codes.

comparing integers, including character codes.

It's used to perform an implied subtraction of a source operand from a destination operand, without modifying either operand.

The result of this operation affects various flags in the CPU, which can then be used for conditional branching, making it a crucial component in creating conditional logic structures.

The **CMP instruction** performs an implied subtraction of the source operand from the destination operand. However, the actual subtraction is not performed. Instead, the status flags are set according to the result of the subtraction.

Here's a breakdown of how the CMP instruction affects flags based on the comparison results:

CMP Results	ZF	CF	Unsigned Operands	Signed Operands
Destination < source	0	1	Carry flag is set	Sign flag and Carry flag are set
Destination > source	0	0	Carry flag is not set	Sign flag and Carry flag are not set
Destination = source	1	0	Carry flag is not set	Sign flag is not set

### ***Unsigned Operands. When comparing two unsigned operands:***

If the destination is less than the source, the Zero Flag (ZF) is set to 0, and the Carry Flag (CF) is set to 1. If the destination is greater than the source, ZF is set to 0, and CF is set to 0. If the destination equals the source, ZF is set to 1, and CF is set to 0.

## *Signed Operands. When comparing two signed operands:*

If the destination is less than the source, the Sign Flag (SF) is not equal to the Overflow Flag (OF). If the destination is greater than the source, SF is equal to OF. If the destination equals the source, the Zero Flag (ZF) is set to 1.

```
271 mov ax, 5  
272 cmp ax, 10  
273 ; ZF = 0 and CF = 1
```

In this example, when AX (with a value of 5) is compared to 10, the CMP instruction sets the Zero Flag (ZF) to 0 because 5 is not equal to 10. The Carry Flag (CF) is set to 1 because subtracting 10 from 5 would require a borrow. Example 2:

```
276 mov ax, 1000  
277 mov cx, 1000  
278 cmp cx, ax  
279 ; ZF = 1 and CF = 0
```

Here, when AX and CX both contain 1000, the CMP instruction sets the Zero Flag (ZF) to 1 because subtracting one 1000 from the other results in zero. The Carry Flag (CF) is set to 0 because no borrow is required. Example 3:

```
282 mov si, 105  
283 cmp si, 0  
284 ; ZF = 0 and CF = 0
```

In this case, when SI (with a value of 105) is compared to 0, the CMP instruction sets both the Zero Flag (ZF) and the Carry Flag (CF) to 0 because subtracting 0 from 105 generates a positive, nonzero value.

CMP, when used in conjunction with conditional jump instructions, allows you to create conditional logic structures, akin to high-level programming languages' IF statements, in assembly language. It's a powerful tool for controlling the flow of your programs based on comparisons between values in registers or memory locations.

## ***Setting and Clearing Flags***

It seems you've provided a portion of assembly language code related to setting and clearing individual CPU flags, as well as conditional jumps. I'll explain this code in detail. Setting and Clearing Individual CPU Flags.

### ***Setting the Zero Flag***

To set the Zero flag, you can use the TEST or AND instruction. In the code:

287 test al, 0 ; set Zero flag

This instruction tests the value in the al register against 0. If the result is zero, the Zero flag is set.

### ***Clearing the Zero Flag***

To clear the Zero flag, you can use the OR instruction with 1:

```
290 or al, 1 ; clear Zero flag
```

This instruction logically ORs the al register with 1, ensuring that the Zero flag is cleared.

### ***Setting the Sign Flag***

To set the Sign flag, you can use the OR instruction with the highest bit of an operand (bit 7 in the al register) set to 1:

```
296 or al, 80h ; set Sign flag
```

This operation sets the highest bit of al to 1, which sets the Sign flag.

### ***Clearing the Sign Flag***

To clear the Sign flag, you can use the AND instruction with the highest bit (bit 7) set to 0:

```
299 and al, 7Fh ; clear Sign flag
```

This operation clears the highest bit of al, ensuring that the Sign flag is cleared.

### ***Setting the Carry Flag***

To set the Carry flag, you can use the STC (Set Carry) instruction:

306 `stc ; set Carry flag`

This instruction sets the Carry flag, indicating a carry condition.

### *Clearing the Carry Flag*

To clear the Carry flag, you can use the CLC (Clear Carry) instruction:

310 `clc ; clear Carry flag`

This instruction clears the Carry flag, indicating no carry condition.

### *Setting the Overflow Flag*

To set the Overflow flag, you can add two positive values that produce a negative sum. This condition naturally sets the Overflow flag.

### *Clearing the Overflow Flag*

To clear the Overflow flag, you can use the OR instruction with an operand of 0:

313 `or eax, 0 ; clear Overflow flag`

This operation performs a logical OR with 0, ensuring that the Overflow flag is cleared.

The provided code also mentions the relationship between flags (SF, OF, ZF) and the results of

comparisons and arithmetic operations.

It's crucial to understand these flag behaviors when working with assembly language programming.

## Booleans & 64-Bit Mode

In 64-bit mode, instructions work similarly to how they do in 32-bit mode, but with some differences due to the larger register size.

**Operand Size:** When you operate on 64-bit registers or memory operands with a source operand that's smaller than 32 bits, all bits in the destination operand are affected eg.

```
316 mov rax, allones    ; RAX = FFFFFFFFFFFFFF  
317 and rax, 80h        ; RAX = 0000000000000080
```

Here, the and operation affects all 64 bits of RAX.

**32-Bit Operand:** However, when you use a 32-bit constant or register as the source operand, only the lower 32 bits of the destination operand are modified. For instance:

```
320 mov rax, allones    ; RAX = FFFFFFFFFFFFFF  
321 and rax, 80808080h  ; RAX = FFFFFFFF80808080
```

In this case, only the lower 32 bits of RAX are changed by the and operation.

**Memory Operand:** The same rules apply when the destination operand is in memory, not just in registers.

**Special Handling for 32-Bit Operands:** You need to be careful when dealing with 32-bit operands because they behave differently from other operand sizes in 64-bit mode.

Understanding these distinctions is crucial when writing assembly code for 64-bit systems.

=====

## Questions:

=====

**Question:** How can you clear the high 8 bits of AX without changing the low 8 bits using a single 16-bit operand instruction?

**Answer:** You can clear the high 8 bits of AX by using the AND instruction with the 16-bit mask 00FFh. The instruction would look like and ax, 00FFh.

**and ax, 00FFh**

**Question:** How can you set the high 8 bits of AX without changing the low 8 bits using a single 16-

**bit operand instruction?**

Answer: You can set the high 8 bits of AX by using the OR instruction with a 16-bit value.

**or ax, FF00h**

**Question: What instruction can you use to reverse all the bits in EAX with a single instruction?**

Answer: To reverse all the bits in EAX, you can use the XOR instruction with a mask where all bits are set to FFFFFFFFh. The instruction would be xor eax, FFFFFFFFh.

**xor eax, FFFFFFFFh**

**Question: How can you set the Zero flag if the 32-bit value in EAX is even and clear the Zero flag if EAX is odd?**

Answer: You can set the Zero flag if the 32-bit value in EAX is even and clear the Zero flag if EAX is odd using the TEST instruction and conditional jumps. Here's an example:

```
test eax, 1      ; Test if the least significant bit is set
jz even_label    ; Jump if Zero flag is set (EAX is even)
; Code for odd case here
even_label:
; Code for even case here
```

Question: How can you convert an uppercase character in AL to lowercase using a single instruction, but without modifying AL if it's already lowercase?

Answer: To convert an uppercase character in AL to lowercase without modifying it if it's already lowercase, you can use conditional instructions like this:

```
343 cmp al, 'A'      ; Compare AL with 'A'
344 jl not_uppercase ; Jump if AL is less than 'A' (not uppercase)
345 cmp al, 'Z'      ; Compare AL with 'Z'
346 jg not_uppercase ; Jump if AL is greater than 'Z' (not uppercase)
347 add al, 32        ; Convert uppercase to lowercase ('A'- 'a' = 32)
348 not_uppercase:
349 ; Continue with your code here
```

This code first checks if AL is between 'A' and 'Z' (inclusive) using CMP and conditional jumps (JL and JG). If it's within that range, it adds 32 to AL, converting the uppercase letter to lowercase.

## *Conditional Jumps*

---

### *Conditional Structures in x86*

---

x86 does not have explicit high-level logic structures in its instruction set, but you can implement them using a combination of comparisons and jumps. Two steps are involved in executing a conditional statement:

An operation such as CMP, AND, or SUB modifies the CPU status flags. A conditional jump instruction tests the flags and causes a branch to a new address.

The following example compares EAX to zero. The **JZ (Jump if zero) instruction** jumps to label L1 if the Zero flag was set by the CMP instruction:

```
08 cmp eax, 0      ; Compare the value in EAX with 0
09 jz L1           ; Jump to label L1 if the Zero Flag (ZF) is set
```

Here's a breakdown:

**cmp eax, 0:** This instruction compares the value in the EAX register to zero. After this instruction, the Zero Flag (ZF) will be set if EAX is equal to zero.

**jz L1:** This is a conditional jump instruction. It checks the Zero Flag (ZF). If ZF is set (meaning the comparison result was zero), it jumps to the label L1.

So, in simple terms, this code checks if the value in the EAX register is zero. If it is, it jumps to L1. If not, it continues executing the code below the jz instruction.

```
14 and dl, 10110000b ; Perform a bitwise AND operation on the DL register
15 jnz L2             ; Jump to label L2 if the Zero Flag (ZF) is not set
```

Here's the explanation:

**and dl, 10110000b:** This instruction performs a bitwise AND operation between the value in the DL

register and the binary value 10110000. This operation affects the Zero Flag (ZF). If the result of the AND operation is zero, ZF will be cleared; otherwise, it will be set.

**jnz L2:** This is a conditional jump instruction, just like in the previous example. It checks the Zero Flag (ZF), but this time, it jumps to the label L2 if the ZF is not set (meaning the result of the AND operation was not zero).

So, in this code, if the result of the bitwise AND operation is not zero, it will jump to L2. Otherwise, it will continue with the code after the jnz instruction.

These conditional jumps are essential for controlling program flow in assembly language and are often used for implementing conditional statements like if-else constructs. If you have any more questions or need further clarification, feel free to ask!

---

It seems like you've provided information about conditional jump instructions in assembly language, specifically using the CMP instruction and various flag conditions. Let's break this down further.

**Conditional jump instructions** in assembly language allow you to branch to a destination label based on the state of certain CPU status flags.

These flags are commonly set by arithmetic, comparison, and boolean instructions. Here's a breakdown of some common conditional jump instructions:

**JE (Jump if Equal):** This instruction jumps to a destination label when the Zero flag is set, indicating that the compared values are equal.

```
021 cmp eax, 5  
022 je L1 ; Jump to L1 if EAX equals 5
```

**JC (Jump if Carry):** Jumps to a destination label if the Carry flag is set, indicating that a carry occurred in an arithmetic operation.

**JNC (Jump if Not Carry):** Jumps to a destination label if the Carry flag is clear, indicating no carry occurred in an arithmetic operation.

**JZ (Jump if Zero):** Jumps to a destination label when the Zero flag is set, indicating that a value is zero.

**JNZ (Jump if Not Zero):** Jumps to a destination label when the Zero flag is clear, indicating that a value is not zero.

In your example, you're using the CMP instruction to compare the value in the EAX register to 5. If EAX equals 5, the Zero flag is set by the CMP instruction, and the JE instruction jumps to the label L1. If EAX is not equal to 5, the Zero flag is cleared, and the JE instruction does not jump.

---

### **Jumps Based on Specific Flag Values:**

Conditional jumps in this group rely on the states of specific CPU flags to determine whether to take the jump. Here are some common conditional jumps based on specific flag values:

**JE (Jump if Equal):** Jumps when the Zero flag (ZF) is set, indicating that the compared values are

equal.

**JNE (Jump if Not Equal):** Jumps when the Zero flag (ZF) is clear, indicating that the compared values are not equal.

**JZ (Jump if Zero):** Similar to JE, jumps when the Zero flag (ZF) is set.

**JNZ (Jump if Not Zero):** Similar to JNE, jumps when the Zero flag (ZF) is clear.

**JC (Jump if Carry):** Jumps when the Carry flag (CF) is set, indicating a carry occurred.

**JNC (Jump if Not Carry):** Jumps when the Carry flag (CF) is clear, indicating no carry occurred.

**JO (Jump if Overflow):** Jumps when the Overflow flag (OF) is set, indicating signed overflow.

**JNO (Jump if No Overflow):** Jumps when the Overflow flag (OF) is clear, indicating no signed overflow.

**JS (Jump if Sign):** Jumps when the Sign flag (SF) is set, indicating a negative result.

**JNS (Jump if Not Sign):** Jumps when the Sign flag (SF) is clear, indicating a non-negative result.

### **Jumps Based on Equality Between Operands or the Value of (E)CX:**

These jumps are used for comparing values for equality. The value of (E)CX can also be used for comparisons. Examples include:

**JE (Jump if Equal):** Jumps if two values are equal.

JNE (Jump if Not Equal): Jumps if two values are not equal.

JCXZ (Jump if CX is Zero): Jumps if the (E)CX register is zero.

### **Jumps Based on Comparisons of Unsigned Operands:**

These jumps are used for comparing unsigned integers. They consider values without their sign. Examples include:

JA (Jump if Above): Jumps if the result is strictly greater (unsigned) than another value.

JAE (Jump if Above or Equal): Jumps if the result is greater than or equal (unsigned) to another value.

JB (Jump if Below): Jumps if the result is strictly less (unsigned) than another value.

JBE (Jump if Below or Equal): Jumps if the result is less than or equal (unsigned) to another value.

### **Jumps Based on Comparisons of Signed Operands:**

Similar to the previous group, but used for comparing signed integers, considering their sign. Examples include:

JG (Jump if Greater): Jumps if the result is strictly greater (signed) than another value.

JGE (Jump if Greater or Equal): Jumps if the result is greater than or equal (signed) to another value.

**JL (Jump if Less):** Jumps if the result is strictly less (signed) than another value.

**JLE (Jump if Less or Equal):** Jumps if the result is less than or equal (signed) to another value.

### Example 1:

```
027 mov edx, 0A523h ; Move 0A523h into the edx register  
028 cmp edx, 0A523h ; Compare edx with 0A523h  
029 jne L5           ; Jump if not equal to L5  
030 je L1           ; Jump if equal to L1
```

In this example, cmp compares the value in edx with 0A523h. Since they are equal, the jne instruction is not taken, but the je instruction is taken, leading to a jump to L1.

### Example 2:

```
035 mov bx, 1234h    ; Move 1234h into the bx register  
036 sub bx, 1234h    ; Subtract 1234h from bx  
037 jne L5           ; Jump if not equal to L5  
038 je L1           ; Jump if equal to L1
```

In this example, sub subtracts 1234h from bx, resulting in zero. Therefore, the jne instruction is not taken, but the je instruction is taken, leading to a jump to L1.

### Example 3:

```
042 mov cx, 0FFFFh ; Move FFFFh into the cx register  
043 inc cx          ; Increment cx by 1  
044 jcxz L2        ; Jump if cx is zero to L2
```

Here, jcxz checks if the cx register is zero after the inc instruction. Since inc increments cx by 1, it becomes zero. Hence, the jcxz instruction is taken, leading to a jump to L2.

### Example 4:

```
047 xor ecx, ecx    ; Set ecx to zero using XOR  
048 jecxz L2        ; Jump if ecx is zero to L2
```

In this case, xor is used to set ecx to zero. Then, jecxz checks if ecx is zero. Since it is zero, the jecxz instruction is taken, leading to a jump to L2.

These examples demonstrate how conditional jump instructions like je, jne, jcxz, and jecxz work in assembly language to control program flow based on the result of comparisons and the state of registers.

### Unsigned Comparisons (Table Below):

Mnemonic	Description
JA	Jump if above (if $leftOp > rightOp$ )
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if $leftOp \geq rightOp$ )
JNB	Jump if not below (same as JAE)
JB	Jump if below (if $leftOp < rightOp$ )
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if $leftOp \leq rightOp$ )
JNA	Jump if not above (same as JBE)

These comparisons are used when you are dealing with unsigned values, which means that they don't have a sign (positive or negative).

### Signed Comparisons (Table Below):

Mnemonic	Description
JG	Jump if greater (if $leftOp > rightOp$ )
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if $leftOp \geq rightOp$ )
JNL	Jump if not less (same as JGE)
JL	Jump if less (if $leftOp < rightOp$ )
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if $leftOp \leq rightOp$ )
JNG	Jump if not greater (same as JLE)

These comparisons are used when you are dealing with signed values, which have both positive and negative numbers.

### Example 1:

```

051 mov edx, -1
052 cmp edx, 0
053 jnl L5 ; jump not taken (-1 >= 0 is false)
054 jnle L5 ; jump not taken (-1 > 0 is false)
055 jl L1   ; jump is taken (-1 < 0 is true)

```

In this example, you have a signed comparison. jl jumps because -1 is indeed less than 0.

### Example 2:

```
060 mov bx, +32
061 cmp bx, -35
062 jng L5 ; jump not taken (+32 <= -35 is false)
063 jnge L5; jump not taken (+32 < -35 is false)
064 jge L1 ; jump is taken (+32 >= -35 is true)
```

Again, this is a signed comparison. jge jumps because +32 is indeed greater than or equal to -35.

### Example 3:

```
068 mov ecx, 0
069 cmp ecx, 0
070 jg L5 ; jump not taken (0 > 0 is false)
071 jnl L1 ; jump is taken (0 >= 0 is true)
```

This is a signed comparison. jnl jumps because 0 is greater than or equal to 0.

### Example 4:

```
076 mov ecx, 0  
077 cmp ecx, 0  
078 jl L5 ; jump not taken (0 < 0 is false)  
079 jng L1 ; jump is taken (0 <= 0 is true)
```

Here, jng jumps because 0 is indeed less than or equal to 0.

---

## 1. Conditional Jump Applications:

This section discusses how conditional jump instructions in assembly language can be used to test and manipulate status bits. It demonstrates examples of jumping to labels based on specific bit conditions in a status byte. This is a fundamental concept in assembly programming, allowing you to make decisions in your code based on the state of specific bits.

Conditional jump instructions in assembly language are fundamental for controlling the flow of your program based on specific conditions. They are often used to examine and manipulate individual bits in a byte or word of data. The status bits, such as the Zero Flag (ZF), Sign Flag (SF), and others, are set or cleared by various instructions and can be tested using conditional jumps.

In your provided example:

```
090 mov al, status      ; Load the status byte into AL  
091 test al, 00100000b ; Test bit 5 in AL  
092 jnz DeviceOffline ; Jump to DeviceOffline label if bit 5 is set
```

Here's a breakdown of what's happening:

**mov al, status:** This instruction loads the status byte into the AL register. The AL register is commonly used for working with 8-bit data.

**test al, 00100000b:** The test instruction performs a bitwise AND operation between AL and the binary value 00100000b, which sets all bits to zero except bit 5. This effectively tests if bit 5 in AL is set without modifying AL.

**jnz DeviceOffline:** The jnz (Jump if Not Zero) instruction checks the Zero Flag (ZF). If the Zero Flag is not set, it means that bit 5 in AL was not zero (i.e., bit 5 was set). In this case, the program jumps to the DeviceOffline label.

This example demonstrates how conditional jumps can be used to make decisions based on the state of specific bits in the AL register without changing the value of AL.

Remember that conditional jumps can be used to implement complex logic in assembly language, enabling you to create branching and decision-making in your code.

You can use other conditional jump instructions like je (Jump if Equal), jg (Jump if Greater), jl (Jump if Less), and more to handle various conditions.

## 2. Larger of Two Integers:

Here, the code snippet compares two unsigned integers (EAX and EBX) and moves the larger value to EDX. It uses conditional jumps to make the comparison and assignment. This is a basic example of conditional branching based on integer comparisons.

Certainly, let's delve deeper into the code snippet that compares two unsigned integers (EAX and

EBX) and moves the larger value to EDX. This is a great example of conditional branching based on integer comparisons in assembly language:

```
096 mov edx, eax ; Assume EAX is larger
097 cmp eax, ebx ; Compare EAX and EBX
098 jae L1        ; Jump to L1 if EAX is greater or equal
099 mov edx, ebx ; Move EBX to EDX (EAX was not greater)
100 L1:
101 ; EDX now contains the larger integer
```

Here's a step-by-step breakdown of what's happening:

**mov edx, eax:** Initially, the code assumes that EAX contains the larger integer. It copies the value in EAX to EDX. This is the default assignment.

**cmp eax, ebx:** The cmp instruction compares the values in EAX and EBX without changing them. It sets or clears the appropriate flags (e.g., Zero Flag, Carry Flag) based on the comparison result.

**jae L1:** The jae (Jump if Above or Equal) instruction checks the Carry Flag. If the Carry Flag is not set, it means that EAX is greater than EBX (unsigned comparison). In this case, the program jumps to the L1 label.

**mov edx, ebx:** If the jae condition is not met (EAX is not greater than EBX), the program proceeds to this line and moves the value in EBX to EDX. This effectively updates EDX with the larger integer, which is now in EBX.

**L1::** This is the label where execution continues after the conditional jump. At this point, EDX holds the larger of the two integers, whether it was initially in EAX or EBX.

holds the larger of the two integers, whether it was initially in EAX or EBX.

This code snippet demonstrates how conditional branching is used to compare two integers and select the larger one, updating the EDX register accordingly.

It's important to note that the jae instruction is used for unsigned integer comparison.

If you were comparing signed integers, you would use different conditional jump instructions like jge (Jump if Greater or Equal) or jl (Jump if Less).

### 3. Smallest of Three Integers:

This section shows how to find the smallest of three unsigned 16-bit integers (V1, V2, and V3) and assigns the result to AX. It uses a series of conditional jumps to compare and select the smallest value.

Certainly, let's go through the code snippet that finds the smallest of three unsigned 16-bit integers (V1, V2, and V3) and assigns the result to the AX register. This code uses a series of conditional jumps to make the comparisons and selection:

```
104 .data
105     V1 WORD ?
106     V2 WORD ?
107     V3 WORD ?
108
109 .code
110     mov ax, V1      ; Assume V1 is the smallest
111     cmp ax, V2      ; Compare AX and V2
112     jbe L1          ; Jump to L1 if AX is less than or equal to V2
113     mov ax, V2      ; Move V2 to AX (V1 is not the smallest)
114     L1:
115     cmp ax, V3      ; Compare AX and V3
116     jbe L2          ; Jump to L2 if AX is less than or equal to V3
117     mov ax, V3      ; Move V3 to AX (V2 or V1 is not the smallest)
118     L2:
119     ; AX now contains the smallest integer among V1, V2, and V3
```

Here's a step-by-step explanation of how this code works:

The code starts with the assumption that V1 is the smallest integer and loads the value of V1 into the AX register.

It then compares the value in AX (which now holds V1) with the value of V2 using the cmp instruction. The jbe (Jump if Below or Equal) instruction checks whether AX is less than or equal to V2.

If AX is less than or equal to V2 (the jbe condition is met), the program jumps to the label L1. In this case, V1 remains the smallest integer in AX.

If AX is not less than or equal to V2 (the jbe condition is not met), it means V2 is smaller, and the program updates AX with the value of V2.

The program then continues to compare the current value in AX (either V1 or V2) with V3 using the same cmp and jbe instructions. If AX is less than or equal to V3, it keeps the smallest value. If not, it updates AX with V3.

After these comparisons and conditional jumps, AX will contain the smallest of the three unsigned 16-bit integers (V1, V2, and V3).

This code demonstrates how to find the smallest integer among three values using conditional branching in assembly language.

#### **4. Loop until Key Pressed:**

In this part, a loop continuously runs until a standard alphanumeric key is pressed. It uses the ReadKey method from the Irvine32 library to check for a key press. If no key is present, the loop continues with a 10-millisecond delay between iterations. This is a practical example of waiting for user input in assembly code.

Certainly, the provided code is an example of creating a loop that continuously runs until a standard alphanumeric key is pressed.

It uses the ReadKey method from the Irvine32 library to check for a key press, and if no key is present, it continues with a 10-millisecond delay between iterations.

This is a practical way to wait for user input in assembly code. Let's break down the code:

```
123 .data  
124 char BYTE ?  
125  
126 .code  
127 L1:  
128     mov eax, 10          ; Create a 10 ms delay  
129     call Delay  
130     call ReadKey        ; Check for a key press  
131     jz L1                ; If no key is pressed, repeat the loop  
132     mov char, AL          ; Save the character in the 'char' variable
```

Here's how this code works step by step:

**mov eax, 10:** This line sets up a delay by loading the value 10 into the EAX register. The Delay subroutine is then called to introduce a 10-millisecond pause. This delay is important to give the system some time to process other tasks and to avoid rapidly consuming CPU resources in a tight loop.

**call ReadKey:** The ReadKey subroutine is called to check for a key press. The result of this function is stored in the AL register. If a key is pressed, AL will contain the ASCII code of the key; otherwise, it will be 0.

**jz L1:** The jz instruction (Jump if Zero) checks whether the Zero Flag (ZF) is set. If AL is 0, it means no key was pressed, and the program jumps back to the L1 label, continuing the loop.

**mov char, AL:** If a key is pressed (i.e., AL is not 0), the ASCII code of the pressed key is stored in the char variable.

The loop continues until a key is pressed, and when a key is pressed, its ASCII code is stored in the char variable. This way, you can wait for and capture user input in your assembly program.

This is a practical way to handle user input in assembly code, especially when you want to wait for specific keypresses in a controlled manner.

---

The provided code is a simple example of how to search for the first nonzero value in an array of 16-bit integers.

```
135 ; Scanning an Array (ArrayScan.asm)
136 ; Scan an array for the first nonzero value.
137 INCLUDE Irvine32.inc
138 .data
139     intArray SWORD 0,0,0,0,1,20,35,-12,66,4,0
140     noneMsg BYTE "A non-zero value was not found",0
141 .code
142     main PROC
143         ; Initialize registers and variables
144         mov esi, 0          ; Index for array traversal
145         mov ecx, LENGTHOF intArray ; Length of the array
146         mov ebx, ADDR intArray ; Address of the array
147         mov al, 0            ; Clear AL register to store the result (found or not)
148     searchLoop:
149         cmp word ptr [ebx + esi * 2], 0 ; Compare the current element with zero
150         jnz foundNonZero      ; Jump if not zero
151         inc esi              ; Increment index
152         loop searchLoop      ; Continue loop until ecx is zero
153         mov al, 1              ; Set AL to 1 if no nonzero value found
154         jmp done
155     foundNonZero:
156         mov al, 0              ; Set AL to 0 if a nonzero value is found
157     done:
158         ; Display appropriate message based on AL value
159         cmp al, 0
160         je noNonZeroFound
161         mov edx, OFFSET noneMsg
162         call WriteString
163         jmp endProgram
```

```
164 noNonZeroFound:  
165     ; Display the first nonzero value found  
166     mov edx, [ebx + esi * 2]  
167     call WriteInt  
168  
169 endProgram:  
170     call Crlf  
171     exit  
172 main ENDP  
173  
174 END main
```

Explanation:

We define the array intArray containing 16-bit integers and a message noneMsg.

In the code section, we initialize registers and variables. esi is used to keep track of the array index, ecx holds the length of the array, and ebx stores the address of intArray. al is initially set to 0, which will be used to determine if a nonzero value is found.

We use a loop labeled as searchLoop to traverse the array and compare each element to 0 using the cmp instruction. If the element is not zero (jnz instruction), we jump to the foundNonZero label.

If we reach the end of the loop without finding a nonzero value, we set al to 1 to indicate that no nonzero value was found.

We have separate code for displaying messages based on the value of al. If al is 0, we display the

nonzero value found; if it's 1, we display the "non-zero value not found" message.

The program then ends by calling Crlf and exiting.

You can uncomment different test data configurations in the .data section to test the program with various arrays.

=====

## ***Encryption Program Overview***

=====

This assembly program demonstrates a simple symmetric encryption technique using the XOR operation. The program follows these steps:

- **User Input:** The user enters a plain text message.
- **Encryption:** The program encrypts the plain text by XORing each character with a single character key and displays the cipher text.
- **Decryption:** It then decrypts the cipher text using the same key and displays the original plain text.

```
177 INCLUDE Irvine32.inc
178
179 KEY = 239           ; The encryption key (single character)
180 BUFMAX = 128        ; Maximum buffer size for input
181
182 .data
183     sPrompt BYTE "Enter the plain text:",0
184     sEncrypt BYTE "Cipher text: ",0
185     sDecrypt BYTE "Decrypted: ",0
186     buffer BYTE BUFMAX+1 DUP(0)
187     bufSize DWORD ?
188
189 .code
190     main PROC
```

The program starts by including the Irvine32 library for input and output functions.

KEY is defined as the encryption key, set to 239.

BUFMAX defines the maximum buffer size for input.

```
194    call InputTheString  
195    call TranslateBuffer  
196    mov edx, OFFSET sEncrypt  
197    call DisplayMessage  
198    call TranslateBuffer  
199    mov edx, OFFSET sDecrypt  
200    call DisplayMessage  
201    exit  
202 main ENDP
```

The main procedure calls InputTheString to get user input, TranslateBuffer for encryption, and DisplayMessage to show the cipher text. It repeats this process for decryption.

```
206 InputTheString PROC  
207     pushad  
208     mov edx, OFFSET sPrompt  
209     call WriteString  
210     mov ecx, BUFMAX  
211     mov edx, OFFSET buffer  
212     call ReadString  
213     mov bufSize, eax  
214     call Crlf  
215     popad  
216     ret  
217 InputTheString ENDP
```

InputTheString procedure prompts the user for input, reads it into the buffer, and stores its length in bufSize.

```
221 DisplayMessage PROC  
222     pushad  
223     call WriteString  
224     mov edx, OFFSET buffer  
225     call WriteString  
226     call Crlf  
227     call Crlf  
228     popad  
229     ret  
230 DisplayMessage ENDP
```

DisplayMessage procedure displays a given message (in EDX) followed by the contents of the buffer and two line breaks.

```
235 TranslateBuffer PROC  
236     pushad  
237     mov ecx, bufSize  
238     mov esi, 0 L1:  
239     xor buffer[esi], KEY  
240     inc esi  
241     loop L1  
242     popad  
243     ret  
244 TranslateBuffer ENDP
```

TranslateBuffer procedure translates the string in the buffer by XORing each byte with the encryption key (KEY).

**Final Note:**

The program uses a single-character key (which is not secure in real-world scenarios). The exercises suggest using a multi-character encryption key for stronger security. This program is a basic example to understand the concept of XOR-based encryption in assembly language. In practice, encryption algorithms like AES or RSA are used for secure data protection.

---

**Which jump instructions follow unsigned integer comparisons?**

Jump instructions following unsigned integer comparisons typically include JA (Jump if Above), JAE (Jump if Above or Equal), JB (Jump if Below), and JBE (Jump if Below or Equal).

**Which jump instructions follow signed integer comparisons?**

Jump instructions following signed integer comparisons usually include JG (Jump if Greater), JGE (Jump if Greater or Equal), JL (Jump if Less), and JLE (Jump if Less or Equal).

**Which conditional jump instruction is equivalent to JNAE?**

JNAE stands for "Jump if Not Above or Equal," and its equivalent for signed comparisons is JB which stands for "Jump if Below."

**Which conditional jump instruction is equivalent to the JNA instruction?**

The JNA instruction stands for "Jump if Not Above," and its equivalent for signed comparisons is JL, which stands for "Jump if Less."

**Which conditional jump instruction is equivalent to the JNGE instruction?**

JNGE stands for "Jump if Not Greater or Equal," and its equivalent for signed comparisons is JG, which stands for "Jump if Greater."

(Yes/No): Will the following code jump to the label named Target?

```
247 mov ax, 8109h  
248 cmp ax, 26h  
249 jg Target
```

Yes, the code will jump to the label named "Target" if the value in the ax register (8109h) is greater than the immediate value 26h. This is because jg stands for "Jump if Greater."

## *Conditional Loops*

---

## ***LOOPZ and LOOPE Instructions***

---

The **LOOPZ** (**L**oop **I**f **Z**ero) and **LOOPE** (**L**oop **I**f **E**qual) instructions are conditional loop instructions used in assembly language programming.

They both share the same opcode and have identical behavior based on the condition of the Zero Flag (ZF). Here's a simplified explanation:

**ECX (or RCX in 64-bit mode)** is the loop counter register. ECX is decremented by 1 in each iteration of the loop.

The **loop continues if** ECX is greater than 0 and the Zero Flag (ZF) is set (indicating the result of the previous operation was zero).

If the condition is met, the program jumps to the specified destination label. If the condition is not met (ECX not greater than 0 or ZF not set), the loop exits, and control proceeds to the next instruction.

For example, the following code snippet will sum the elements of an array using the LOOPZ instruction:

```
256 ; sum the elements of the array `array`  
257 mov ecx, array_size  
258 loopz sum_loop  
259  
260 ; sum_loop:  
261 add eax, [array + ecx - 1]  
262 dec ecx  
263 jnz sum_loop
```

In this example, the loop counter register ECX is initialized with the size of the array.

The LOOPZ instruction then decrements ECX and adds the element at array + ECX - 1 to the accumulator register EAX. If ECX is still greater than 0 and the Zero flag is set, the loop will jump back to the sum\_loop label.

Otherwise, execution will fall through to the next instruction, which will be the end of the loop.

**LOOPZ (Loop if Zero):** This instruction is used to create a loop with an additional condition. The condition is that the Zero Flag (ZF) must be set for the loop to continue. Here's the syntax:

## LOOPZ destination

These instructions do not affect any other status flags. LOOPE is essentially the same as LOOPZ, and they **share the same opcode**. They both have the same behavior and conditions as described above.

These instructions are often used for implementing loops in assembly language, where you want to repeat a block of code a specific number of times while a certain condition is met (in this case, the Zero Flag being set). Loop a specific number of times based on the value in the loop counter (ECX) and a condition (Zero Flag set).

The LOOPZ and LOOPE instructions can be used in a variety of other ways as well. For example, they can be used to implement nested loops, to search for a value in an array, or to reverse a string.

=====

### ***LOOPNZ (Loop if Not Zero)***

### ***LOOPNE (Loop if Not Equal) Instructions***

=====

The **LOOPNZ (Loop if Not Zero)** and **LOOPNE (Loop if Not Equal)** instructions are used in assembly language programming to create loops that repeat a block of code while a certain condition is met.

These instructions are quite similar and often interchangeable, as they share the same opcode and perform similar tasks.

The LOOPNZ instruction continues looping while two conditions are met: The **unsigned value** of the **ECX register** is **greater than zero** after being decremented. The **Zero Flag (ZF) is clear**. The syntax for LOOPNZ is:

**LOOPNZ destination**

**Here's how it works:**

- Decrement ECX by 1.
  - If ECX > 0 and ZF = 0 (i.e., the Zero Flag is clear), jump to the specified destination label.
  - If ECX becomes zero or ZF is set, the loop terminates, and control passes to the next instruction.
- 

The **LOOPNE** instruction is equivalent to LOOPNZ in terms of functionality and shares the same opcode.

**It also performs the following tasks:**

- Decrement ECX by 1.
- If ECX > 0 and ZF = 0 (i.e., the Zero Flag is clear), jump to the specified destination label.
- If ECX becomes zero or ZF is set, the loop terminates, and control passes to the next instruction.

In essence, both LOOPNZ and LOOPNE create loops that continue while a counter (usually stored in ECX) is not zero and the Zero Flag is not set. They are often used for iterating through arrays or data structures until a specific condition is met.

---

Here's an example code excerpt that uses LOOPNZ to scan an array until a non-negative number is found:

```
270 .data
271     array SWORD -3,-6,-1,-10,10,30,40,4
272     sentinel SWORD 0
273 .code
274     mov esi, OFFSET array
275     mov ecx, LENGTHOF array
276     L1:
277     test WORD PTR [esi], 8000h ; Test sign bit
278     pushfd
279     add esi, TYPE array
280     popfd
281     loopnz L1
282     jnz quit
283     sub esi, TYPE array
284     quit:
```

This code iterates through the array, testing the sign bit of each element, and continues the loop until a nonnegative value is found. If none is found, it stops when ECX becomes zero and jumps to the quit label, where ESI points to the sentinel value (0) located after the array.

Now, let's break down the code inside the .code section:

**mov esi, OFFSET array:** This instruction initializes the ESI register with the memory address of the array variable, effectively pointing to the beginning of the array.

**mov ecx, LENGTHOF array:** It loads the ECX register with the length of the array, which is the number of elements in the array.

**L1:** This is a label for the beginning of a loop.

**test WORD PTR [esi], 8000h:** The test instruction checks the sign bit of the current array element by bitwise ANDing it with 8000h (hexadecimal representation of a signed word with the sign bit set).

**pushfd:** It pushes the processor flags onto the stack. This is done to save the state of the Zero Flag (ZF) because the add instruction that follows modifies the flags.

**add esi, TYPE array:** ESI is incremented by the size of a single array element (TYPE array), effectively moving to the next position in the array.

**popfd:** The flags saved by pushfd are popped from the stack, restoring the previous state.

**loopnz L1:** The LOOPNZ instruction decrements ECX by 1 and checks if ECX > 0 and ZF = 0. If both conditions are met, it jumps to the L1 label, continuing the loop. Otherwise, if ECX becomes zero or ZF is set, the loop terminates.

**jnz quit:** If the loop completes without finding a nonnegative value, it jumps to the quit label.

**sub esi, TYPE array:** If a nonnegative value is found, ESI is left pointing to that value.

The code efficiently iterates through the array, testing each element's sign bit. If it finds a nonnegative value, ESI points to that value; otherwise, it points to the sentinel value (0) after the array. This logic allows you to handle different cases depending on the outcome of the loop.

---

**(True/False): The LOOPE instruction jumps to a label when (and only when) the Zero flag is clear.**

True. The LOOPE (Loop If Equal) instruction jumps to a label when the Zero flag (ZF) is clear and the ECX register is greater than zero.

**(True/False): In 32-bit mode, the LOOPNZ instruction jumps to a label when ECX is greater than zero and the Zero flag is clear.**

True. The LOOPNZ (Loop If Not Zero) instruction in 32-bit mode jumps to a label when the ECX register is greater than zero and the Zero flag (ZF) is clear.

**(True/False): The destination label of a LOOPZ instruction must be no farther than ±128 or ±127 bytes from the instruction immediately following LOOPZ.**

False. The destination label of a LOOPZ instruction must be no farther than ±128 bytes from the instruction immediately following LOOPZ. This is because the offset for a short jump (like the one used by LOOPZ) is limited to a signed 8-bit value, which covers a range of -128 to +127 bytes. Modify the LOOPNZ example in Section 6.4.2 so that it scans for the first negative value in the array. Change the array initializers so they begin with positive values.

To modify the LOOPNZ example to scan for the first negative value in the array, you can change the array initialization to start with positive values. Here's an example in C:

```
288 int values[] = { 1, 2, 3, -4, 5, 6 };
289 int array_length = sizeof(values) / sizeof(values[0]);
290 int found_negative = 0;
291
292 __asm {
293     mov ecx, array_length
294     mov esi, 0 ; Index for accessing the array
295
296     start_loop:
297         cmp [values + esi * 4], 0 ; Compare the current element with zero
298         jns not_negative ; Jump if not negative
299         ; Code to handle negative value goes here (set found_negative flag, etc.)
300         jmp loop_end
301
302     not_negative:
303         inc esi ; Increment the array index
304         loop start_loop
305
306     loop_end:
307 }
```

Challenge: The LOOPNZ example in Section 6.4.2 relies on a sentinel value to handle the possibility that a positive value might not be found. What might happen if you removed the sentinel?

If you remove the sentinel value from the LOOPNZ example, it means that there is no clear indicator to stop the loop when a positive value is not found in the array. Without a sentinel value, the loop would continue running indefinitely, potentially causing an infinite loop in your program. To avoid

this, it's essential to have some mechanism, like a sentinel value or a counter, to terminate the loop when a specific condition is not met.

-----

Certainly, here's an example of how you might modify the LOOPNZ code to remove the sentinel value and handle the case where a positive value might not be found. In this modified code, we use a counter to limit the number of iterations:

```

312 int values[] = { -1, -2, -3, -4, -5 }; // Array with only negative values
313 int array_length = sizeof(values) / sizeof(values[0]);
314 int iterations = 0; // Counter for loop iterations
315
316 __asm {
317     mov ecx, array_length
318     mov esi, 0 ; Index for accessing the array
319
320     start_loop:
321         cmp [values + esi * 4], 0 ; Compare the current element with zero
322         jns not_negative ; Jump if not negative
323         ; Code to handle negative value goes here (set found_negative flag, etc.)
324         jmp loop_end
325
326     not_negative:
327         inc esi ; Increment the array index
328         loop_start:
329         inc iterations
330         cmp iterations, array_length ; Check if we have iterated through the entire array
331         jge loop_end ; If iterations >= array_length, exit the loop
332         jmp start_loop ; Continue looping
333
334     loop_end:
335 }

```

In this code, we've introduced an `iterations` counter, which increments with each iteration of the loop. If `iterations` becomes greater than or equal to `array_length`, it means we've iterated through the entire array without finding a positive value, and we exit the loop. This prevents an infinite

loop from occurring when there are no positive values in the array.

## ***Conditional Structures***

**Conditional structures** in programming allow you to make decisions based on conditions. You've likely used these in high-level languages like C++. These structures involve evaluating a condition and executing different sets of instructions based on whether the condition is true or false.

**Block-Structured IF Statements:** In the context of C++ and similar languages, an IF statement consists of a boolean expression followed by two sets of statements. One set of statements executes when the expression is true, and the other set when the expression is false. The "else" part is optional, meaning you can have just an "if" without an "else."

**Translation to Assembly:** When you translate high-level code into assembly language, you need to perform two main steps:

**Evaluate the Boolean Expression:** This step involves comparing values in registers or memory locations in such a way that it affects a CPU status flag. In your example, they use the CMP instruction to compare op1 and op2. This sets flags in the CPU.

**Conditional Jumps:** Based on the CPU status flags set by the comparison, you decide which branch of the code to execute. In your example, they use JNE (Jump if Not Equal) to skip ahead if op1 is not equal to op2.

**Code Example:** The code example you provided in C++ checks if op1 is equal to op2. If they are equal,

it sets X to 1 and Y to 2.

Now, let's look at the assembly code they provided:

```
340 mov eax, op1      ; Move op1 into the EAX register
341 cmp eax, op2      ; Compare op1 and op2
342 jne L1            ; Jump if not equal (if op1 != op2)
343 mov X, 1           ; Assign 1 to X
344 mov Y, 2           ; Assign 2 to Y
345 L1:
```

**mov eax, op1:** This line moves the value of op1 into the EAX register for comparison.

**cmp eax, op2:** It then compares the values in EAX (which now holds op1) and op2.

**jne L1:** If the comparison shows that op1 is not equal to op2, it jumps to the label L1.

**mov X, 1 and mov Y, 2:** Inside L1, it assigns 1 to X and 2 to Y. This code allows for efficient execution by letting the code "fall through" when the boolean condition is true. It avoids unnecessary jumps, making it more compact.

The alternative implementation using JE (Jump if Equal) would be slightly less compact but functionally equivalent.

---

**..it jumps to the label L1. why not say it jumps to the procedure L1?**

**ChatGPT**

You're correct; the term "procedure" is often used in programming to refer to a block of code that has a specific purpose or function. In the context of assembly language, especially in the example you provided, "L1" is more commonly referred to as a "label."

In assembly language, labels are used to mark specific locations in the code, typically for branching purposes. So, when I mentioned "jumps to the label L1," it means that the program's execution flow will transfer to the point in the code marked by the label "L1."

While the term "procedure" is indeed used in programming, it's usually associated with higher-level languages like C++ or C# and refers to a named block of code that can be called or invoked with specific parameters. In assembly language, labels are more commonly used to mark positions in the code for control flow.

So, you could also say "jumps to the procedure L1" if you were discussing this in a higher-level language context, but in the context of assembly language, "jumps to the label L1" is the more accurate terminology.

---

**In high-level programming languages, labels and procedures are distinct concepts:**

- **Labels:** Labels are not typically used in high-level languages like C++, C#, Python, etc. In these languages, you don't typically define labels explicitly. Instead, control flow is managed using constructs like functions, loops, and conditional statements. Labels are more commonly associated with assembly language or lower-level languages.

- **Procedures (Functions):** Procedures and functions are an integral part of high-level programming languages. They are named blocks of code that can be called or invoked with specific parameters. Functions in high-level languages serve various purposes, such as performing a specific task, returning values, or modifying data. They encapsulate a set of instructions and can be reused throughout the program.

### So, in summary:

- In high-level languages, you use functions (or procedures) to encapsulate code with a specific purpose and make your code more modular and readable.
  - Labels are more commonly associated with low-level languages like assembly, where they are used as markers for control flow and branching.
  - The terminology and usage can vary between different programming languages, but this distinction is generally applicable.
- 

We want to implement these conditional structures in real life:

#### **Example 2: Setting Cluster Size Based on Volume Capacity**

In this example, the pseudocode is used to set the cluster size of a disk volume in the NTFS file system. The cluster size depends on the volume's capacity, specifically, whether it's less than 16 terabytes.

Here's the assembly language implementation:

```
349 mov clusterSize, 8192 ; Assume a larger cluster size  
350  
351 cmp terrabytes, 16 ; Compare terrabytes with 16  
352 jae next ; If greater or equal, jump to 'next'  
353  
354 mov clusterSize, 4096 ; Set clusterSize to 4096 for smaller volume  
355  
356 next:
```

mov clusterSize, 8192: Initially, the code assumes a larger cluster size of 8192.

cmp terrabytes, 16: It compares the variable terrabytes with 16.

jae next: If terrabytes is greater than or equal to 16, it jumps to the next label.

mov clusterSize, 4096: Inside the next label, it sets clusterSize to 4096, indicating a smaller cluster size.

The code effectively changes the cluster size based on the volume size, as described in the pseudocode.

### Example 3: Conditional Routine Calls

In this example, the pseudocode involves calling different routines based on a condition, specifically, whether op1 is greater than op2.

Here's the assembly language implementation:

```
360 mov eax, op1      ; Move op1 to a register
361 cmp eax, op2      ; Compare op1 and op2
362 jg A1             ; If op1 > op2, jump to A1 (call Routine1)
363 call Routine2     ; Otherwise, call Routine2
364 jmp A2             ; Jump to A2 (exit the IF statement)

365
366 A1:
367 call Routine1
368
369 A2:
```

**mov eax, op1:** It moves the value of op1 into the EAX register.

**cmp eax, op2:** It compares the values in EAX (which now holds op1) and op2.

**jg A1:** If op1 is greater than op2, it jumps to A1, which calls Routine1.

**call Routine2:** If the comparison is false, it calls Routine2.

After executing either Routine1 or Routine2, it jumps to A2, which marks the exit point for the IF statement.

The code effectively calls different routines based on the condition provided in the pseudocode.

## ***WhiteBox Testing***

**White box testing**, also known as clear box testing, glass box testing, transparent box testing, or structural testing, is a software testing method that examines the internal structure, design, and coding of an application to verify input-output flow and improve design, usability, and security. It is one of two parts of the box testing approach to software testing, the other being **black box testing**.

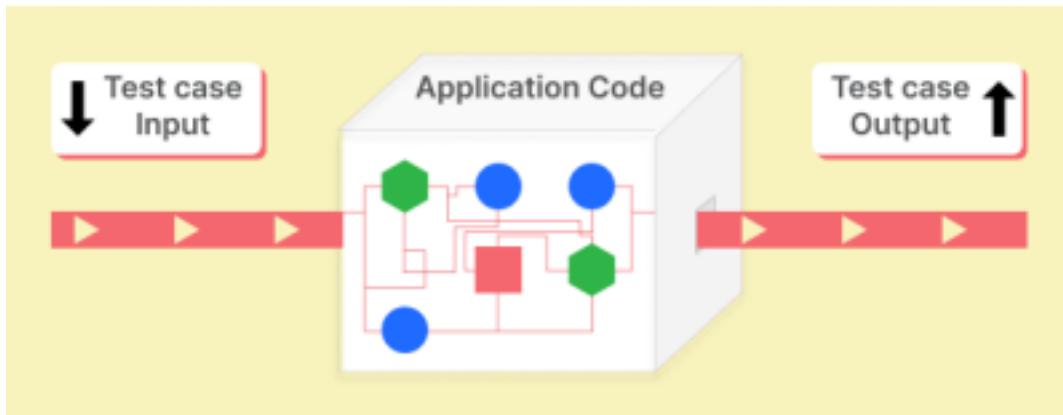
White box testing involves testing the internal logic and execution paths of a subroutine by examining the source code.

# White Box Testing

Unit Testing

Integration Testing

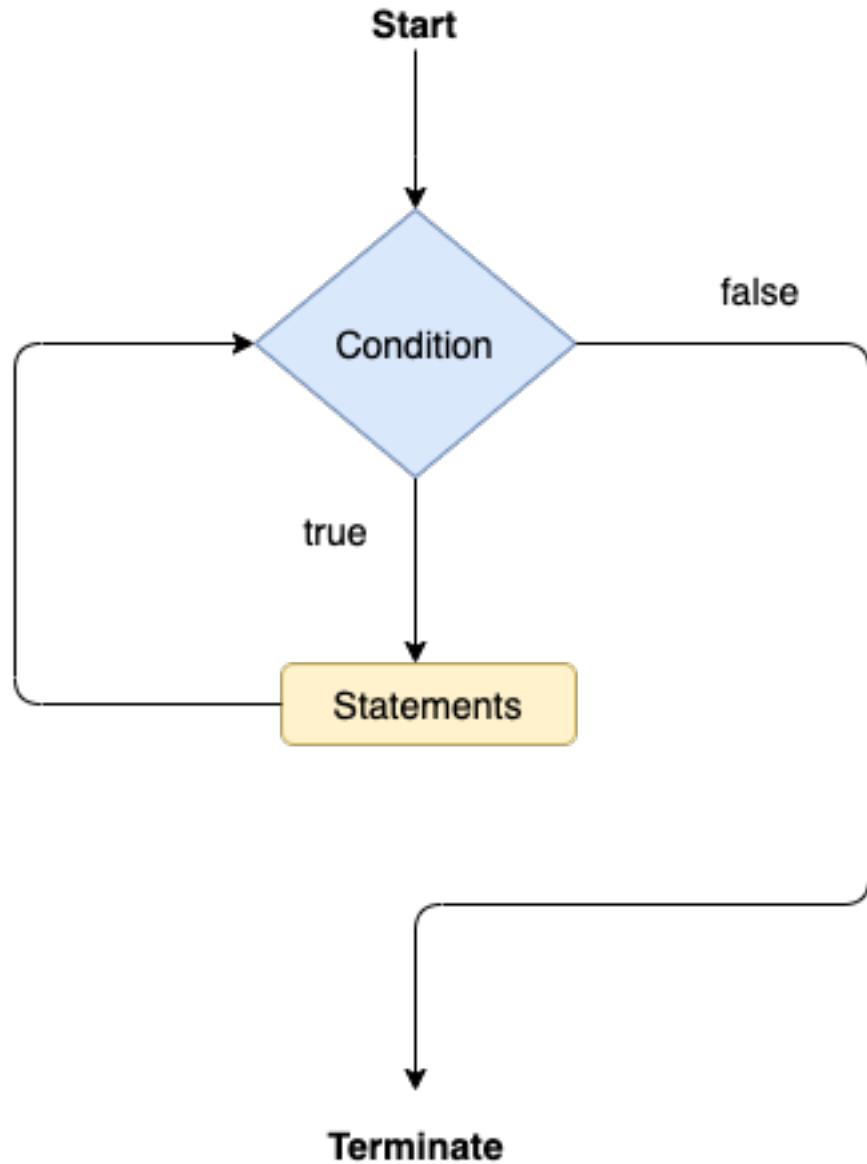
System Testing



In **white box testing**, testers have access to the source code of the application and use this knowledge to design test cases that can **verify the correctness of the software at the code level**. White box testing is often used to test the following aspects of a software application:

```
$(function(){cards();});
$(window).on('resize', function(){cards();
    function cards(){
        var width = $(window).width();
        if(width < 750){
            cardssmallscreen();
        }else{
            cardsbigscreen();
        }
    }
    function cardssmallscreen(){
        var cards = $('.card').length;
        height = 0;
        d2 = 2;
        for(i=0;i<cards;i++){
            $(".card").eq(i).css("height", height);
            height += 20;
        }
    }
},function cardsbigscreen(){
    var cards = $('.card').length;
    height = 0;
    d2 = 2;
    for(i=0;i<cards;i++){
        $(".card").eq(i).css("height", height);
        height += 20;
    }
});
```

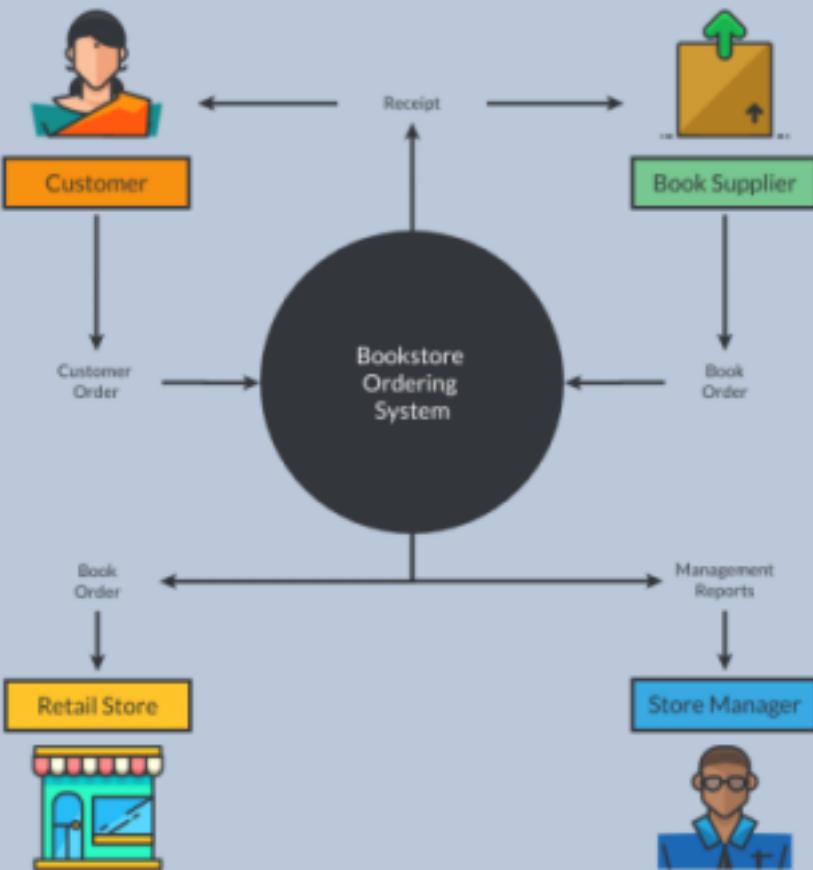
**Control flow:** White box testing can be used to ensure that all of the possible execution paths through an application are tested. This can be done by using techniques such as control flow analysis and path testing.



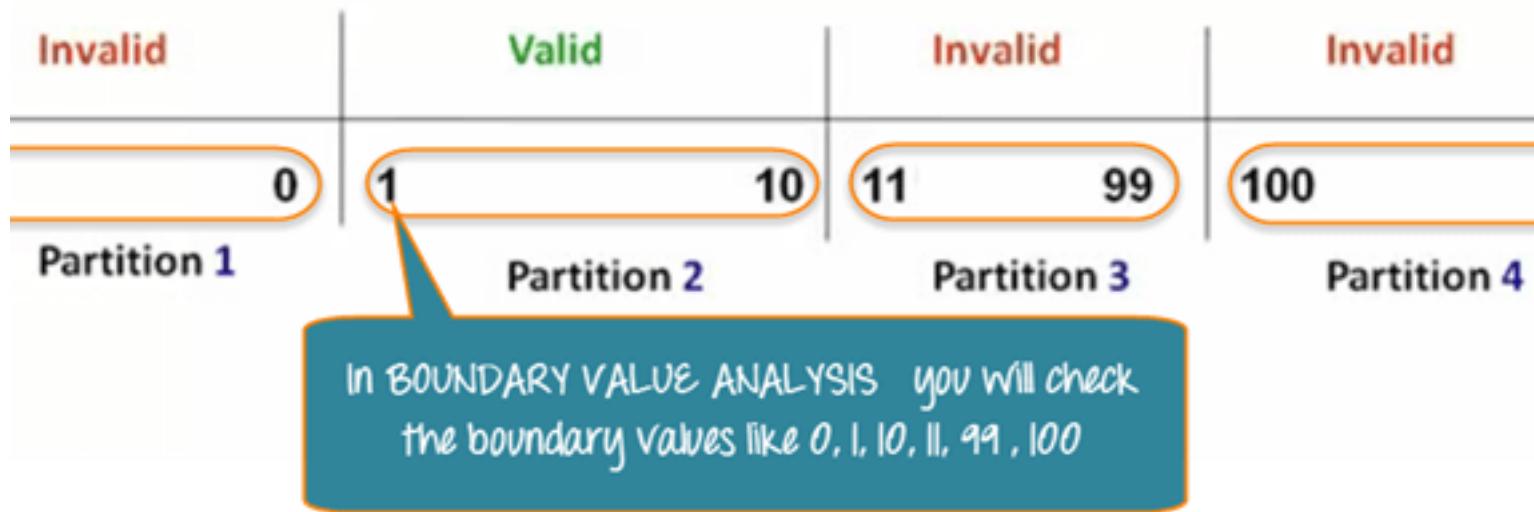
**Data flow:** White box testing can be used to ensure that all of the possible data flows through an application are tested. This can be done by using techniques such as data flow analysis and taint analysis.

# Bookstore Ordering System

(Data Flow Diagram)



**Boundary values:** White box testing can be used to ensure that the application behaves correctly at the boundaries of its input and output values. This can be done by using techniques such as equivalence partitioning and boundary value analysis.



**Error handling:** White box testing can be used to ensure that the application handles errors correctly. This can be done by designing test cases that trigger different types of errors.

# Error Handling in C

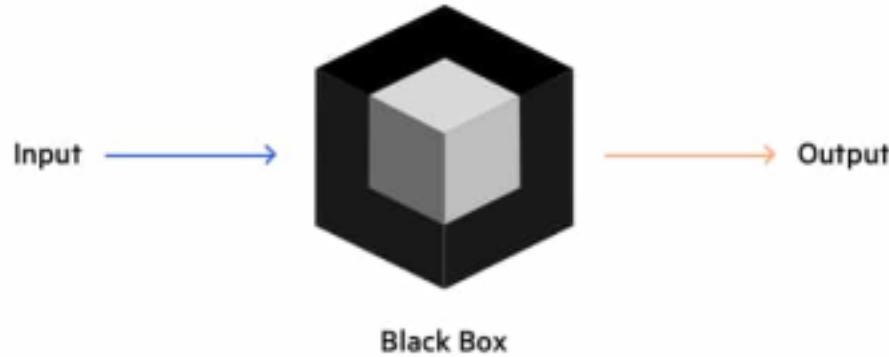


White box testing is a powerful tool for ensuring the quality of software applications.

However, it is important to note that white box testing alone cannot guarantee that an application is bug-free.

**Black box testing** is also necessary to test the application from a user's perspective and to ensure that it meets all of its functional requirements.

## **Black Box Testing**



### **Benefits of white box testing:**

- Identify bugs early in the development process.
- Improve code quality by identifying potential design problems and inefficiencies.
- Improve security by identifying potential vulnerabilities

### **Drawbacks of white box testing:**

- Time-consuming and expensive.
- Difficult to perform if testers do not have a good understanding of the source code.
- Cannot guarantee that an application is bug-free.

In your provided assembly language code example, you're implementing a nested IF-ELSE statement and conducting white box testing by assigning different values to the variables and tracing the execution paths.

Let's break down the code and the testing results:

```
1: mov eax, op1      ; Move op1 into eax register
2: cmp eax, op2      ; Compare op1 with op2
3: jne L2            ; Jump to L2 if op1 != op2
4: mov eax, X         ; Move X into eax register
5: cmp eax, Y         ; Compare X with Y
6: jg L1              ; Jump to L1 if X > Y
7: call Routine2     ; Call Routine2
8: jmp L3              ; Jump to L3 and exit
9: L1: call Routine1 ; Call Routine1
10: jmp L3             ; Jump to L3 and exit
11: L2: call Routine3 ; Call Routine3
12: L3:                 ; Exit point
```

```
387 if op1 == op2
388     if X > Y
389         call Routine1
390     else
391         call Routine2
392     end if
393
394 else
395     call Routine3
396 end if
```

Table 6-6 shows the results of white box testing of the sample code. In the first four columns test values have been assigned to op1, op2, X, and Y. The resulting execution paths are verified in columns 5 and 6.

TABLE 6-6 Testing the Nested IF Statement.

op1	op2	X	Y	Line Execution Sequence	Calls
10	20	30	40	1, 2, 3, 11, 12	Routine3
10	20	40	30	1, 2, 3, 11, 12	Routine3
10	10	30	40	1, 2, 3, 4, 5, 6, 7, 8, 12	Routine2
10	10	40	30	1, 2, 3, 4, 5, 6, 9, 10, 12	Routine1

The first four columns show the test values assigned to op1, op2, X, and Y. The fifth column shows the execution path that is taken, based on the test values. The sixth column shows the output that is produced, based on the execution path.

For example, in the first test case, the values of op1 and op2 are both 10, and the values of X and Y are 20 and 30, respectively. Since op1 is equal to op2, the execution path will follow the first branch of the IF statement, and Routine1 will be called. The output of Routine1 is unspecified, but it is likely to return a value that indicates that the condition  $op1 == op2$  is true.

In the second test case, the values of op1 and op2 are both 10, and the values of X and Y are 30 and

20, respectively. Again, since op1 is equal to op2, the execution path will follow the first branch of the IF statement. However, this time, the condition  $X > Y$  is also true, so the execution path will follow the first branch of the nested IF statement. This will result in Routine1 being called.

In the third test case, the values of op1 and op2 are 10 and 20, respectively, and the values of X and Y are 30 and 20, respectively. Since op1 is not equal to op2, the execution path will follow the second branch of the IF statement. The condition  $X > Y$  is false, so the execution path will follow the second branch of the nested IF statement. This will result in Routine2 being called.

In the fourth test case, the values of op1 and op2 are 10 and 20, respectively, and the values of X and Y are 20 and 30, respectively. Since op1 is not equal to op2, the execution path will follow the second branch of the IF statement. The condition  $X > Y$  is also false, so the execution path will fall through to the ELSE clause of the nested IF statement. This will result in Routine3 being called.

White box testing is a valuable tool for ensuring that complex conditional statements are working as expected. By testing all possible combinations of input values, programmers can be confident that their code will handle all possible scenarios correctly.

## *Short-circuit evaluation(AND)*

**Short-circuit evaluation** is a technique used by compilers and interpreters to optimize the evaluation of Boolean expressions.

With short-circuit evaluation, the second operand of an AND expression is only evaluated if the first operand is true.

This is because if the first operand is false, then the overall expression must be false, regardless of the value of the second operand.

The following assembly language code implements short-circuit evaluation for the AND operator:

```
399 if (al > bl) AND (bl > cl)
400     X = 1
401 end if
```

```
404 cmp al, bl
405 jbe next
406 cmp bl, cl
407 jbe next
408 mov X, 1
409 next:
```

- This code first compares the values of the registers al and bl.
- If al is less than or equal to bl, then the second operand of the AND expression is not evaluated, and the program jumps to the next label.

- Otherwise, the program compares the values of the registers bl and cl.
- If bl is less than or equal to cl, then the program jumps to the next label.
- Otherwise, the program stores the value 1 in the register X and jumps to the next label.
- The next label is used to exit the code, regardless of whether the AND expression evaluated to true or false.

The following assembly language code implements short-circuit evaluation for the AND operator without using a jbe instruction:

```

413 cmp al, bl
414 ja L1
415 jmp next
416 L1:
417 cmp bl, cl
418 ja L2
419 jmp next
420 L2:
421 mov X, 1
422 next:

```

This code is functionally equivalent to the previous example, but it uses a ja instruction instead of a jbe instruction. The ja instruction jumps to the specified label if the first operand is greater than the second operand.

The following table shows the difference between the two code examples:

Instruction	Description
<code>cmp al, bl</code>	Compares the values of the registers <code>al</code> and <code>bl</code> .
<code>jbe next</code>	Jumps to the next label if <code>al</code> is less than or equal to <code>bl</code> .
<code>cmp bl, cl</code>	Compares the values of the registers <code>bl</code> and <code>cl</code> .
<code>ja L1</code>	Jumps to the <code>L1</code> label if <code>bl</code> is greater than <code>al</code> .
<code>jmp next</code>	Jumps to the next label.
<code>L1:</code>	Label.
<code>mov X, 1</code>	Stores the value 1 in the register <code>X</code> .
<code>L2:</code>	Label.

The first code example is more efficient because it uses a `jbe` instruction instead of a `ja` instruction. The `jbe` instruction can be implemented as a single machine instruction, while the `ja` instruction may require multiple machine instructions.

In practice, the compiler will typically generate the most efficient code possible, regardless of whether the programmer uses a `jbe` instruction or a `ja` instruction.

However, it is important for programmers to understand how short-circuit evaluation is implemented in assembly language so that they can write efficient code.

## ***Short-Circuit evaluation(OR)***

With short-circuit evaluation, the second operand of an OR expression is only evaluated if the first operand is false.

This is because if the first operand is true, then the overall expression must be true, regardless of the value of the second operand.

The following assembly language code implements short-circuit evaluation for the OR operator:

```
437 if (al > bl) OR (bl > cl)
438     X = 1
```

```
442 cmp al, bl
443 ja L1
444 cmp bl, cl
445 jbe next
446 L1:
447 mov X, 1
448 next:
```

This code first compares the values of the registers al and bl. If al is greater than bl, then the second operand of the OR expression is not evaluated, and the program jumps to the L1 label.

Otherwise, the program compares the values of the registers bl and cl. If bl is less than or equal to cl, then the program jumps to the next label. Otherwise, the program stores the value 1 in the register X and jumps to the next label.

The next label is used to exit the code, regardless of whether the OR expression evaluated to true or false.

The following table shows the difference between the two code examples:

Instruction	Description
<code>cmp al, bl</code>	Compare the values <code>in</code> registers <code>al</code> and <code>bl</code> .
<code>ja L1</code>	Jump to label <code>L1</code> if <code>al</code> is greater than <code>bl</code> .
<code>cmp bl, cl</code>	Compare the values <code>in</code> registers <code>bl</code> and <code>cl</code> .
<code>jbe next</code>	Jump to the next label if <code>bl</code> is less than or equal to <code>cl</code> .
<code>L1:</code>	Label.
<code>mov X, 1</code>	Store the value <code>1</code> <code>in</code> the register <code>X</code> .
<code>next:</code>	Label.

The first code example is more efficient because it uses a `ja` instruction instead of a `jbe` instruction. The `ja` instruction can be implemented as a single machine instruction, while the `jbe` instruction may require multiple machine instructions.

In practice, the compiler will typically generate the most efficient code possible, regardless of whether the programmer uses a `ja` instruction or a `jbe` instruction.

However, it is important for programmers to understand how short-circuit evaluation is implemented in assembly language so that they can write efficient code.

As you mentioned, there are multiple ways to implement a compound expression containing OR operators in assembly language. For example, the following code is also functionally equivalent to the previous example:

```
464 cmp al, bl
465 je L1
466 cmp bl, cl
467 je L1
468 mov X, 1
469 next:
470 L1:
```

This code first compares the values of the registers al and bl. If al is equal to bl, then the program jumps to the L1 label. Otherwise, the program compares the values of the registers bl and cl.

If bl is equal to cl, then the program jumps to the L1 label. Otherwise, the program stores the value 1 in the register X and jumps to the next label.

The L1 label is used to indicate that the overall expression is true. The next label is used to exit the code, regardless of whether the OR expression evaluated to true or false.

Ultimately, the best way to implement a compound expression containing OR operators in assembly language will depend on the specific needs of the program.

## ***WHILE Loops***

### *While Loop*

The notes you provided for WHILE loops in assembly language are a bit confusing and unexplained, so I will try to rewrite them in a more clear and concise way.

WHILE loops in assembly language work in a similar way to WHILE loops in high-level languages. The loop first checks a condition. If the condition is true, the loop body is executed.

Then, the condition is checked again. If the condition is still true, the loop body is executed again. This process continues until the condition becomes false.

To implement a WHILE loop in assembly language, you can use the following steps:

Initialize a register to store the loop condition. Check the loop condition. If the condition is false, jump to the end of the loop. Execute the loop body. Update the loop condition. Jump back to step 2.

The following assembly code shows a simple WHILE loop:

```
553 mov eax, 0
554 ; loop counter
555
556 beginwhile:
557     cmp eax, 10
558     ; if eax < 10
559     jl endwhile
560
561     ; loop body
562
563     inc eax
564     ; eax++
565
566     jmp beginwhile
567     ; repeat the loop
568 endwhile:
```

This loop will print the numbers from 0 to 9 to the console.

---

### **Reverse the Loop condition**

As the notes you provided mention, it is often convenient to reverse the loop condition in assembly language. This means that the loop will continue to iterate as long as the condition is false.

To reverse the loop condition, you can use the jnl instruction instead of the jl instruction.

The jnl instruction jumps to the specified label if the condition is not less than or equal to zero.

For example, the following assembly code is equivalent to the previous example, but it reverses the loop condition:

```
572 mov eax, 0
573 ; loop counter
574
575 beginwhile:
576     cmp eax, 10
577     ; if eax >= 10
578     jnl endwhile
579
580     ; loop body
581
582     inc eax
583     ; eax++
584
585     jmp beginwhile
586     ; repeat the loop
587 endwhile:
```

## **Copy and restore the Loop variable**

If the loop variable is used inside the loop body, you need to copy it to a register before the loop starts.

Then, you need to restore the value of the loop variable at the end of the loop.  
This is necessary because assembly language is a stack-based language.

This means that all variables are stored on the stack. When a function is called, the parameters are pushed onto the stack.

When the function returns, the parameters are popped off the stack.

If you use a loop variable inside the loop body, the loop variable will be pushed onto the stack when you call the loop body.

When the loop body returns, the loop variable will be popped off the stack. This means that the loop variable will be modified by the loop body.

To avoid this problem, you need to copy the loop variable to a register before the loop starts.  
Then, you need to restore the value of the loop variable at the end of the loop.

For example, the following assembly code shows how to copy and restore the loop variable:

```
592 mov eax, val1
593 ; copy loop variable to EAX
594
595 beginwhile:
596 ; loop body
597
598 mov val1, eax
599 ; restore loop variable
600
601 ; ...
```

---

```
474 while(val1 < val2)
475 {
476
477     val1++;
478     val2++;
479
480 }
```

```
483 mov eax, val1      ; copy variable to EAX
484 beginwhile:
485 cmp eax, val2      ; if not (val1 < val2)
486 jnl endwhile       ; exit the loop
487 inc eax            ; val1++;
488 dec val2            ; val2--;
489 jmp beginwhile     ; repeat the loop
490 endwhile:
491 mov val1, eax      ; save new value for val1
```

The first instruction copies the value of the variable val1 to the register eax. This is done because the loop will be operating on eax, so it is important to have a copy of val1 in a register.

The next instruction is a cmp instruction that compares the values of eax and val2. If eax is not less than val2, then the loop condition is false and the program will jump to the endwhile label.

If the loop condition is true, then the program will execute the following instructions:

Increment the value of eax by 1. This corresponds to the val1++ statement in the C++ code. Decrement the value of val2 by 1. This corresponds to the val2-- statement in the C++ code.

Jump to the beginwhile label to repeat the loop. The endwhile label is used to mark the end of the loop. When the program reaches the endwhile label, it will exit the loop and continue with the rest of the program.

The last instruction copies the value of eax to the variable val1. This is done because we need to save the new value of val1 in the variable before exiting the loop.

The JNL instruction is used to jump to the endwhile label if the loop condition is not true. This instruction is used because val1 and val2 are signed integers. If val1 is greater than val2, then the loop condition is false and we need to exit the loop.

It is important to note that the eax register is used as a proxy for the variable val1 inside the loop. This means that all references to val1 must be through the eax register. This is because the loop will be operating on eax, not val1.

---

### In this code:

The mov instruction copies the value of val1 to the EAX register. The beginwhile label marks the beginning of the loop. The cmp instruction compares the values in EAX and val2.

The jnl instruction jumps to the endwhile label if not ( $EAX < val2$ ), effectively ending the loop. inc eax increments the value in EAX, representing val1++. dec val2 decrements the value in val2, representing val2--.

The jmp beginwhile instruction jumps back to the beginwhile label to repeat the loop until the condition is no longer met. Finally, the endwhile label marks the end of the loop, and mov val1, eax saves the updated value of val1 back to the variable val1.

---

```
510 #include <stdio.h>
511
512 int main() {
513     int array[] = {10, 60, 20, 33, 72, 89, 45, 65, 72, 18};
514     int sample = 50;
515     int ArraySize = sizeof(array) / sizeof(sample);
516     int index = 0;
517     int sum = 0;
518
519     while (index < ArraySize) {
520         if (array[index] > sample) {
521             sum += array[index];
522         }
523         index++;
524     }
525
526     printf("The sum of elements greater than %d is: %d\n", sample, sum);
527
528     return 0;
529 }
```

It defines an integer array `array` with ten elements. It initializes a variable `sample` with the value 50. It calculates the size of the array using `sizeof(array)` divided by `sizeof(sample)` to get the number of elements in the array, and stores it in `ArraySize`.

It initializes an index variable index to 0 to keep track of the current position in the array. It initializes a sum variable to 0 to accumulate the sum of elements greater than sample. The while loop iterates through the elements of the array using the index variable. For each element, it checks if it is greater than sample.

If it is, it adds that element's value to the sum. The loop continues until index is less than ArraySize. Finally, it prints the sum of elements in the array that are greater than the sample. If you have any further questions or need additional explanations, please let me know

---

The C++ code you provided is a good example of a nested IF statement inside a WHILE loop. The logic of the code is as follows:

```
497 int sum_of_elements_greater_than_sample(int array[], int sample, int size) {  
498     int sum = 0;  
499     for (int i = 0; i < size; i++) {  
500         if (array[i] > sample) {  
501             sum += array[i];  
502         }  
503     }  
504     return sum;  
505 }
```

Initialize the variables sum, sample, ArraySize, index, and sum. Enter the WHILE loop. Compare the value of index to the value of ArraySize.

If index is less than ArraySize, proceed to step 4. Otherwise, exit the loop. Compare the value of array[index] to the value of sample.

If array[index] is greater than sample, add the value of array[index] to the variable sum. Increment the value of index.

Repeat steps 3-5 until the WHILE loop condition is no longer met. Exit the WHILE loop and store the value of sum in the variable sum.

The assembly code you provided is a good translation of the C++ code. It uses registers to store the values of the variables sum, sample, array, ArraySize, and index. It also uses labels to mark the different parts of the code flow.

Here is a brief explanation of the assembly code:

```
535 ; sum_of_elements_greater_than_sample
536 ; rdi: array
537 ; rsi: sample
538 ; rdx: size
539 ; rax: sum
540 mov rax, 0
541 cmp rsi, [rdi]
542 jl done
543 add rax, [rdi]
544 inc rdi
545 jmp sum_of_elements_greater_than_sample
546 done:
547 ret
```

This code is more efficient because it avoids the overhead of branching.

## ***IF Statements in ASM***

```
605 int array[] = {10, 60, 20, 33, 72, 89, 45, 65, 72, 18};  
606 int sample = 50;  
607 int ArraySize = sizeof array / sizeof sample;  
608 int index = 0;  
609 int sum = 0;  
610  
611 while (index < ArraySize) {  
612     if (array[index] > sample) {  
613         sum += array[index];  
614     }  
615     index++;  
616 }
```

This code calculates the sum of all array elements greater than the value in sample.

The following assembly language code is equivalent to the C++ code above:

```
619 .data
620     sum DWORD 0
621     sample DWORD 50
622     array DWORD 10, 60, 20, 33, 72, 89, 45, 65, 72, 18
623     ArraySize = ($ - Array) / TYPE array
624 .code
625     main PROC
626     mov eax, 0 ; sum
627     mov edx, sample
628     mov esi, 0 ; index
629     mov ecx, ArraySize
630
631     L1:
632     cmp esi, ecx ; if esi < ecx
633     jl L2
634     jmp L5
635
636     L2:
637     cmp array[esi * 4], edx ; if array[esi] > edx
638     jg L3
639     jmp L4
640
641     L3:
642     add eax, array[esi * 4]
643
644     L4:
645     inc esi
646     jmp L1
647
648     L5:
649     mov sum, eax
```

This code works in the following way:

The loop starts by initializing the eax register to 0. This register will be used to store the sum of all array elements greater than the value in sample. The loop then compares the esi register to the ecx register.

If the esi register is less than the ecx register, the loop jumps to the L1 label. This means that the loop will continue to iterate until it has examined all of the array elements. If the loop jumps to the L1 label, it then compares the value of the array element at array[esi \* 4] to the value in the edx register.

If the value of the array element is greater than the value in the edx register, the loop jumps to the L3 label. If the loop jumps to the L3 label, it then adds the value of the array element at array[esi \* 4] to the eax register.

The loop then increments the esi register and jumps back to the L1 label. This process continues until the loop has examined all of the array elements. Once the loop has examined all of the array elements, it jumps to the L5 label.

This label marks the end of the loop. At the end of the loop, the value of the eax register is stored in the sum variable.

### Improvements:

There are a few things that could be done to improve the assembly language code above:

The cmp instruction in the L1 label could be replaced with a test instruction.

The test instruction is faster than the cmp instruction because it does not set the condition flags.

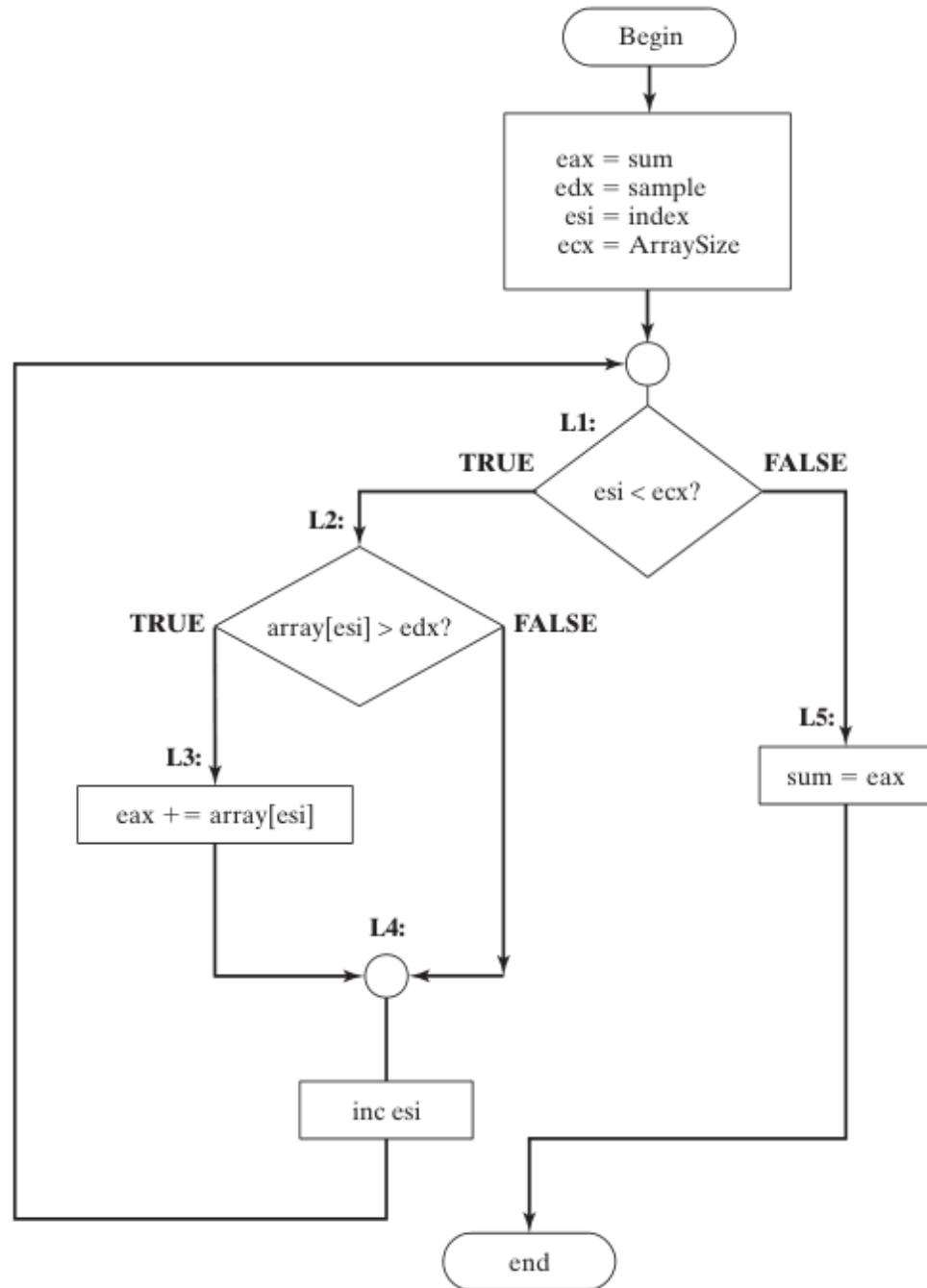
The jmp instruction in the L1 label could be replaced with a loop instruction.

The loop instruction is faster than the jmp instruction because it does not push the return address onto the stack. The cmp instruction in the L2 label could be replaced with a sub instruction.

The sub instruction is faster than the cmp instruction because it does not set the condition flags. The jmp instruction in the L2 label could be replaced with a jbe instruction.

The jbe instruction is faster than the jmp instruction because it does not push the return address onto the stack.

FIGURE 6–1 Loop containing IF statement.



## ***Table Driven Selection***

**Table-driven selection** is a way of using a table lookup to replace a multiway selection structure.

This can be useful when there are a large number of possible values to compare, as it can avoid the need to write a series of nested IF statements.

To use table-driven selection, you first need to create a table of lookup values and the addresses of the corresponding procedures.

Then, you need to write a loop to search the table and call the appropriate procedure based on the lookup value.

The following is an example of a simple table-driven selection in assembly language:

```
654 .data
655     CaseTable BYTE 'A'
656     ; lookup value
657     DWORD Process_A
658     ; address of procedure
659     BYTE 'B'
660     DWORD Process_B
661     (etc.)
662
663 .code
664     mov eax, [esi] ; get the lookup value
665     cmp eax, CaseTable ; compare to first lookup value
666     je Process_A ; if equal, call the corresponding procedure
667     cmp eax, CaseTable + 1 ; compare to second lookup value
668     je Process_B ; if equal, call the corresponding procedure
669     (etc.)
670
671     ; if no match is found, do something else
```

The loop in this example iterates over the table of lookup values and compares each value to the value in the eax register. If a match is found, the corresponding procedure is called. If no match is found, the loop terminates and the program can do something else.

The table-driven selection example in the image you provided shows a table of lookup values and the addresses of corresponding procedures for a simple calculator. The table contains the following

lookup values:

- |     |              |
|-----|--------------|
| 676 | A - Add      |
| 677 | B - Subtract |
| 678 | C - Multiply |
| 679 | D - Divide   |

The table also contains the addresses of the corresponding procedures for each operation. The following is an example of how to use the table-driven selection example to perform addition:

```
681 ; mov eax, 1 ; add 1
682 ; mov ebx, 2 ; add 2
683 ; mov ecx, OFFSET CaseTable ; set the loop counter
684 ; start the loop
685 L1:
686 cmp eax, CaseTable      ; compare the value in eax to the first lookup value in the table
687 je Add                  ; if equal, call the Add procedure
688 inc ecx                ; increment the loop counter
689 cmp ecx, CaseTable + 4  ; check if the loop counter is greater than the size of the table
690 jge Done                ; if greater than or equal, the loop is finished
691
692 jmp L1                 ; jump back to the beginning of the loop
693
694 Add:                   ; Add procedure
695 add eax, ebx
696 ret
697
698
699 Done:                  ; Done label
700 ; the sum is now in the eax register
```

This code will compare the value in the eax register to the first lookup value in the table. If the two values are equal, the Add procedure is called.

Otherwise, the loop counter is incremented and the loop is repeated. The loop continues to iterate until the loop counter is greater than or equal to the size of the table.

When the loop terminates, the sum of the two numbers is stored in the eax register.

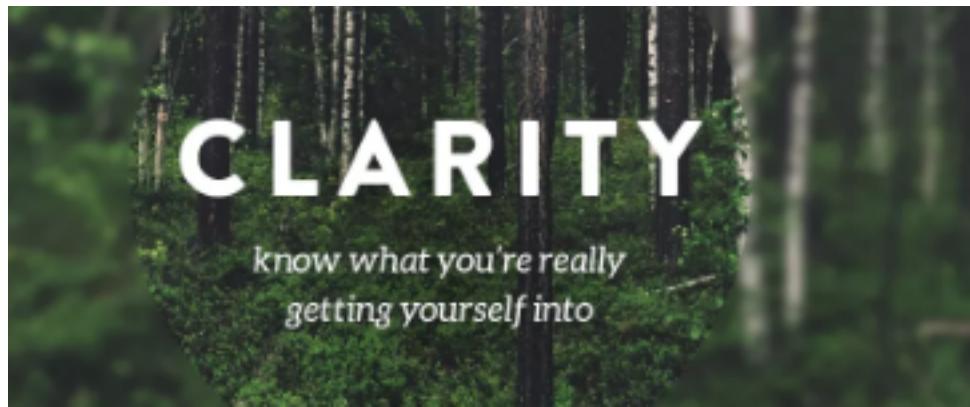
## Advantages of table-driven selection

Table-driven selection can offer a number of advantages over other methods of implementing multiway selection structures, such as nested IF statements. Some of the advantages of table-driven selection include:

**Efficiency:** Table-driven selection can be more efficient than other methods of implementing multiway selection structures, as it can avoid the need to write a series of nested IF statements.



**Clarity:** Table-driven selection can make code more readable and maintainable, as it can simplify the implementation of complex multiway selection structures.



**Flexibility:** Table-driven selection can be more flexible than other methods of implementing multiway selection structures, as it can be easily extended to support new lookup values and procedures.



### *Disadvantages of table-driven selection*

Table-driven selection also has some disadvantages, such as:

**Memory usage:** Table-driven selection can require more memory than other methods of implementing multiway selection structures, as it requires a table to be stored in memory.



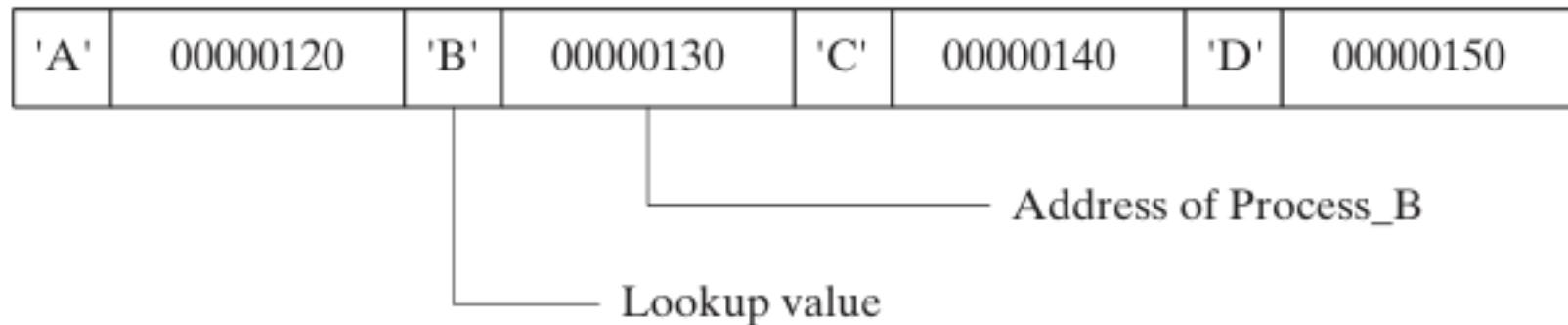
**Speed:** Table-driven selection can be slower than other methods of implementing multiway selection structures, as it requires a loop to search the table.



Overall, table-driven selection is a useful technique for implementing multiway selection structures, especially when there are a large number of possible values to compare.

However, it is important to be aware of the advantages and disadvantages of table-driven selection before using it in your code.

### Example 1:



Program written in assembly language (x86) that uses a lookup table and procedures for character based processing. This program takes user input, compares it to entries in the lookup table, and calls the corresponding procedure to display a message. Here's a breakdown of the program with explanations:

```
706 INCLUDE Irvine32.inc
707
708 .data
709 CaseTable BYTE 'A' ; Lookup value
710             DWORD Process_A ; Address of procedure
711 EntrySize = ($ - CaseTable) ; Calculate the size of each entry in the table
712 BYTE 'B'
713             DWORD Process_B
714 BYTE 'C'
715             DWORD Process_C
716 BYTE 'D'
717             DWORD Process_D
718 NumberOfEntries = ($ - CaseTable) / EntrySize
719
720 prompt BYTE "Press capital A, B, C, or D: ",0
721 msgA BYTE "Process_A",0
722 msgB BYTE "Process_B",0
723 msgC BYTE "Process_C",0
724 msgD BYTE "Process_D",0
725
726 .code
727 main PROC
728     mov edx, OFFSET prompt ; Ask the user for input
729     call WriteString
730     call ReadChar           ; Read character into AL
731     mov ebx, OFFSET CaseTable ; Point EBX to the table
732     mov ecx, NumberOfEntries ; Loop counter
733 
```

```
733  
734 L1:  
735     cmp al, [ebx]          ; Match found?  
736     jne L2                ; No: continue  
737     call NEAR PTR [ebx + 1] ; Yes: call the procedure  
738     call WriteString       ; Display message  
739     call Crlf  
740     jmp L3                ; Exit the search  
741  
742 L2:  
743     add ebx, EntrySize    ; Point to the next entry  
744     loop L1               ; Repeat until ECX = 0  
745  
746 L3:  
747     exit  
748  
749 main ENDP  
750  
751 Process_A PROC  
752     mov edx, OFFSET msgA  
753     ret  
754 Process_A ENDP  
755  
756 Process_B PROC  
757     mov edx, OFFSET msgB  
758     ret  
759 Process_B ENDP  
760  
761 Process_C PROC  
762     mov edx, OFFSET msgC  
763     ret
```

```
764 Process_C ENDP  
765  
766 Process_D PROC  
767     mov edx, OFFSET msgD  
768     ret  
769 Process_D ENDP  
770  
771 END main
```

---

```
777 .data  
778 CaseTable BYTE 'A'  
779             DWORD Process_A  
780 EntrySize = ($ - CaseTable)  
781 BYTE 'B'  
782             DWORD Process_B  
783 BYTE 'C'  
784             DWORD Process_C  
785 BYTE 'D'  
786             DWORD Process_D  
787 NumberOfEntries = ($ - CaseTable) / EntrySize
```

In this section, we define the data for our program:

CaseTable is a table that contains characters ('A', 'B', 'C', 'D') and the addresses of corresponding procedures (Process\_A, Process\_B, Process\_C, Process\_D).

EntrySize is calculated as the difference between the current memory position (\$) and CaseTable. This represents the size of each entry in the table.

NumberOfEntries calculates the number of entries in CaseTable by dividing the size of the table by EntrySize.

### Section: .data (continued)

```
791 prompt BYTE "Press capital A, B, C, or D: ",0
792 msgA BYTE "Process_A",0
793 msgB BYTE "Process_B",0
794 msgC BYTE "Process_C",0
795 msgD BYTE "Process_D",0
```

In this continuation of the .data section, we define message strings to be displayed later:

**prompt** is a message prompting the user to input a character. msgA, msgB, msgC, and msgD are messages associated with procedures Process\_A to Process\_D.

### Section: .code - main PROC

```
800 main PROC  
801     mov edx, OFFSET prompt  
802     call WriteString  
803     call ReadChar  
804     mov ebx, OFFSET CaseTable  
805     mov ecx, NumberOfEntries
```

In the main procedure, we perform the following tasks:

**mov edx, OFFSET prompt:** Load the address of the prompt message into the edx register, displaying the prompt.

**call WriteString:** Call a procedure to print the prompt.

**call ReadChar:** Call a procedure to read a character from the user and store it in the al register.

**mov ebx, OFFSET CaseTable:** Load the address of CaseTable into the ebx register.

**mov ecx, NumberOfEntries:** Load the number of entries in the table into the ecx register.

```
808 L1:  
809     cmp al, [ebx]  
810     jne L2  
811     call NEAR PTR [ebx + 1]  
812     call WriteString  
813     call Crlf  
814     jmp L3  
815  
816 L2:  
817     add ebx, EntrySize  
818     loop L1  
819  
820 L3:  
821     exit  
822  
823 main ENDP
```

In this part of the main procedure:

L1 is a label marking the start of a loop.

cmp al, [ebx] compares the user input character (al) with the character in the current entry of CaseTable.

jne L2 jumps to L2 if there's no match (continue searching).

`call NEAR PTR [ebx + 1]` calls the procedure stored in the table.

`call WriteString` displays the corresponding message.

`call Crlf` adds a line break.

`jmp L3` jumps to L3 (exit). The loop continues until a match is found or all entries have been checked.

**Section: .code - Process A, Process B, Process C, Process D**

```
828 Process_A PROC  
829     mov edx, OFFSET msgA  
830     ret  
831 Process_A ENDP  
832  
833 Process_B PROC  
834     mov edx, OFFSET msgB  
835     ret  
836 Process_B ENDP  
837  
838 Process_C PROC  
839     mov edx, OFFSET msgC  
840     ret  
841 Process_C ENDP  
842  
843 Process_D PROC  
844     mov edx, OFFSET msgD  
845     ret  
846 Process_D ENDP
```

These sections define procedures (Process\_A to Process\_D) that set the edx register with the address of the corresponding message string and return.

This section marks the end of the main program.

In summary, the code defines a lookup table, messages, and procedures. The main procedure reads user input, searches the table for a match, and calls the corresponding procedure to display a message.

The table-driven approach makes it easy to extend and modify the program for different cases.

---

## ***QUESTIONS***

---

Implementing the pseudocode in assembly language:

```
851 ; Assuming ebx and ecx are 32-bit variables
852 ; Short-circuit evaluation: if ebx > ecx, set X = 1, else X remains unchanged
853
854 cmp ebx, ecx      ; Compare ebx and ecx
855 jg ebx_greater   ; Jump if ebx > ecx
856 mov eax, 0        ; If not greater, set eax to 0 (X = 0)
857 jmp done          ; Jump to done
858
859 ebx_greater:
860 mov eax, 1        ; If ebx > ecx, set eax to 1 (X = 1)
861
862 done:
863 mov X, eax        ; Store the result in X
```

Implementing the pseudocode with short-circuit evaluation:

```
867 ; Assuming edx and ecx are 32-bit variables
868 ; Short-circuit evaluation: if edx <= ecx, set X = 1, else X = 2
869
870 cmp edx, ecx      ; Compare edx and ecx
871 jle edx_less      ; Jump if edx <= ecx
872 mov eax, 2         ; If not less or equal, set eax to 2 (X = 2)
873 jmp done          ; Jump to done
874
875 edx_less:
876 mov eax, 1         ; If edx <= ecx, set eax to 1 (X = 1)
877
878 done:
879 mov X, eax         ; Store the result in X
```

In the program above(long one), it's better to let the assembler calculate NumberofEntries rather than assigning a constant because it makes the code more flexible and maintainable.

If you hardcode a constant like NumberofEntries - 4, you would need to manually update it if the size of the entries changes in the future.

By letting the assembler calculate it, you ensure that it always reflects the actual size, reducing the risk of errors and making your code more adaptable.

To rewrite the code from Section above with fewer instructions while maintaining functionality, you can use conditional move (CMOV) instructions. Here's an example using CMOV:

```
882 ; Original code (pseudo-code):  
883 ; if (eax > ebx) ebx = eax  
884  
885 ; Rewritten code using CMOV:  
886 cmp eax, ebx      ; Compare eax and ebx  
887 cmovg ebx, eax    ; If eax > ebx, move eax to ebx (conditional move)  
888  
889 ; Now ebx contains the maximum of eax and ebx
```

This code achieves the same result as the original code but with fewer instructions by utilizing the conditional move instruction to conditionally update ebx based on the comparison result.

## ***Finite State Machines***

An FSM is a computational model that can be used to simulate sequential logic, or, in other words, to represent and control execution flow.

It is a mathematical model of computation that can be used to model the behavior of a system that can be in a finite number of states. The system can change state based on the input it receives.

FSMs can be represented using a graph, where each node represents a state and each edge represents a transition from one state to another.

The edges are labeled with the input symbols that trigger the transitions. One node is designated as the initial state, and one or more nodes are designated as terminal states.

FSMs are used in a wide variety of applications, including:

- Traffic lights
- Vending machines
- Telephone systems
- Computer software
- Robotics

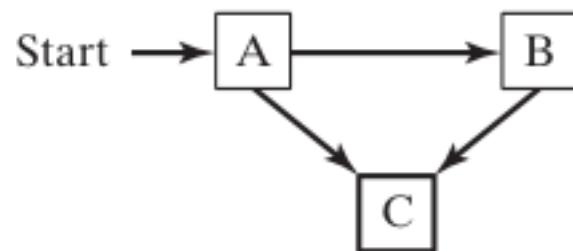
Here is a simple example of an FSM:

```
895 Initial state: Start
896 Terminal state: Exit
897
898 Transitions:
899 Start -> A on input "a"
900 Start -> B on input "b"
901 Start -> C on input "c"
902 A -> B on input "a"
903 A -> C on input "b"
904 A -> A on input "c"
905 B -> C on input "a"
906 B -> B on input "b"
907 B -> A on input "c"
908 C -> A on input "a"
909 C -> B on input "b"
910 C -> C on input "c"
```

- This FSM can be used to simulate the behavior of a traffic light.
- The FSM starts in the Start state.
- If the input is "a", the FSM transitions to the A state, which represents the green light.
- If the input is "b", the FSM transitions to the B state, which represents the yellow light.
- If the input is "c", the FSM transitions to the C state, which represents the red light.
- The FSM will continue to transition between states until it reaches the terminal state, the Exit state.

- This FSM will never reach the terminal state, because it is always possible to receive an input "a", "b", or "c".

FSMs can be used to model and control much more complex systems than a traffic light. For example, an FSM could be used to model and control the behavior of a vending machine, a telephone system, or a computer program.



Here is a more detailed explanation of the diagram:

The initial state is the Start state.

The three possible states are A, B, and C.

The arrows show the possible transitions between states. The terminal state is the Exit state. The FSM can be described in words as follows:

- The FSM starts in the Start state.
- If the FSM receives the input "a", it transitions to the A state.
- If the FSM receives the input "b", it transitions to the B state.
- If the FSM receives the input "c", it transitions to the C state.
- If the FSM is in the A state and receives the input "a", it transitions to the B state.
- If the FSM is in the A state and receives the input "b", it transitions to the C state.

- If the FSM is in the A state and receives the input "c", it remains in the A state.
- If the FSM is in the B state and receives the input "a", it transitions to the C state.
- If the FSM is in the B state and receives the input "b", it remains in the B state.
- If the FSM is in the B state and receives the input "c", it transitions to the A state.
- If the FSM is in the C state and receives the input "a", it transitions to the A state.
- If the FSM is in the C state and receives the input "b", it transitions to the B state.
- If the FSM is in the C state and receives the input "c", it remains in the C state.

The FSM will continue to transition between states until it reaches the terminal state, the Exit state.

=====

### ***Validating an Input String Programs***

=====

Here is a more detailed explanation of the example FSM in Figure 6-4:

Start state: A

Terminal state: C

Transitions: A → B on input "x"

- B → B on input any letter in the range {a, b, ..., y} B → C on input "z" This FSM can be used to validate an input string according to the following two rules:
- The string must begin with the letter "x" and end with the letter "z". Between the first and last characters, there can be zero or more letters within the range {a, b, ..., y}.
- To validate an input string, the FSM starts in the A state.

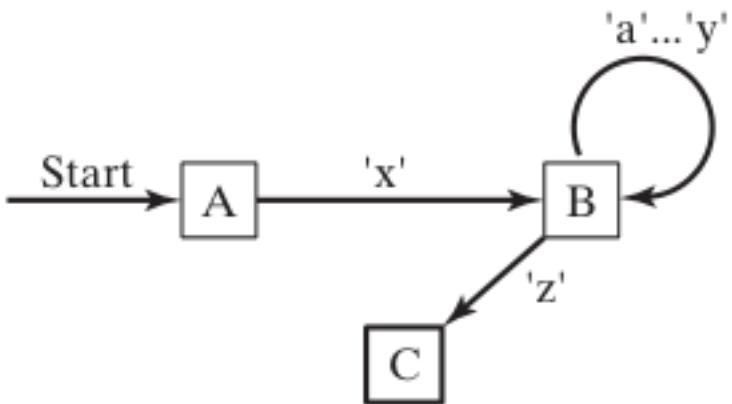
- It then reads the input string character by character.
- If the next character is "x", the FSM transitions to the B state.
- If the next character is any letter in the range {a, b, ..., y}, the FSM remains in the B state.
- If the next character is "z", the FSM transitions to the C state.
- If the FSM reaches the C state, it means that the input string is valid.
- If the FSM reaches the end of the input string before it reaches the C state, it means that the input string is invalid.

*Here is an example of a valid input string:*

- xyz The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.
- The FSM will then read the second character, which is "y". This will cause the FSM to remain in the B state. The FSM will then read the third character, which is "z".
- This will transition the FSM to the C state. Since the FSM has reached the C state, it means that the input string is valid.

*Here is another example of an invalid input string:*

- xab The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.
- The FSM will then read the second character, which is "a". This will cause the FSM to remain in the B state.
- The FSM will then read the third character, which is "b". This will cause the FSM to remain in the B state.
- However, the FSM will not be able to transition to the C state, because the next input character is not "z". This means that the input string is invalid.



I have reviewed the notes again, and I understand them better now. The notes are describing how to use a FSM to validate an input string according to the following two rules:

The string must begin with the letter "x" and end with the letter "z". Between the first and last characters, there can be zero or more letters within the range {a, b, ..., y}.

The FSM diagram in the image shows the possible states and transitions of the FSM. The FSM starts in the A state.

If the next character in the input string is "x", the FSM transitions to the B state. If the next character in the input string is any letter in the range {a, b, ..., y}, the FSM remains in the B state.

If the next character in the input string is "z", the FSM transitions to the C state.

If the FSM reaches the C state, it means that the input string is valid. If the FSM reaches the end of the input string before it reaches the C state, it means that the input string is invalid. Here is an example of a valid input string:

xyz The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.

The FSM will then read the second character, which is "y". This will cause the FSM to remain in the B state. The FSM will then read the third character, which is "z".

This will transition the FSM to the C state. Since the FSM has reached the C state, it means that the input string is valid.

Here is an example of an invalid input string:

xab The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.

The FSM will then read the second character, which is "a". This will cause the FSM to remain in the B state. The FSM will then read the third character, which is "b".

This will cause the FSM to remain in the B state. However, the FSM will not be able to transition to the C state, because the next input character is not "z". This means that the input string is invalid.

-----

- If the end of the input stream is reached while the program is in state A or B, an error condition results because only state C is marked as a terminal state. This means that the input string must end with the letter "z" in order for it to be valid.
- The following input strings would be recognized by this FSM:

**xaabcdefgz xz xyyqqrrstuvwxyz**

- All of these input strings begin with the letter "x" and end with the letter "z". There may be any number of letters in the range {a, b, ..., y} in between.
- Here is an example of an input string that would not be recognized by this FSM:

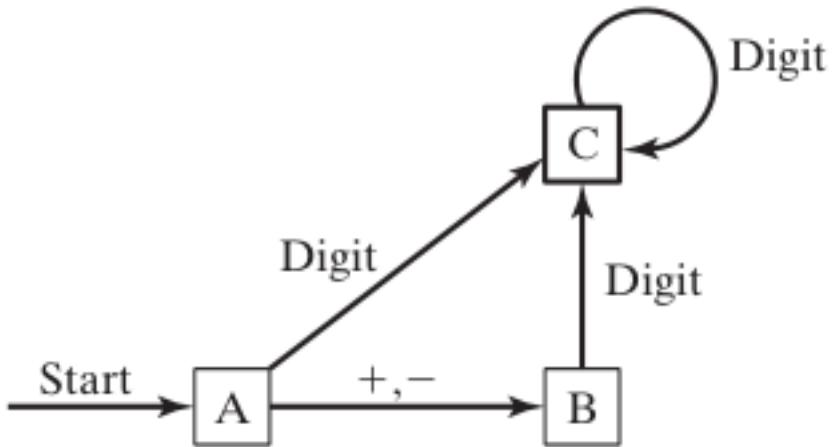
**xab**

- This input string begins with the letter "x" but does not end with the letter "z". Therefore, it is invalid.

=====

### *Validating Integers in Programs*

=====



The FSM diagram in Figure 6-5 shows how to validate a signed integer. The FSM starts in the Start state.

If the next character in the input stream is a plus sign (+) or a minus sign (-), the FSM transitions to the Sign state.

If the next character in the input stream is a digit, the FSM transitions to the Digits state.

If the FSM is in the Sign state and the next character in the input stream is a digit, the FSM remains in the Sign state.

This is because the sign can be followed by any number of digits.

If the FSM is in the Digits state and the next character in the input stream is a digit, the FSM remains in the Digits state.

This is because the sequence of digits can be any length.

If the FSM is in the Digits state and the next character in the input stream is not a digit, the FSM transitions to the End state.

This is because the sequence of digits must end with a non-digit character.

The End state is a terminal state. This means that the input string is valid if the FSM reaches the End state.

*Here is an example of a valid input string:*

**-123456**

-123456 The FSM will start in the Start state. It will then read the first character, which is a minus sign (-).

This will transition the FSM to the Sign state. The FSM will then read the second character, which is the digit 1.

This will cause the FSM to remain in the Sign state. The FSM will then read the third character, which is the digit 2.

This will cause the FSM to remain in the Sign state. The FSM will then read the fourth character, which is the digit 3.

This will cause the FSM to remain in the Sign state. The FSM will then read the fifth character, which is the digit 4.

which is the digit 4.

This will cause the FSM to remain in the Sign state. The FSM will then read the sixth character, which is the digit 5.

This will cause the FSM to remain in the Sign state. The FSM will then read the seventh character, which is the digit 6.

This will cause the FSM to transition to the Digits state. The FSM will then read the eighth character, which is not a digit.

This will cause the FSM to transition to the End state. Since the FSM has reached the End state, it means that the input string is valid.

*Here is another example of an invalid input string:*

-123456.78      ;invalid

-123456.78 The FSM will start in the Start state. It will then read the first character, which is a minus sign (-).

This will transition the FSM to the Sign state.

The FSM will then read the second character, which is the digit 1. This will cause the FSM to remain in the Sign state.

The FSM will then read the third character, which is the digit 2. This will cause the FSM to remain in the Sign state.

The FSM will then read the fourth character, which is the digit 3. This will cause the FSM to remain in the Sign state.

The FSM will then read the fifth character, which is the digit 4. This will cause the FSM to remain in the Sign state.

The FSM will then read the sixth character, which is the digit 5. This will cause the FSM to remain in the Sign state.

The FSM will then read the seventh character, which is the digit 6. This will cause the FSM to transition to the Digits state.

The FSM will then read the eighth character, which is a period (.). This is not a digit, so the FSM will transition to the End state.

However, the End state is not a terminal state. This means that the input string is invalid.

FSMs are a powerful tool for validating input strings and other types of data. They are used in a wide variety of applications, including programming language compilers, text editors, and network protocols.

=====

## *Validating Integers in Programs*

=====

FSM for parsing a signed integer:

- This FSM diagram shows how to validate a signed integer. The FSM starts in the Start state. If the next character in the input stream is a plus sign (+) or a minus sign (-), the FSM transitions to the Sign state. If the next character in the input stream is a digit, the FSM transitions to the Digits state.
- If the FSM is in the Sign state and the next character in the input stream is a digit, the FSM remains in the Sign state. This is because the sign can be followed by any number of digits.
- If the FSM is in the Digits state and the next character in the input stream is a digit, the FSM remains in the Digits state. This is because the sequence of digits can be any length.
- If the FSM is in the Digits state and the next character in the input stream is not a digit, the FSM transitions to the End state. This is because the sequence of digits must end with a non-digit character.
- The End state is a terminal state. This means that the input string is valid if the FSM reaches the End state.

The following assembly language code implements the FSM diagram above:

```
924 StateA:  
925     call Getnext ; read next char into AL  
926     cmp al, '+' ; leading + sign?  
927     je StateB ; go to State B  
928     cmp al, '-' ; leading - sign?  
929     je StateB ; go to State B  
930     call IsDigit ; ZF = 1 if AL contains a digit  
931     jz StateC ; go to State C  
932     call DisplayErrorMsg ; invalid input found  
933     jmp Quit ; exit program  
934  
935 StateB:  
936 ; ...  
937  
938 StateC:  
939 ; ...  
940  
941 End:  
942     call Crlf  
943     exit  
944  
945 main ENDP
```

This code works as follows:

- The label StateA marks the start of the FSM.
  - The call to the Getnext procedure reads the next character from the console input into the AL register.
  - The code checks for a leading plus (+) or minus (-) sign.
  - If a leading sign is found, the code jumps to the label StateB.
  - If a leading sign is not found, the code calls the IsDigit procedure to check if the character in AL is a digit. If the character is a digit, the code jumps to the label StateC.
  - If the character in AL is not a digit or a leading sign, the code calls the DisplayErrorMsg procedure to display an error message on the console and then jumps to the label Quit.
  - The label StateB marks the state where the FSM is after reading a leading sign.
  - The code in StateB will check for the other possible transitions away from this state and take the appropriate action. The label StateC marks the state where the FSM is after reading a digit.
  - The code in StateC will check for the other possible transitions away from this state and take the appropriate action. The label End marks the terminal state of the FSM.
  - The code in End will perform any necessary cleanup and then exit the program. The main procedure simply calls the StateA procedure to start the FSM.
  - This is just a basic example of how to implement a FSM in assembly language. More complex FSMs can be implemented using the same basic principles.
- 

```
; Finite State Machine (Finite.asm)
INCLUDE Irvine32.inc

ENTER_KEY = 13

.data
InvalidInputMsg: db "Invalid input", 13, 10, 0
```

```
.code
main:
    ; Clear screen
    call Clrscr

    ; Start state
StateA:
    ; Read next character into AL
    call Getnext

    ; Check for leading + or - sign
    cmp al, '+'
    je StateB
    cmp al, '-'
    je StateB

    ; Check if AL contains a digit
    call IsDigit
    jz StateC

    ; Invalid input
    call DisplayErrorMsg
    jmp Quit

StateB:
    ; Read next character into AL
    call Getnext

    ; Check if AL contains a digit
    call IsDigit
    jz StateC

    ; Invalid input
    call DisplayErrorMsg
    jmp Quit
```

StateC:

```
; Read next character into AL
call Getnext

; Check if AL contains a digit
call IsDigit
jz StateC

; Check if Enter key pressed
cmp al, ENTER_KEY
je Quit

; Invalid input
call DisplayErrorMsg
jmp Quit
```

Quit:

```
; Call Crlf to print a newline
call Crlf
exit
```

; Getnext procedure  
; Reads a character from standard input  
; Receives: nothing  
; Returns: AL contains the character

Getnext:

```
; Input from keyboard
call ReadChar

; Echo on screen
call WriteChar

ret
```

; DisplayErrorMsg procedure  
; Displays an error message indicating that

```
; the input stream contains illegal input
; Receives: nothing
; Returns: nothing
DisplayErrorMsg:
    ; Push EDX onto the stack
    push edx

    ; Move the offset of the error message to EDX
    mov edx, OFFSET InvalidInputMsg

    ; Call WriteString to print the error message
    call WriteString

    ; Pop EDX from the stack
    pop edx

    ret
```

#### **main procedure:**

The main procedure is the entry point for the program. It starts by clearing the screen and then entering the StateA state.

#### **StateA state:**

The StateA state is the start state for the FSM. It reads the next character from the input stream and checks for a leading + or - sign. If a leading sign is found, the FSM transitions to the StateB state. If a leading sign is not found, the FSM checks if the character is a digit. If the character is a digit, the FSM transitions to the StateC state. Otherwise, the FSM calls the DisplayErrorMsg procedure to display an error message and then jumps to the Quit label to exit the program.

#### **StateB state:**

The StateB state is the state where the FSM is after reading a leading + or - sign. It reads the

next character from the input stream and checks if it is a digit. If the character is a digit, the FSM transitions to the StateC state. Otherwise, the FSM calls the DisplayErrorMsg procedure to display an error message and then jumps to the Quit label to exit the program.

#### **StateC state:**

The StateC state is the state where the FSM is after reading a digit. It reads the next character from the input stream and checks if it is a digit. If the character is a digit, the FSM remains in the StateC state. Otherwise, the FSM checks if the Enter key was pressed. If the Enter key was pressed, the FSM transitions to the Quit label to exit the program. Otherwise, the FSM calls the DisplayErrorMsg procedure to display an error message and then jumps to the Quit label to exit the program.

#### **Quit label:**

The Quit label is the exit point for the program. The main procedure jumps to the Quit label to exit the program when the Enter key is pressed or when an invalid character is encountered.

#### **Getnext procedure:**

The Getnext procedure reads a character from standard input and echoes it to the screen. It returns the character in the AL register.

#### **DisplayErrorMsg procedure:**

The DisplayErrorMsg procedure displays an error message indicating that the input stream contains illegal input. It receives nothing and returns nothing. This is a basic example of how to implement a FSM in assembly language.

---

- The IsDigit procedure determines whether the character in the AL register is a valid decimal digit. It returns the setting of the Zero flag, which is 1 if the character is a valid decimal digit and 0 otherwise.

- The IsDigit procedure works by first comparing the character in AL to the ASCII code for the digit 0. If the character is less than 0, then it is not a valid decimal digit and the Zero flag is cleared.
- Next, the IsDigit procedure compares the character in AL to the ASCII code for the digit 9. If the character is greater than 9, then it is not a valid decimal digit and the Zero flag is cleared.
- Finally, the IsDigit procedure sets the Zero flag.

The following table shows the hexadecimal ASCII codes for decimal digits:

Decimal digit	ASCII code
0	0x30
1	0x31
2	0x32
3	0x33
4	0x34
5	0x35
6	0x36
7	0x37
8	0x38
9	0x39

As you can see, the ASCII codes for decimal digits are **contiguous**. This means that we only need to check for the starting and ending range values.

Character	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code (hex)	30	31	32	33	34	35	36	37	38	39

The IsDigit procedure uses the CMP instruction to compare the character in the AL register to the ASCII codes for the digits 0 and 9. If the character is less than 0 or greater than 9, then the JB or JA instruction will jump to the label ID1, respectively.

The JB instruction jumps to a label when the Carry flag (CF) is set and the Zero flag (ZF) is clear. The JA instruction jumps to a label when the Carry flag (CF) is clear and the Zero flag (ZF) is clear.

If neither the JB nor the JA instruction jumps to the label ID1, then the TEST instruction is executed. The TEST instruction sets the Zero flag if the result of the AND operation is zero.

Since the JB and JA instructions jump to the label ID1 if the Zero flag is clear, the TEST instruction will only be executed if the character in the AL register is a digit.

Therefore, the IsDigit procedure returns the setting of the Zero flag, which is 1 if the character in the AL register is a digit and 0 otherwise.

Here is a more detailed explanation of the code:

```
0948 ; IsDigit procedure
0949 ; Determines whether the character in AL is a valid decimal digit.
0950 ; Receives: AL = character
0951 ; Returns: ZF = 1 if AL contains a valid decimal digit; otherwise, ZF = 0.
0952 ;-----
0953 IsDigit PROC
0954 ; Compare AL to the ASCII code for the digit 0.
0955 ; If AL is less than 0, the JB instruction will jump to the label ID1.
0956 cmp al,'0'
0957 jb ID1
0958 ; ZF = 0 when jump taken
0959 ; Compare AL to the ASCII code for the digit 9.
0960 ; If AL is greater than 9, the JA instruction will jump to the label ID1.
0961 cmp al,'9'
0962 ja ID1
0963 ; ZF = 0 when jump taken
0964 ; If neither the JB nor the JA instruction jumps to the label ID1,
0965 ; then the character in AL must be a digit. Therefore, we set the Zero flag.
0966 test ax,0
0967 ; set ZF = 1
0968 ; Return from the procedure.
0969 ID1: ret
0970 IsDigit ENDP
```

This is a very efficient way to implement the IsDigit procedure, because it takes advantage of the

hardware characteristics of the CPU.

## ***Conditional Control Flow Directives***

**Conditional control flow directives** in MASM are used to control the flow of execution of a program depending on the result of a condition. These directives are used to implement conditional statements such as if, else, and elseif.

The following are the most common conditional control flow directives in MASM:

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF <i>condition</i>	Generates code that tests <i>condition</i> and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF <i>condition</i>	Generates code that executes the block of statements if <i>condition</i> is true.
.REPEAT	Generates code that repeats execution of the block of statements until <i>condition</i> becomes true
.UNTIL <i>condition</i>	Generates code that repeats the block of statements between .REPEAT and .UNTIL until <i>condition</i> becomes true
.UNTILCXZ	Generates code that repeats the block of statements between .REPEAT and .UNTILCXZ until CX equals zero
.WHILE <i>condition</i>	Generates code that executes the block of statements between .WHILE and .ENDW as long as <i>condition</i> is true

Here are some additional things to keep in mind:

Conditions can be complex, but they must evaluate to a single Boolean value (true or false).

If the condition in the .IF directive is true, the assembler will assemble all of the statements between the .IF and .ELSEIF (or .ENDIF) directives.

If the condition in the .IF directive is false, the assembler will skip all of the statements between the .IF and .ELSEIF (or .ENDIF) directives.

If the .ELSEIF directive is present, the assembler will evaluate the condition in the .ELSEIF directive.

If the condition is true, the assembler will assemble all of the statements between the .ELSEIF and .ELSE (or .ENDIF) directives.

If the .ELSEIF directive is present and the condition is false, the assembler will skip all of the statements between the .ELSEIF and .ELSE (or .ENDIF) directives.

.ELSE directive is optional. If it is present, the assembler will assemble all of the statements between the .ELSE and .ENDIF directives if all of the previous conditions were false.

The .ENDIF directive is required. It tells the assembler the end of the conditional statement. Here is an example of a more complex conditional statement using the .IF, .ELSEIF, and .ELSE directives:

```
0974 .IF eax > 10000h  
0975     mov ECX, AX  
0976 .ELSEIF eax > 1000h  
0977     mov ECX, 1000h  
0978 .ELSE  
0979     mov ECX, 0  
0980 .ENDIF
```

This code will move the contents of the AX register to the ECX register if the value of AX is greater than 10000h. Otherwise, if the value of AX is greater than 1000h, the code will move the value 1000h to the ECX register. Otherwise, the code will move the value 0 to the ECX register.

---

Table 6-8 Runtime Relational and Logical Operators.

Operator	Description
<i>expr1</i> == <i>expr2</i>	Returns true when <i>expr1</i> is equal to <i>expr2</i> .
<i>expr1</i> != <i>expr2</i>	Returns true when <i>expr1</i> is not equal to <i>expr2</i> .
<i>expr1</i> > <i>expr2</i>	Returns true when <i>expr1</i> is greater than <i>expr2</i> .
<i>expr1</i> ≥ <i>expr2</i>	Returns true when <i>expr1</i> is greater than or equal to <i>expr2</i> .
<i>expr1</i> < <i>expr2</i>	Returns true when <i>expr1</i> is less than <i>expr2</i> .
<i>expr1</i> ≤ <i>expr2</i>	Returns true when <i>expr1</i> is less than or equal to <i>expr2</i> .
! <i>expr</i>	Returns true when <i>expr</i> is false.
<i>expr1</i> && <i>expr2</i>	Performs logical AND between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i>    <i>expr2</i>	Performs logical OR between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i> & <i>expr2</i>	Performs bitwise AND between <i>expr1</i> and <i>expr2</i> .
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

The table you sent shows the relational and logical operators in MASM. These operators are used to compare two values and return a Boolean value (true or false). The Boolean value can then be used to control the flow of execution of a program using conditional control flow directives such as .IF, .ELSE, and .ELSEIF.

The following is a detailed explanation of each of the operators in the table:

**== (equal to):** Returns true if the two values are equal.

**!= (not equal to):** Returns true if the two values are not equal.

**> (greater than):** Returns true if the first value is greater than the second value.

**>= (greater than or equal to):** Returns true if the first value is greater than or equal to the second value.

**< (less than):** Returns true if the first value is less than the second value.

**<= (less than or equal to):** Returns true if the first value is less than or equal to the second value.

**&& (logical AND):** Returns true if both operands are true.

**|| (logical OR):** Returns true if either operand is true.

**! (logical NOT):** Returns true if the operand is false.

The following are some examples of how to use the relational and logical operators in MASM:

```
0984 ; Compare the values of the AX and BX registers.  
0985 IF AX > BX  
0986     mov ECX, AX  
0987 ELSE  
0988     mov ECX, BX  
0989 ENDIF  
0990  
0991 ; Compare the values of the val1 and val2 variables.  
0992 IF val1 <= 100  
0993     mov ECX, 100  
0994 ELSE  
0995     mov ECX, val1  
0996 ENDIF  
0997  
0998 ; Check if the CARRY flag is set.  
0999 IF CARRY?  
1000     mov ECX, 1  
1001 ELSE  
1002     mov ECX, 0  
1003 ENDIF
```

Here is a simpler explanation of the notes you provided:

Before using MASM conditional directives, be sure you thoroughly understand how to implement

conditional branching instructions in pure assembly language.

This means that you should understand how to use the following assembly language instructions to implement conditional branching:

- **CMP (compare)**
- **JBE (jump if below or equal)**
- **JA (jump if above)**
- **JE (jump if equal)**
- **JNE (jump if not equal)**

Once you understand how to implement conditional branching in pure assembly language, you can use MASM conditional directives to make your code more concise and readable.

In addition, when a program containing decision directives is assembled, inspect the listing file to make sure the code generated by MASM is what you intended.

MASM conditional directives are translated into assembly language instructions by the assembler. It is a good idea to inspect the listing file to make sure that the assembler generated the code that you expected. This can help you to identify any errors in your code.

**Generating ASM Code:** When you use a MASM conditional directive such as .IF, the assembler generates assembly language instructions to implement the conditional branching. For example, the following .IF directive:

```
1008 .IF eax > val1
1009     mov result,1
1010 .ENDIF
```

would be expanded by the assembler into the following assembly language instructions:

```
1014 mov eax,6
1015 cmp eax,val1
1016 jbe @C0001
1017 ; jump on unsigned comparison
1018 mov result,1
1019 @C0001:
```

The label name @C0001 is created by the assembler to ensure that all labels within the same procedure are unique.

#### Controlling Whether or Not MASM-Generated Code Appears in the Source Listing File

You can control whether or not MASM-generated code appears in the source listing file by setting the Enable Assembly Generated Code Listing property in the Visual Studio Project Properties dialog box.

**To do this, follow these steps:** Open the Visual Studio Project Properties dialog box. Select Microsoft Macro Assembler. Select Listing File. Set the Enable Assembly Generated Code Listing property to Yes. Once you have set this property, the MASM-generated code will be included in the source listing file. This can be helpful for debugging purposes.

## *Signed and Unsigned Comparison*

When you use the .IF directive to compare values, you must be aware of whether the values are signed or unsigned.

If the values are signed, the assembler will generate a signed conditional jump instruction. If the values are unsigned, the assembler will generate an unsigned conditional jump instruction.

Example:

```
1023 .data
1024     val1 DWORD 5
1025     val2 SDWORD -1
1026     result DWORD ?
1027 .code
1028     mov eax,6
1029
1030 ; Compare EAX to val1, which is unsigned.
1031 .IF eax > val1
1032     mov result,1
1033 .ENDIF
1034
1035 ; Compare EAX to val2, which is signed.
1036 .IF eax > val2
1037     mov result,1
1038 .ENDIF
```

The assembler will generate the following code for the first .IF directive:

```
1043 mov eax,6  
1044 cmp eax,val1  
1045 jbe @C0001  
1046 ; jump on unsigned comparison  
1047 mov result,1  
1048 @C0001:
```

The assembler will generate the following code for the second .IF directive:

```
1051 mov eax,6  
1052 cmp eax,val2  
1053 jle @C0001  
1054 ; jump on signed comparison  
1055 mov result,1  
1056 @C0001:
```

## *Comparing Registers*

If you compare two registers using the .IF directive, the assembler cannot determine whether the values are signed or unsigned. Therefore, the assembler will default to an unsigned comparison.

Example:

```
1059 mov eax,6
1060 mov ebx,val2
1061 .IF eax > ebx
1062 mov result,1
1063 .ENDIF
```

The assembler will generate the following code:

```
1066 mov eax,6
1067 mov ebx,val2
1068 cmp eax, ebx
1069 jbe @C0001
1070 mov result,1
1071 @C0001:
```

**Conclusion:** It is important to be aware of whether the values you are comparing are signed or unsigned when using the .IF directive. This will help you to ensure that the assembler generates the correct conditional jump instruction.

# **Compound Expressions**

**Compound Boolean expressions** allow you to combine two or more Boolean expressions using the logical OR and AND operators.

The **logical OR operator** (`||`) returns true if either of the Boolean expressions is true. The logical AND operator (`&&`) returns true if both of the Boolean expressions are true. Using Compound Expressions with the .IF Directive

You can use compound expressions with the .IF directive to control the flow of execution of your program.

For example, the following .IF directive uses the logical OR operator to compare the values of the eax and ebx registers:

```
1074 .IF eax > 10 || ebx > 20
1075     mov ecx, 1
1076 .ELSE
1077     mov ecx, 0
1078 .ENDIF
```

This code will move the value 1 to the ecx register if the value of eax is greater than 10 or the value of ebx is greater than 20. Otherwise, the code will move the value 0 to the ecx register.

The following .IF directive uses the logical AND operator to compare the values of the eax and ebx registers:

```
1081 .IF eax > 10 && ebx > 20  
1082     mov ecx, 1  
1083 .ELSE  
1084     mov ecx, 0  
1085 .ENDIF
```

This code will move the value 1 to the ecx register only if the value of eax is greater than 10 and the value of ebx is greater than 20. Otherwise, the code will move the value 0 to the ecx register.

Compound Boolean expressions can be used to create more complex conditional statements using the .IF directive. This can be helpful for controlling the flow of execution of your program in response to different conditions.

=====

### *SetCursorPosition Example*

=====

The SetCursorPosition procedure sets the cursor position to the specified coordinates. It receives two input parameters: DL (X-coordinate) and DH (Y-coordinate).

The procedure first checks if the X-coordinate and Y-coordinate are within the valid ranges. If either coordinate is out of range, the procedure displays an error message and exits.

The following code shows the range checking code in the SetCursorPosition procedure

```
1090 .IF (dl < 0) || (dl > 79)
1091 mov edx,OFFSET BadXCoordMsg
1092 call WriteString
1093 jmp quit
1094 .ENDIF
1095
1096 .IF (dh < 0) || (dh > 24)
1097 mov edx,OFFSET BadYCoordMsg
1098 call WriteString
1099 jmp quit
1100 .ENDIF
```

The .IF directive is used to check if the X-coordinate or Y-coordinate is out of range. The logical OR operator (||) is used to combine the two conditions.

If either condition is true, the procedure displays an error message and exits.

If the X-coordinate and Y-coordinate are within the valid ranges, the procedure calls the Gotoxy procedure to set the cursor position.

The following code shows the code that sets the cursor position:

```
call Gotoxy
```

The Gotoxy procedure is a built-in MASM procedure that sets the cursor position to the specified

coordinates.

The SetCursorPosition procedure is an example of how to use the .IF directive to range check input parameters. This can be helpful for preventing errors in your program.

=====

### ***College Registration Example***

=====

The college registration example you provided uses the .IF, .ELSEIF, and .ENDIF directives to implement a multiway branch structure. The structure checks the student's grade average and number of credits to determine whether or not the student can register.

The following is a simpler explanation of the code:

```
1109 .data
1110     TRUE  = 1
1111     FALSE = 0
1112     gradeAverage WORD 275
1113     ; test value
1114     credits WORD 12
1115     ; test value
1116     OkToRegister BYTE ?
1117 .code
1118     mov OkToRegister, FALSE
1119
1120     ; Check if the student's grade average is greater than 350.
1121     .IF gradeAverage > 350
1122         mov OkToRegister, TRUE
1123     .ELSEIF (gradeAverage > 250) && (credits <= 16)
1124         mov OkToRegister, TRUE
1125     .ELSEIF (credits <= 12)
1126         mov OkToRegister, TRUE
1127     .ENDIF
```

The .IF directive is used to check the first condition: whether the student's grade average is greater than 350. If the condition is true, the mov instruction sets the OkToRegister variable to TRUE.

The .ELSEIF directive is used to check the second condition: whether the student's grade average is

greater than 250 and the number of credits the student wants to take is less than or equal to 16. If both conditions are true, the `mov` instruction sets the `OkToRegister` variable to TRUE.

The `.ELSEIF` directive is used to check the third condition: whether the number of credits the student wants to take is less than or equal to 12. If the condition is true, the `mov` instruction sets the `OkToRegister` variable to TRUE.

If none of the conditions are true, the `OkToRegister` variable will remain set to FALSE.

The following is a breakdown of the generated code that you sent:

```
1132 mov OkToRegister, FALSE
1133 cmp word ptr gradeAverage, 350
1134 jbe @C0006
1135 mov byte ptr OkToRegister, TRUE
1136 jmp @C0008
1137 @C0006:
1138 cmp word ptr gradeAverage, 250
1139 jbe @C0009
1140 cmp word ptr credits, 16
1141 ja @C0009
1142 byte ptr OkToRegister, TRUE
1143 mov
1144 jmp @C0008
1145 @C0009:
1146 cmp word ptr credits, 12
1147 ja @C0008
1148 mov
```

- The first line of code moves the value FALSE to the OkToRegister variable.
- The next two lines of code compare the student's grade average to 350. If the grade average is greater than 350, the program jumps to the label @C0008. Otherwise, the program continues to the next line of code.
- The next three lines of code compare the student's grade average to 250 and the number of credits the student wants to take to 16. If both conditions are true, the program jumps to the label @C0008.

Otherwise, the program continues to the next line of code.

- The next two lines of code compare the number of credits the student wants to take to 12. If the number of credits is less than or equal to 12, the program jumps to the label @C0008. Otherwise, the program continues to the next line of code.
- The label @C0008 is a jump target. If the program jumps to this label, the OkToRegister variable will be set to TRUE.
- The program exits at the end of the code.

## ***Creating Loops with .REPEAT and .WHILE***

Here is a more complete explanation of the .REPEAT and .WHILE directives:

=====

### ***.Repeat directive***

=====

The .REPEAT directive creates a loop that executes the statements in the loop body once before testing the condition following the .UNTIL directive.

If the condition is false, the loop body is executed again. The loop continues to execute until the condition becomes true.

Syntax:

```
1153 .REPEAT  
1154 statements  
1155 .UNTIL condition
```

```
1159 mov eax, 0  
1160 .REPEAT  
1161     inc eax  
1162     call WriteDec  
1163     call Crlf  
1164 .UNTIL eax == 10
```

This code will display the values 1 through 10 to the console.

=====

### ***.While directive***

=====

The **.WHILE** directive creates a loop that tests the condition before executing the loop body. If the condition is false, the loop body is skipped. The loop continues to execute until the condition becomes false.

Syntax:

```
1169 .WHILE condition  
1170 statements  
1171 .ENDW
```

```
1173 mov eax, 0  
1174 .WHILE eax < 10  
1175     inc eax  
1176     call WriteDec  
1177     call Crlf  
1178 .ENDW
```

This code will also display the values 1 through 10 to the console.

### ***Differences Between .REPEAT and .WHILE***

The main difference between the .REPEAT and .WHILE directives is that the .REPEAT directive executes the loop body at least once, even if the condition is false. The .WHILE directive, on the other hand, will skip the loop body if the condition is false.

### ***Which Directive to Use?***

In general, you should use the .WHILE directive for loops where you need to test the condition before executing the loop body. This is because the .WHILE directive is more efficient than the .REPEAT directive, as it avoids executing the loop body if the condition is false.

However, there are some cases where you may want to use the .REPEAT directive. For example, you may want to use the .REPEAT directive if you need to initialize a variable before executing the loop body.

### **Conclusion**

The .REPEAT and .WHILE directives are two powerful tools for creating loops in MASM. By understanding the differences between the two directives, you can choose the right directive for your needs.

---

The notes you provided are unclear and incomplete because they do not provide a complete explanation of how to implement the pseudocode using the .WHILE and .IF directives.

Here is a more complete explanation:

```
1184 .data
1185     X DWORD 0
1186     op1 DWORD 2
1187     ; test data
1188     op2 DWORD 4
1189     ; test data
1190     op3 DWORD 5
1191     ; test data
1192 .code
1193     mov eax, op1
1194     mov ebx, op2
1195     mov ecx, op3
1196
1197     .WHILE eax < ebx
1198         inc eax
1199
1200         .IF eax == ecx
1201             mov X, 2
1202         .ELSE
1203             mov X, 3
1204         .ENDIF
1205     .ENDW
```

This code will loop from the value of op1 to the value of op2, incrementing op1 on each iteration.

Within the loop, the code uses the .IF directive to check if op1 is equal to op3.

If it is, the code moves the value 2 to X. Otherwise, the code moves the value 3 to X.

The .WHILE directive will continue to loop until op1 is greater than or equal to op2.

### ***Here is a breakdown of the code:***

The code you provided is a loop that executes the following steps:

1. Moves the values of the variables op1, op2, and op3 to the registers eax, ebx, and ecx, respectively.
  2. Starts a .WHILE loop that will continue to execute until eax is greater than or equal to ebx.
  3. Increments the eax register by 1.
  4. Uses the .IF directive to check if eax is equal to ecx.
- If eax is equal to ecx, the code moves the value 2 to the variable X.
  - Otherwise, the code moves the value 3 to the variable X.
  - Ends the .WHILE loop.

This loop will essentially iterate from the value of op1 to the value of op2, incrementing op1 on each iteration. Within the loop, the code checks if op1 is equal to op3. If it is, the code moves the value 2 to X. Otherwise, the code moves the value 3 to X.

Here is a simpler explanation:

- Input: Three variables: op1, op2, and op3.
- Output: The variable X.
- Algorithm:
  1. Initialize X to 0.

Iterate from op1 to op2, incrementing op1 on each iteration.▪ If op1 is equal to op3, set X to 2. Otherwise, set X to 3.

=====

## Questions

=====

Convert an ASCII digit in AL to its corresponding binary value:

```
1211 cmp al, '0'    ; Compare AL with ASCII '0'  
1212 jb done        ; If AL is less than '0', it's not a valid digit  
1213 cmp al, '9'    ; Compare AL with ASCII '9'  
1214 ja done        ; If AL is greater than '9', it's not a valid digit  
1215 sub al, '0'    ; Convert ASCII digit to binary by subtracting '0'  
1216 done:
```

Calculate the parity of a 32-bit memory operand:

```
1221 xor eax, eax ; Clear EAX (parity result)
1222 xor ebx, ebx ; Clear EBX (loop counter)
1223 loop_start:
1224 xor al, [edi + ebx] ; XOR AL with the next byte in memory
1225 inc ebx ; Increment loop counter
1226 cmp ebx, 32 ; Compare loop counter with 32
1227 jl loop_start ; If not all 32 bits processed, continue
```

Generate a bit string in EAX representing members in SetX not in SetY:

```
1232 ; Assuming SetX and SetY are two memory operands of the same size (e.g., 32 bits)
1233 mov eax, SetX ; Load SetX into EAX
1234 and eax, not SetY ; Apply NOT operation to SetY and AND with SetX
```

Jump to label L1 when DX <= CX:

Jump to label L2 when AX > CX (signed comparison):

Clear bits 0 and 1 in AL and jump based on the destination operand:

```
1238 cmp dx, cx ; Compare DX and CX
1239 jbe L1       ; Jump to L1 if DX <= CX
1240
1241 cmp ax, cx ; Compare AX and CX (signed comparison)
1242 jg L2       ; Jump to L2 if AX > CX
1243
1244 and al, 0xFC ; Clear bits 0 and 1 in AL
1245 test al, al ; Test if AL is zero
1246 jz L3       ; Jump to L3 if AL is zero
1247 jmp L4       ; Jump to L4 (if AL is not zero)
```

Let's start with implementing the pseudocode for the first exercise using short-circuit evaluation in assembly language. The pseudocode is as follows:

```
1251 if( val1 > ecx ) AND ( ecx > edx )
1252     X = 1
1253 else
1254     X = 2;
```

Here's the corresponding assembly code:

```
1258 ; Assuming val1, ecx, edx, and X are 32-bit variables
1259 ; Also, assuming the condition is checked within a function
1260
1261 cmp dword [val1], ecx          ; Compare val1 with ecx
1262 jle else_condition           ; Jump to else_condition if val1 <= ecx
1263
1264 cmp ecx, edx                ; Compare ecx with edx
1265 jle else_condition           ; Jump to else_condition if ecx <= edx
1266
1267 mov dword [X], 1             ; Set X to 1 if both conditions are met
1268 jmp done                     ; Jump to done to skip the else block
1269
1270 else_condition:
1271 mov dword [X], 2             ; Set X to 2 if conditions are not met
1272
1273 done:
1274 ; Rest of the code continues here
```

In this code, we first compare val1 with ecx. If val1 is less than or equal to ecx, we jump to the else\_condition label, effectively skipping the  $X = 1$  assignment.

Then, we compare ecx with edx. If ecx is less than or equal to edx, we also jump to the else\_condition label.

If both conditions are met (`val1 > ecx` and `ecx > edx`), we set X to 1. Otherwise, if either condition is not met, we set X to 2.

The `jmp done` statement ensures that we skip the `else_condition` block when both conditions are met.

-----

### ***Exercise 8:***

Implement the following pseudocode using short-circuit evaluation:

```
1280 if( ebx > ecx ) OR ( ebx > val1 )
1281     X = 1
1282 else
1283     X = 2
```

Here's the corresponding assembly code:

```
1286 ; Assuming ebx, ecx, val1, and X are 32-bit variables
1287 ; Also, assuming the condition is checked within a function
1288
1289 cmp ebx, ecx          ; Compare ebx with ecx
1290 jg set_X_to_1         ; Jump to set_X_to_1 if ebx > ecx
1291
1292 cmp ebx, val1         ; Compare ebx with val1
1293 jg set_X_to_1         ; Jump to set_X_to_1 if ebx > val1
1294
1295 ; If neither condition is met, set X to 2 and continue
1296 mov dword [X], 2
1297 jmp done
1298
1299 set_X_to_1:
1300 mov dword [X], 1      ; Set X to 1 if either condition is met
1301
1302 done:
1303 ; Rest of the code continues here
```

In this code, we first compare ebx with ecx. If ebx is greater than ecx, we jump to the `set_X_to_1` label, effectively setting X to 1.

Next, we compare ebx with val1. If ebx is greater than val1, we again jump to `set_X_to_1`, ensuring that X is set to 1 if either condition is met.

If neither condition is met, we set X to 2 and continue with the code.

---

### **Exercise 9:**

Implement the following pseudocode using short-circuit evaluation:

```
1307 if( ebx > ecx AND ebx > edx) OR ( edx > eax )
1308     X = 1
1309 else
1310     X = 2
```

Here's the corresponding assembly code:

```
1314 ; Assuming ebx, ecx, edx, eax, and X are 32-bit variables
1315 ; Also, assuming the condition is checked within a function
1316
1317 ; Check the first condition: ebx > ecx AND ebx > edx
1318 cmp ebx, ecx          ; Compare ebx with ecx
1319 jle check_second_condition ; If ebx <= ecx, skip the second condition
1320
1321 cmp ebx, edx          ; Compare ebx with edx
1322 jle check_second_condition ; If ebx <= edx, skip the second condition
1323
1324 ; If we reach here, both conditions are met, so set X to 1
1325 mov dword [X], 1
1326 jmp done
1327
1328 check_second_condition:
1329 ; Check the second condition: edx > eax
1330 cmp edx, eax          ; Compare edx with eax
1331 jle set_X_to_2          ; If edx <= eax, set X to 2 and skip to done
1332
1333 ; If we reach here, the second condition is met, so set X to 1
1334 mov dword [X], 1
1335 jmp done
1336
1337 set_X_to_2:
1338 ; If neither condition is met, set X to 2
1339 mov dword [X], 2
1340
1341 done:
1342 ; Rest of the code continues here
```

In this code, we first check the first condition: ebx > ecx AND ebx > edx. If either of these subconditions is not met, we skip to check\_second\_condition.

If both subconditions are met, we set X to 1 and jump to done.

In check\_second\_condition, we check the second condition: edx > eax. If this condition is met, we set X to 1 and jump to done. If the second condition is not met, we set X to 2.

This code implements the pseudocode with short-circuit evaluation as requested.

Now, we have one more exercise remaining:

---

### **Exercise 10:**

Implement the following pseudocode using short-circuit evaluation:

```
1347 while N > 0
1348     if N != 3 AND (N < A OR N > B)
1349         N = N - 2
1350     else
1351         N = N - 1
```

Here's the corresponding assembly code:

---

```
1356 ; Assuming N, A, B are 32-bit signed integers
1357 ; Also, assuming this code is part of a larger program
1358 while_loop:
1359     cmp dword [N], 0          ; Compare N with 0
1360     jle end_while            ; If N <= 0, exit the loop
1361     cmp dword [N], 3          ; Compare N with 3
1362     je skip_decrement       ; If N == 3, skip to else (decrement by 1)
1363     ; Check the second condition: N < A OR N > B
1364     cmp dword [N], [A]        ; Compare N with A
1365     jge else_decrement       ; If N >= A, skip to else (decrement by 1)
1366     cmp dword [N], [B]        ; Compare N with B
1367     jle else_decrement       ; If N <= B, skip to else (decrement by 1)
1368     ; If neither condition is met, decrement by 2
1369     sub dword [N], 2
1370     jmp continue_while      ; Continue the loop
1371 skip_decrement:
1372     ; N == 3, decrement by 1
1373     sub dword [N], 1
1374     jmp continue_while
1375 else_decrement:
1376     ; If any condition is met, decrement by 1
1377     sub dword [N], 1
1378 continue_while:
1379     ; Loop back to the beginning of the while loop
1380     jmp while_loop
1381 end_while:
1382 ; Rest of the code continues here
```

In this code, we use a while loop to repeatedly check the conditions and decrement N accordingly. The loop continues as long as N is greater than 0.

If N is equal to 3, we decrement it by 1 (skip to the else part). If N is not equal to 3 and either N is greater than A or N is less than B, we decrement N by 2.

Otherwise, we decrement N by 1. The loop continues until N becomes less than or equal to 0.

-----

In the provided assembly code, `continue_while`, `end_while`, and `else_decrement` are not procedures or labels that you explicitly call; they are labels used for control flow within the while loop. Let me clarify their roles:

**continue\_while:** This is a label used to mark the point in the code where the loop should continue if none of the conditions for decrementing N are met. It is not a procedure or function that you explicitly call; instead, it's a reference point for the code to jump back to the beginning of the while loop.

**end\_while:** This is also a label used to mark the end of the while loop. When the condition  $N \leq 0$  is met, the code jumps to this label to exit the loop and continue with the rest of the program.

**else\_decrement:** This label is used to mark the point in the code where N is decremented by 1 when the condition  $N \neq 3$  AND ( $N < A$  OR  $N > B$ ) is not met. It's used to skip the decrement by 2 and directly decrement by 1.

These labels are simply reference points in the code to control the flow of execution within the while loop. You don't explicitly call them as procedures or functions. Instead, the program flow

jumps to these labels based on conditional branches (jmp) to implement the logic of the while loop.

I hope this helps!

## *Questions for this topic*

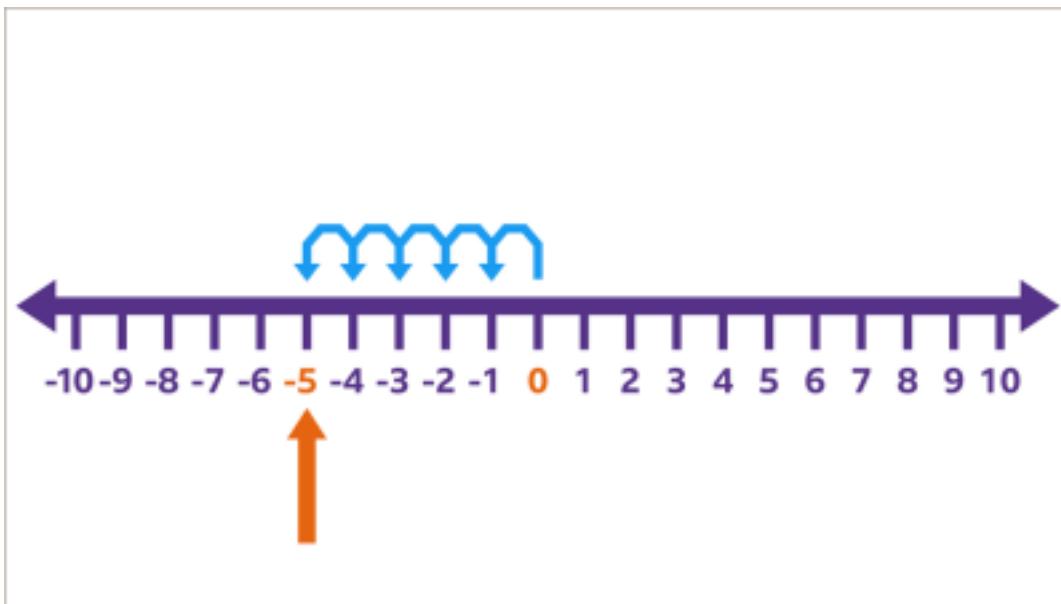
I can explain the suggestions provided for testing your code:

**Using a Debugger:** Debugging your code using a debugger is a crucial step in ensuring its correctness. Debuggers allow you to step through your code line by line, inspect variables, and identify issues. You can use debugging tools provided by your development environment (e.g., Visual Studio Debugger).

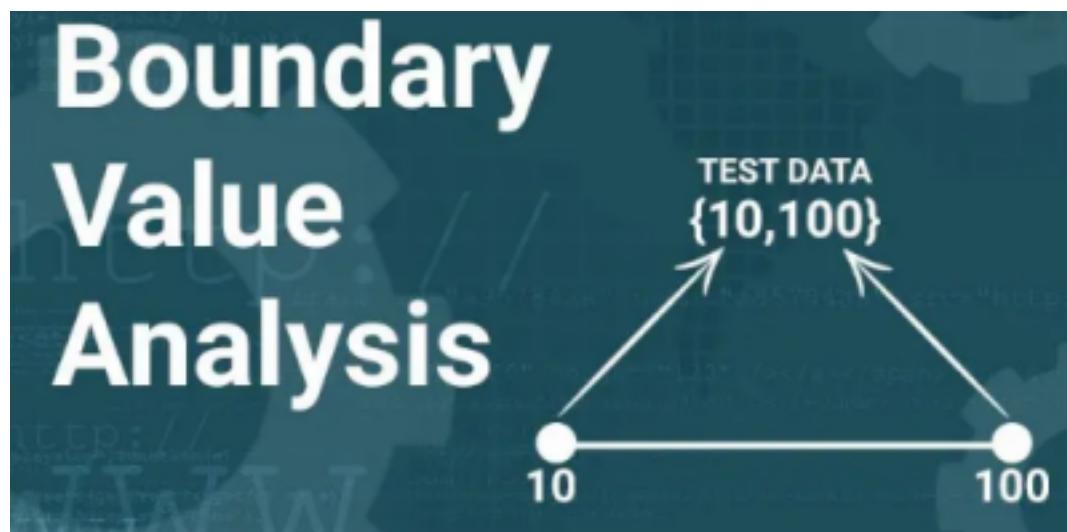


**Testing with Negative Values:** If your code deals with signed data, it's essential to include

negative values in your test cases to cover all possible scenarios.



**Testing at Boundaries:** When a range of input values is specified, test your code with values that fall before, on, and after these boundaries. This helps verify how your code handles edge cases.

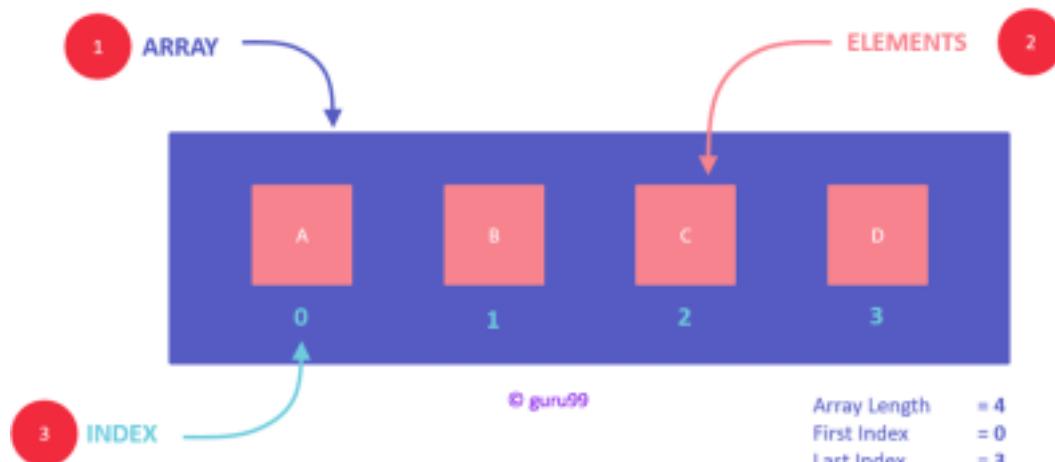


**Multiple Test Cases:** Create multiple test cases with different inputs and conditions. This ensures that your code is robust and can handle a variety of scenarios.



**Using a Debugger for Array Operations:** When working with arrays, especially when modifying them, a debugger's Memory window can be very useful. It allows you to inspect the array's contents in hexadecimal or decimal representation.

## CONCEPT DIAGRAM



**Checking Register Preservation:** If you have a procedure that modifies registers, consider calling it twice in a row. This helps verify that the procedure correctly preserves register values between calls.

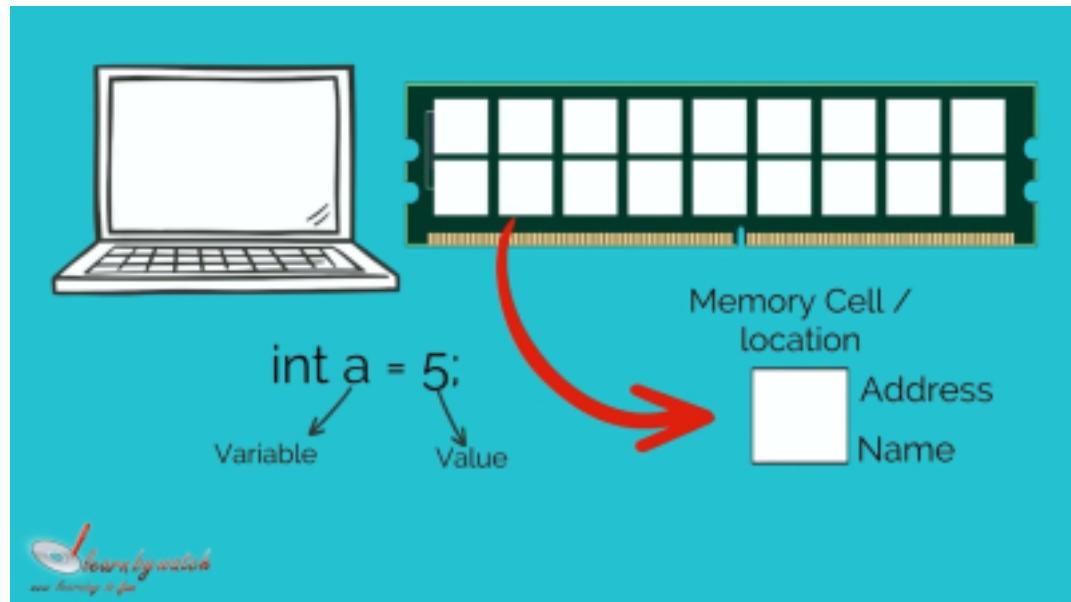


**Parameter Passing for Multiple Arrays:** When passing multiple arrays to a procedure, it's a good practice not to refer to arrays by name inside the procedure. Instead, set registers like ESI or EDI to the offsets of the arrays before calling the procedure. Use indirect addressing ([esi] or [edi]) inside the procedure to access array elements.

## Passing Parameters

The screenshot shows a debugger interface with two main panes. The left pane displays assembly code with labels like 'MAIN PROC', 'RESTART', and several 'MOV' instructions. The right pane shows the state of registers: ECX contains 100, EDI contains 300, and ESI contains 200. The stack dump on the far right shows memory starting at address 200 containing the values 100, 300, and 200, which correspond to the register values. This illustrates how arrays are passed by offset rather than by name.

**Local Variables in Procedures:** If you need to create variables for use only within a procedure, you can declare them using the .data directive before the variable and the .code directive afterward. Initialize these variables within the procedure to ensure they start with the correct values when the procedure is called multiple times.



```
1386 .data
1387 array DWORD 10 DUP (?) ; Define an array to hold the random integers
1388 .code
1389 FillArray PROC
1390     ; Parameters:
1391     ; edi = pointer to the array
1392     ; ecx = N (number of elements)
1393     ; ebx = j (lower bound)
1394     ; edx = k (upper bound) || you can initialize random number generator (optional)
1395     call InitializeRandom
1396     ; Loop to fill the array with random numbers
1397     fill_loop:
1398         mov eax, ebx          ; Load lower bound (j) into eax
1399         sub eax, 1           ; Subtract 1 to make j inclusive
1400         add eax, edx          ; Calculate the range (k - j + 1)
1401         call GetRandom        ; Get a random number in [0, range)
1402         add eax, ebx          ; Add j to the random number to fit [j, k]
1403         mov [edi], eax        ; Store the random number in the array
1404         add edi, 4            ; Move to the next element
1405         loop fill_loop       ; Repeat for N elements
1406     ret
1407 FillArray ENDP
1408 main:
1409     ; Usage example:
1410     mov edi, OFFSET array ; Pointer to the array
1411     mov ecx, 10             ; N = 10 elements
1412     mov ebx, 1              ; Lower bound (j)
1413     mov edx, 100            ; Upper bound (k)
1414     call FillArray
1415     ; Call FillArray again with different j and k values if needed, Verify the results using a debugger
1416     ;(you can inspect the contents of the 'array' variable), and the rest of the program
```

This code defines a procedure called `FillArray`, which fills an array with random integers within the specified range. The main program demonstrates how to use this procedure with different values of `j` and `k`.

```
=====
```

### *Exercise 1: Summing an Array*

```
=====
```

This exercise requires you to create a procedure that returns the sum of all array elements within the range  $[j, k]$ . You'll pass a pointer to the array, the size of the array, and the values of `j` and `k` as parameters to the procedure. The sum should be returned in the `EAX` register, and all other register values should be preserved between calls.

Here's a sample assembly code for this exercise:

```
1421 .data
1422     array SDWORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ; Example array of signed doublewords
1423 .code
1424 SumInRange PROC
1425     ; Parameters:
1426     ;    edi = pointer to the array
1427     ;    ecx = size of the array
1428     ;    ebx = j (lower bound)
1429     ;    edx = k (upper bound)
1430     xor eax, eax           ; Clear EAX to store the sum
1431     sum_loop:
1432         mov esi, [edi]      ; Load the next element into ESI
1433         cmp esi, ebx        ; Compare with lower bound (j)
1434         jl not_in_range    ; Jump if less than j
1435         cmp esi, edx        ; Compare with upper bound (k)
1436         jg not_in_range    ; Jump if greater than k
1437         add eax, esi        ; Add to the sum
1438     not_in_range:
1439         add edi, 4          ; Move to the next element
1440         loop sum_loop      ; Repeat for all elements
1441     ret
1442 SumInRange ENDP
1443 main:                      ; Usage example:
1444     mov edi, OFFSET array  ; Pointer to the array
1445     mov ecx, 10            ; Size of the array
1446     mov ebx, 2              ; Lower bound (j)
1447     mov edx, 7              ; Upper bound (k)
1448     call SumInRange
1449     ; The sum will be in the EAX register
1450     ; Call SumInRange again with different j and k values if needed
1451     ; Rest of the program
```

This code defines a procedure called SumInRange, which calculates the sum of array elements within the specified range [j, k]. The main program demonstrates how to use this procedure with different values of j and k.

```
=====
```

### ***Exercise 1: TestScore Evaluation***

```
=====
```

This exercise requires you to create a procedure named CalcGrade that receives an integer value between 0 and 100 and returns a single capital letter grade in the AL register. The grade returned should be based on specified ranges.

Here's a sample assembly code for this exercise:

```
1455 .data
1456     grade CHAR ? ; Variable to store the grade
1457 .code
1458 CalcGrade PROC
1459     ; Parameter:
1460     ;    eax = integer value between 0 and 100
1461     cmp eax, 0
1462     jl invalid_input      ; Input is less than 0, return 'F'
1463     cmp eax, 60
1464     jl grade_F            ; Input is less than 60, return 'F'
1465     cmp eax, 70
1466     jl grade_D            ; Input is less than 70, return 'D'
1467     cmp eax, 80
1468     jl grade_C            ; Input is less than 80, return 'C'
1469     cmp eax, 90
1470     jl grade_B            ; Input is less than 90, return 'B'
1471     grade_A:
1472         mov al, 'A'        ; Input is 90 or greater, return 'A'
1473         jmp done
1474     grade_B:
1475         mov al, 'B'        ; Input is between 80 and 89, return 'B'
1476         jmp done
1477     grade_C:
1478         mov al, 'C'        ; Input is between 70 and 79, return 'C'
1479         jmp done
1480     grade_D:
1481         mov al, 'D'        ; Input is between 60 and 69, return 'D'
1482         jmp done
1483     grade_F:
1484         mov al, 'F'        ; Input is between 0 and 59, return 'F'
```

```
1485     invalid_input:  
1486         mov al, '?'           ; Invalid input, return '?'  
1487  
1488     done:  
1489         ret  
1490 CalcGrade ENDP  
1491  
1492 main:  
1493     ; Usage example:  
1494     mov eax, 85            ; Input value (test score)  
1495     call CalcGrade  
1496  
1497     ; The grade will be in the AL register  
1498  
1499     ; Rest of the program
```

This code defines a procedure called CalcGrade, which returns a grade based on the specified ranges. The main program demonstrates how to use this procedure by passing a test score (integer value) and receiving the corresponding grade in the AL register.

=====

Now it's time for you to do your own practice:

## ***Exercise 4: Test Score Evaluation***

Create a program that generates 10 random integers between 50 and 100 (inclusive). For each integer generated, pass it to the CalcGrade procedure, which will return a corresponding letter grade based on specified ranges. Display the integer and its corresponding letter grade. You can use the RandomRange procedure from the Irvine32 library to generate random integers.

## ***Exercise 5: Boolean Calculator (1)***

Create a program that acts as a simple boolean calculator for 32-bit integers. It displays a menu with options to perform logical operations (AND, OR, NOT, XOR) and allows the user to choose an operation. Implement this menu using Table-Driven Selection. When the user selects an operation, call a procedure to display the operation name. Implement this menu-driven program.

## ***Exercise 6: Boolean Calculator (2)***

Continuing from Exercise 5, implement procedures for each of the logical operations (AND, OR, NOT, XOR). Prompt the user for inputs (hexadecimal integers) as required by the chosen operation, perform the operation, and display the result in hexadecimal.

## ***Exercise 7: Probabilities and Colors***

Write a program that randomly selects one of three colors (white, blue, green) with specific probabilities (30%, 10%, 60%). Use a loop to display 20 lines of text, each with a randomly chosen color based on the given probabilities. You can generate a random integer between 0 and 9 and use it to select colors accordingly.

## ***Exercise 8: Message Encryption***

Revise an encryption program to encrypt and decrypt a message using an encryption key consisting of

multiple characters. Implement encryption and decryption by XOR-ing each character of the key against a corresponding byte in the message. Repeat the key as necessary until all plaintext bytes are translated.

Plain text	T	h	i	s		i	s		a	a	P	l	a	i	n	t	e	x	t		m	e	s	s	a	g	e	(etc.)
Key	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	8	X	m	v	#	7

(The key repeats until it equals the length of the plain text...)

### ***Exercise 9: Validating a PIN***

Create a procedure called Validate\_PIN that checks the validity of a 5-digit PIN based on specified digit ranges. The procedure receives a pointer to an array containing the PIN and validates each digit. If any digit is outside its valid range, return the digit's position (1 to 5) in the EAX register; otherwise, return 0. Write a test program that calls Validate\_PIN with valid and invalid PINs and verifies the return values.

Digit Number	Range
1	5 to 9
2	2 to 5
3	4 to 8
4	1 to 4
5	3 to 6

### ***Exercise 10: Parity Checking***

Implement a procedure that checks the parity (even or odd) of bytes in an array. The procedure returns True (1) if the bytes have even parity and False (0) if they have odd parity. Write a test program that calls the procedure with arrays having even and odd parity and verifies the return values.

## **INTEGER ARITHMETIC**

Skip....

# **Shifts and Rotates**

In assembly language, you have direct control over the bits and bytes of data, which allows for highly efficient and platform-specific optimizations. Let's dive a bit deeper into these concepts:

Bit shifting involves moving the bits of a binary number left or right.

**Left Shift (<<)**: Shifting bits to the left by a certain number of positions effectively multiplies the value by 2 for each shift.

**Right Shift (>>)**: Shifting bits to the right by a certain number of positions effectively divides the value by 2 for each shift (for non-negative numbers).

**Logical Shift**: Fills the shifted-in bits with zeros. **Arithmetic Shift**: Preserves the sign bit when shifting right (for signed numbers).

**Bit Rotation**: Bit rotation involves moving bits in a circular manner, so they wrap around. **Left Rotation**: Bits are shifted left, and the bits that go beyond the most significant bit (MSB) are wrapped around to the least significant bit (LSB).

**Right Rotation**: Bits are shifted right, and the bits that go beyond the LSB are wrapped around to the MSB.

## **Uses:**

**Optimized Multiplication and Division**: Bit shifting can be used to perform multiplication and division more efficiently.

For example, shifting left by n positions is equivalent to multiplying by  $2^n$ , and shifting right is equivalent to division by  $2^n$ .

#### ***Data Encryption:***

Bit manipulation plays a crucial role in encryption algorithms like the XOR cipher.

***Computer Graphics:*** Manipulating pixels and colors often involves bit-level operations for tasks like image processing and rendering.

***Hardware Manipulation:*** In embedded systems, controlling hardware often requires setting or clearing specific bits to interact with peripherals. These concepts are indeed powerful and can lead to highly optimized code.

---

Assembly language provides the flexibility to perform arithmetic operations on arbitrary-length integers, a feature not always readily available in high-level languages.

This capability can be particularly useful when dealing with large numbers or implementing custom arithmetic operations.

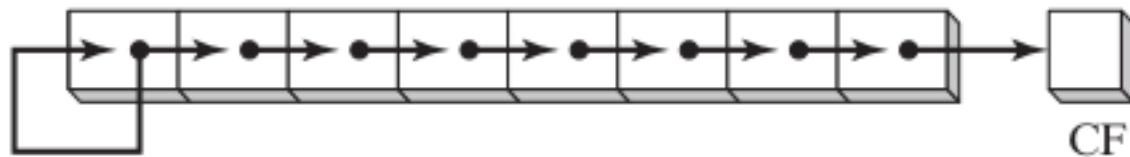
Let's discuss some key points related to shift and rotate instructions in assembly language, especially focusing on x86 processors:

---

#### ***SAL/SAR (Arithmetic Shift Left/Arithmetic Shift Right):***

Similar to SHL and SHR, but SAR preserves the sign bit when shifting right, making it suitable for signed integers.

The newly created bit position is filled with a copy of the original number's sign bit:



Binary 11001111, for example, has a 1 in the sign bit. When shifted arithmetically 1 bit to the right, it becomes 11100111:

1	1	0	0	1	1	1	1	→ (cf)
↓	↓	↓	↓	↓	↓	↓	↓	
1	1	1	0	0	1	1	1	

---

### ***ROL/ROR (Rotate Left/Rotate Right):***

Rotation instructions are used to shift bits in a circular manner, wrapping around from one end to the other. ROL rotates bits left, and ROR rotates bits right.

---

***RCL/RCR (Rotate through Carry Left/Rotate through Carry Right):***

These instructions are similar to ROL and ROR but incorporate the Carry flag, making them useful for multi-precision arithmetic and shifts.

---

# *Shift Left and Shift Right*

---

## *Shift and Rotate Instructions:*

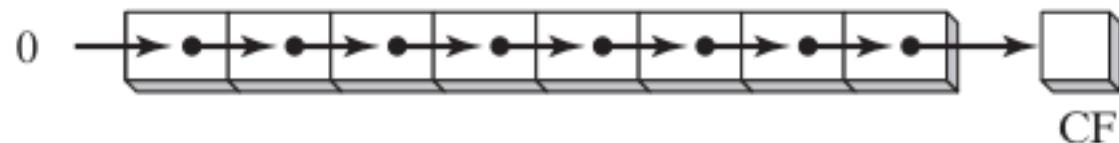
---

Bit shifting involves moving bits within an operand either to the left or right. The x86 processor architecture offers a wide range of shift and rotate instructions, each with specific purposes and effects on flags like Overflow and Carry. Here are a few common shift and rotate instructions on x86:

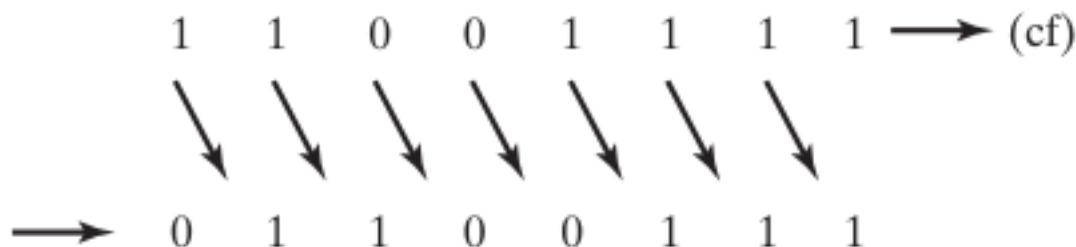
---

### *SHL (Shift Left):*

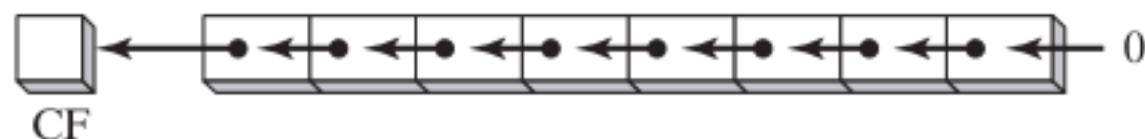
These instructions shift bits left (SHL) or right (SHR) within an operand. SHL multiplies the value by 2 for each shift left, while SHR effectively divides the value by 2 for each shift right.



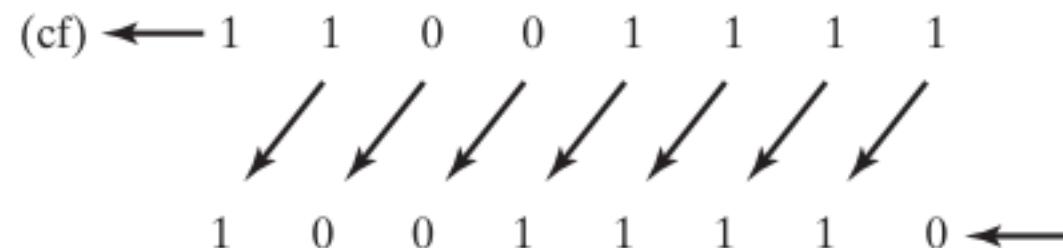
The following illustration shows a single logical right shift on the binary value 11001111, producing 01100111. The lowest bit is shifted into the Carry flag:



The **SHL (shift left) instruction** performs a logical left shift on the destination operand, filling the lowest bit with 0. The highest bit is moved to the Carry flag, and the bit that was in the Carry flag is discarded:



If you shift 11001111 left by 1 bit, it becomes 10011110:



**SHL** destination, count

**SHL reg,imm8**

**SHL mem,imm8**

**SHL reg,CL**

**SHL mem,CL**

The SHL instruction can **only be used to shift integers**, not floating-point numbers.

The imm8 operand must be between 0 and 7, inclusive. For example, the following instruction is invalid:

**shl eax, 8**

If the imm8 operand is greater than 7, the shift count will be wrapped around to the range 0-7. For example, the following instruction is equivalent to the instruction `shl eax, 1`:

**shl eax, 9**

If the CL register is used as the shift count operand, it must contain a value between 0 and 31, inclusive. For example, the following instruction is invalid:

```
shl eax, cl ; cl = 32
```

If the CL register contains a value greater than 31, the shift count will be wrapped around to the range 0-31. For example, the following instruction is equivalent to the instruction `shl eax, 1`:

```
shl eax, cl ; cl = 33
```

*Here is a more clear explanation of the SHL instruction:*

- The SHL instruction shifts the bits of the destination operand to the left by the specified number of bits.
- The highest bit of the destination operand is shifted out and copied into the Carry flag.
- The lowest bit position of the destination operand is assigned zero.

The following table shows the possible operands for the SHL instruction:

Name	Description
reg	A general-purpose register.
mem	A memory location.
imm8	An immediate value between 0 and 7.
CL	The CL register.

*The notes you provided are not clear because they do not explicitly state the following:*

- The **SHL instruction** can only be used to shift integers, **not** floating-point numbers.
- The **imm8** operand must be **between 0 and 7, inclusive**. For example, the following instruction is invalid:
- **shl eax, 8** If the imm8 operand is greater than 7, the shift count will be wrapped around to the range 0-7.

*For example, the following instruction is equivalent to the instruction `shl eax, 1`:*

```
07 mov bl, 8Fh ; BL = 10001111b
08 shl bl, 1
09 ; CF = 1, BL = 00011110b
```

After the SHL instruction is executed, the Carry flag will be set to 1 and the BL register will contain the value 00011110b.

When the SHL instruction is used to shift a value to the left multiple times, the Carry flag will contain the last bit to be shifted out of the most significant bit (MSB).

*For example, the following code shows how to shift the value of the AL register to the left by two bits:*

```
13 mov al, 10000000b  
14 shl al, 2  
15 ; CF = 0, AL = 00000000b
```

After the first SHL instruction is executed, the Carry flag will be set to 1 and the AL register will contain the value 01000000b.

After the second SHL instruction is executed, the Carry flag will be set to 0 and the AL register will contain the value 00000000b.

The SHL instruction can be used to perform a variety of operations, such as multiplying a value by two, converting a binary number to a decimal number, and packing and unpacking data.

---

When a value is shifted rightward multiple times, the **Carry flag contains the last bit to be shifted out of the least significant bit (LSB)**.

The image you sent is correct. It shows that shifting the binary number 00001010 (decimal 10) to the left by two bits is the same as multiplying it by  $2^2$ .

This is because shifting a binary number to the left by one bit is the same as multiplying it by 2. Shifting to the left by two bits is the same as multiplying by  $2^2$ , and so on.

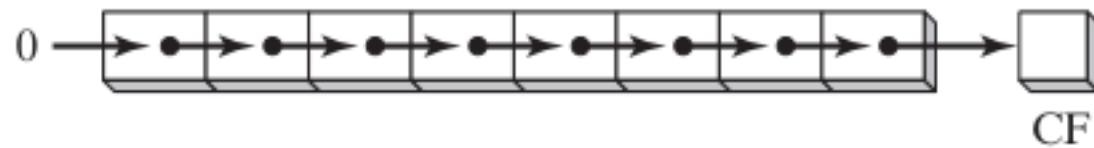
mov dl,10	;	before:	00001010
shl dl,2	;	after:	00101000

After the SHL instruction is executed, the BL register will contain the value 20.

Bitwise multiplication is often used in graphics and signal processing applications. For example, it can be used to scale images, rotate images, and apply filters to images.

## ***Shift Right Instruction***

The SHR instruction performs a logical right shift on the destination operand, replacing the highest bit with a 0.



The lowest bit is copied into the Carry flag, and the bit that was previously in the Carry flag is lost.

```
mov al,0D0h          ; AL = 11010000b
shr al,1           ; AL = 01101000b, CF = 0
```

In a multiple shift operation, the last bit to be shifted out of position 0 (the LSB) ends up in the Carry flag:

```
mov al,00000010b  
shr al,2 ; AL = 00000000b, CF = 1
```

---

**Bitwise division** is a way of dividing two numbers by shifting the bits of the dividend (the number being divided) to the right by the number of bits in the divisor.

The result of this operation is the quotient of the two numbers, rounded down to the nearest integer.

For example, to divide 32 by  $2^1$ , we would shift the bits of 32 to the right by 1 bit. This would result in the value 16, which is the quotient of 32 divided by  $2^1$ .

Here is a step-by-step explanation of bitwise division:

1. Write the dividend and divisor in binary form.
2. Shift the bits of the dividend to the right by the number of bits in the divisor.
3. The result of the shift is the quotient of the two numbers, rounded down to the nearest integer.

For example, to divide 32 by  $2^1$ , we would do the following:

```
mov dl,32      Before: 

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|

 = 32
```

```
shr dl,1      After: 

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|

 = 16
```

In the following example, 64 is divided by  $2^3$ :

```
mov al,01000000b          ; AL = 64  
shr al,3                  ; divide by 8, AL = 00001000b
```

1. Write the dividend and divisor in binary form:

Dividend: 100000

Divisor: 00100

2. Shift the bits of the dividend to the right by 1 bit:

100000 >> 1 = 010000

3. The result of the shift is the quotient of the two numbers, rounded down to the nearest integer:

Quotient = 010000 = 16

Bitwise division can be used to perform a variety of operations, such as:

- Dividing a number by a power of two.
- Converting a hexadecimal number to a decimal number.
- Calculating the remainder of a division operation.

Division of signed numbers by shifting is accomplished using the SAR instruction because it preserves the number's sign bit.

---

For example, if the AX register contains the value 0x1234, then the following instruction will shift the value to the right by one bit:

**SHR AX, 1**

After the instruction is executed, the AX register will contain the value 0x091A.

The highest bit of the original value (0x1) has been shifted out and lost, and the lowest bit of the original value (0x4) has been copied into the Carry flag.

The SHR instruction can be used to perform a variety of operations, including:

In this statement, what's being explained is how the SHR (Shift Right) instruction works using an example with the AX register containing the value 0x1234 (a 16-bit hexadecimal number).

When you apply the SHR instruction to this value, you are shifting the bits of the number to the right by one bit position. Here's a step-by-step breakdown:

right by one bit position. Here's a step-by-step breakdown:

1. Original Value: 0x1234 (binary: 0001001000110100)
  2. Right Shift by One Bit: Shifting the bits to the right by one position results in "0000100100011010."
  3. The highest bit in the original value (0x1) is the leftmost bit, and it gets shifted out and is "lost" because there's no longer space for it in the 16-bit AX register.
  4. The lowest bit in the original value (0x4) is the rightmost bit, and it has been copied into the Carry flag. This means that after the shift, the Carry flag will hold the value 1 (indicating that the bit that was shifted out was a 1).
  5. The new value in the AX register after the shift is 0x091A (binary: 0000100100011010).
- 

### ***Dividing a value by two***

Converting a decimal number to a binary number Unpacking data.

The following table shows some examples of how to use the SHR instruction:

Example	Description
SHR AX, 1	Shifts the value of the AX register to the right by one bit.
SHR BX, 2	Shifts the value of the BX register to the right by two bits.
SHR CX, 4	Shifts the value of the CX register to the right by four bits.
SHR DX, 8	Shifts the value of the DX register to the right by eight bits.

The SHR instruction can also be used to test the value of the Carry flag.

For example, the following instruction will test the value of the Carry flag and set the Zero flag if the Carry flag is clear:

```
SHR AX, 1  
JZ TEST
```

If the Carry flag is clear, then the JZ instruction will jump to the TEST label. Otherwise, the JZ instruction will be skipped.

=====

*Summary b4 we continue*

=====

Certainly, you've provided a detailed explanation of shift and rotate instructions, specifically focusing on SHL (Shift Left) and SHR (Shift Right) instructions in the context of x86 assembly language programming. Here's a summary of the information you've provided:

### ***Shift Left (SHL) Instruction:***

- SHL shifts bits to the left within an operand.
- It multiplies the value by 2 for each left shift.
- The highest bit is moved to the Carry flag, and the lowest bit is set to 0.
- It can only be used for integers, not floating-point numbers.
- The imm8 operand must be between 0 and 7, inclusive.
- If imm8 is greater than 7, the shift count wraps around to 0-7.
- If the CL register is used, it must contain a value between 0 and 31.
- If CL exceeds 31, the shift count wraps around.
- It is often used for operations like multiplying a value by 2 and bitwise multiplication.

### ***Shift Right (SHR) Instruction:***

- SHR performs a logical right shift on the destination operand.
- It replaces the highest bit with 0 and copies the lowest bit to the Carry flag.
- In multiple shifts, the last bit shifted out ends up in the Carry flag.
- It can be used for operations like dividing a value by 2, converting decimal to binary, and unpacking data.
- SAR (Shift Arithmetic Right) is used for signed numbers to preserve the sign bit.

## **Bitwise Division:**

- Bitwise division involves shifting bits of the dividend to the right by the number of bits in the divisor.
- The result is the quotient, rounded down to the nearest integer.
- It's used for operations like dividing by a power of two and converting hexadecimal to decimal.
- SAR is used for signed numbers to maintain the sign bit.

Since bitwise division (right shifting) and bitwise multiplication (left shifting) are essentially inverse operations of each other, you can focus on one of them in your notes and simply mention their relationship. Here's a more concise way to express this:

- **Bitwise Shifting:** This operation involves moving the bits of a binary number.
- **Left Shifting:** Equivalent to multiplying by 2 to the power of the shift count.
- **Right Shifting:** Equivalent to dividing by 2 to the power of the shift count

Bitwise operations, including left and right shifts, are primarily related to binary data manipulation. They work with binary representations, and you can perform these operations on hexadecimal numbers, but the shifts operate at the binary level.

In contrast, when you perform bitwise operations on other number formats like decimal or hexadecimal, the operations affect their binary representations. Bitwise operations are versatile but fundamentally operate on binary data, while left and right shifts are more broadly applicable, working with various data types beyond binary numbers.

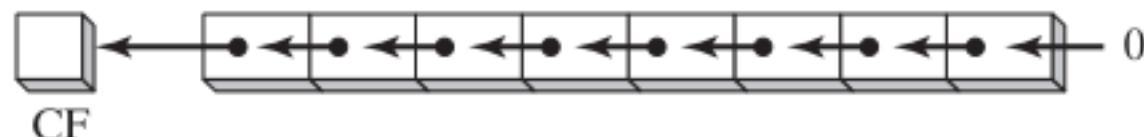
# *Arithmetic Shift Left/Arithmetic Shift Right*

---

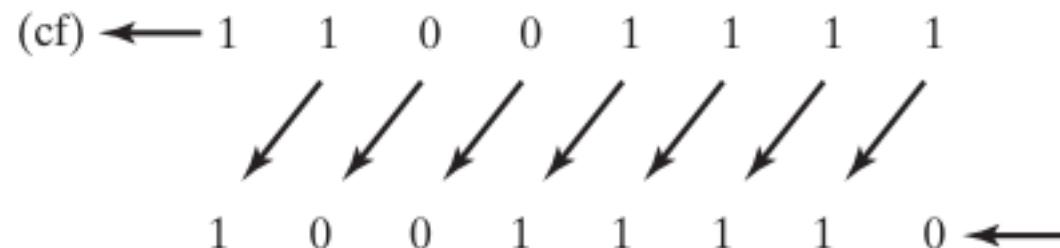
## *SAL (Shift Arithmetic Left) Instruction:*

---

The **SAL instruction**, also known as **Shift Arithmetic Left**, is similar to the SHL (Shift Left) instruction in its behavior. It's used for moving the bits of the destination operand to the left by a specified number of bits.



**Shift Left Operation:** SAL shifts each bit in the destination operand to the next highest bit position. In other words, it moves the bits leftward. If you shift binary 11001111 to the left by one bit, it becomes 10011110:



**Carry Flag:** As the bits shift to the left, the highest bit in the destination operand is moved into the Carry flag. The Carry flag stores the value that was in the most significant bit (MSB) position.

**Lowest Bit:** The lowest bit in the destination operand is assigned the value 0. This means that the bit that was in the least significant bit (LSB) position is set to 0.

**Discarding the Carry Bit:** The bit that was in the Carry flag is effectively discarded in this operation. It is no longer a part of the destination operand.

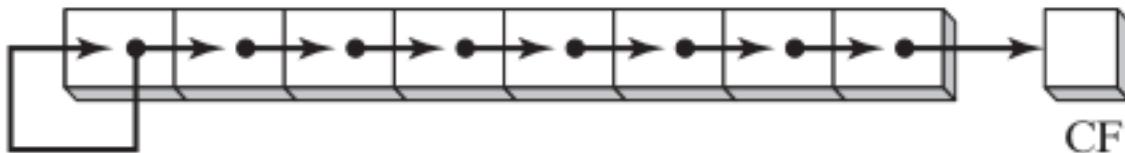
The SAL instruction is typically used to perform logical left shifts and, like SHL, is often used for operations such as multiplying a value by powers of 2. It's a valuable tool for bit manipulation in assembly language programming.

=====

### ***SAR (Shift Arithmetic Right) Instruction:***

=====

The SAR instruction, which stands for Shift Arithmetic Right, is used for shifting the bits of the destination operand to the right by a specified number of bits. Unlike logical right shifts (as in SHR), SAR is used for signed numbers to preserve the sign bit.



**Shift Right Operation:** SAR performs a right shift on the bits of the destination operand, moving them to the right.

### SAR destination, count

**Sign Bit Preservation:** The key feature of SAR is that it preserves the sign bit. This means that when a signed number is shifted right, the most significant bit (MSB), which indicates the sign (0 for positive, 1 for negative), is preserved during the shift.

**Carry Flag:** The lowest bit of the destination operand is moved into the Carry flag, similar to other shift instructions. The Carry flag now stores the value that was in the least significant bit (LSB) position.

**Bit Insertion:** The highest bit position, which was shifted out, is determined by the original sign bit. If the original sign bit was 0 (indicating a positive number), the highest bit is set to 0. If the original sign bit was 1 (indicating a negative number), the highest bit is set to 1. This preserves the signedness of the number.

SAR is used for operations like signed division by powers of 2 and other signed arithmetic operations. It's a crucial instruction in working with signed numbers in assembly language programming, ensuring that the sign bit is maintained correctly during shifts.

=====

### **Examples:**

=====

The following example shows how **SAR duplicates the sign bit**. AL is negative before and after it is shifted to the right:

```
mov al,0F0h          ; AL = 11110000b (-16)
sar al,1           ; AL = 11111000b (-8), CF = 0
```

### **Signed Division**

You can divide a signed operand by a power of 2, using the SAR instruction. In the following example, -128 is divided by 23. The quotient is -16:

```
mov dl,-128          ; DL = 10000000b
sar dl,3           ; DL = 11110000b
```

### **Sign-Extend AX into EAX**

Suppose AX contains a signed integer and you want to extend its sign into EAX. First shift EAX 16 bits to the left, then shift it arithmetically 16 bits to the right:

```
08 mov ax, -128      ;EAX = ????FF80h  
09 shl eax, 16       ;EAX = FF800000h  
10 sar eax, 16       ;EAX = FFFFFFF80h
```

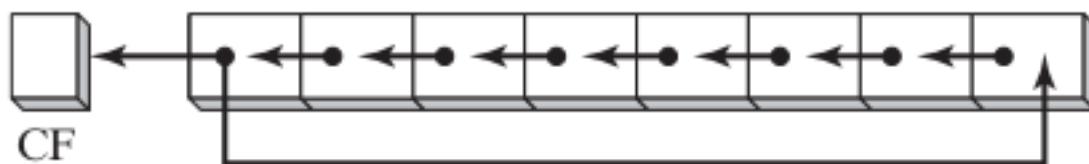
## *Rotate Left/Rotate Right*

---

### *ROL Instruction*

---

ROL (rotate left) instruction is used to perform bitwise rotation, where bits are shifted in a circular fashion. There are different ways to achieve this, including one where the bit leaving one end of the number is immediately copied into the other end.



ROL can also use the Carry flag as an intermediate point for shifted bits.

When you use the ROL instruction, each bit in the operand is shifted to the left. The highest bit (the one farthest to the left) is copied into **both the Carry flag and the lowest bit position** (fart-

hest to the right). The instruction format is similar to that of the SHL instruction.

**Preservation of Bits:** Bit rotation is unique in that it doesn't lose any bits during the process. If a bit is rotated off one end of a number, it appears again at the other end. For example, let's consider the following code:

```
13 mov al, 40h ; AL = 01000000b
14 rol al, 1    ; AL = 10000000b, CF = 0
15 rol al, 1    ; AL = 00000001b, CF = 1
16 rol al, 1    ; AL = 00000010b, CF = 0
```

In this example, we start with the value `01000000b` in AL, and after three rotations, the high bit is copied into both the Carry flag (CF) and bit position 0.

**Multiple Rotations:** When you use a rotation count greater than 1, the Carry flag contains the last bit rotated out of the most significant bit (MSB) position. For instance:

```
20 mov al, 00100000b
21 rol al, 3 ; CF = 1, AL = 00000001b
```

Here, when we rotate AL three times to the left, the Carry flag (CF) holds the last bit shifted out of the most significant bit position.

**Exchanging Groups of Bits:** ROL can be employed to exchange groups of bits within a byte. For example, by rotating a value like `26h` four bits in either direction, you can effectively swap the upper (bits 4-7) and lower (bits 0-3) halves of a byte:

```
26 mov al, 26h  
27 rol al, 4 ; AL = 62h
```

Furthermore, when you rotate a multibyte integer by four bits, you are effectively shifting each hexadecimal digit one position to the right or left. For instance:

```
31 mov ax, 6A4Bh  
32 rol ax, 4 ; AX = A4B6h  
33 rol ax, 4 ; AX = 4B6Ah  
34 rol ax, 4 ; AX = B6A4h  
35 rol ax, 4 ; AX = 6A4Bh
```

In this sequence, you can observe how each four-bit chunk of the value is rotated, eventually returning to the original value.

These instructions provide valuable tools for bit manipulation and data reordering in assembly language programming.

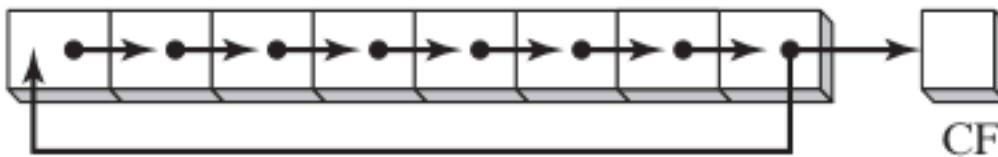
---

### **ROR Instruction**

---

The **ROR (rotate right) instruction** is used to perform bitwise rotation to the right. In this operation, each bit is shifted to the right, and the lowest bit (the one farthest to the right) is copied into both the Carry flag (CF) and the highest bit position (the one farthest to the left).

The instruction format for ROR is the same as that for SHL.



When you use the ROR instruction, each bit in the operand is shifted to the right. The lowest bit is copied into both the Carry flag (CF) and the highest bit position. Here's an example to illustrate this:

```
39 mov al, 01h ; AL = 00000001b
40 ror al, 1    ; AL = 10000000b, CF = 1
41 ror al, 1    ; AL = 01000000b, CF = 0
```

In this example, the initial value in AL is 00000001b, and after two rotations to the right, the lowest bit is copied into both the Carry flag (CF) and the highest bit position.

**Multiple Rotations:** When you perform multiple rotations with a count greater than 1, the Carry flag (CF) contains the last bit rotated out of the least significant bit (LSB) position. For example:

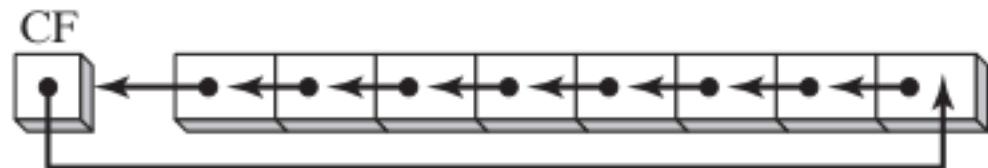
```
46 mov al, 00000100b
47 ror al, 3 ; AL = 10000000b, CF = 1
```

In this case, after three right rotations, the lowest bit ( $00000100b$ ) is shifted to the right, and the Carry flag (CF) holds the last bit, which is 1.

The ROR instruction is valuable for tasks that involve bitwise rotation to the right and is essential for various bit manipulation operations in assembly language programming.

### **RCL(Rotate Carry Left) Instruction**

**RCL (Rotate Carry Left) Instruction:** The RCL instruction is used to perform bitwise rotation to the left, similar to the ROL instruction.



However, it has the additional feature of incorporating the Carry flag (CF) as an extra bit at the high end of the operand. Here's how it works:

- Each bit is shifted to the left, and the Carry flag is copied to the least significant bit (LSB).
- The most significant bit (MSB) is then copied into the Carry flag.

Example:

```
52 clc          ;CF = 0
53 mov bl, 88h    ;CF,BL = 0 10001000b
54 rcl bl, 1      ;CF,BL = 1 00010000b
55 rcl bl, 1      ;CF,BL = 0 00100001b
```

In this example, the initial value of BL is 10001000b, and after two RCL instructions, the bits are shifted to the left, with the CF acting as an extra bit and being involved in the rotation.

### *Recovering a Bit from the Carry Flag*

RCL can be used to recover a bit that was previously shifted into the Carry flag.

In the following example, it checks the lowest bit of the "testval" variable by shifting its lowest bit into the Carry flag.

If the lowest bit is 1, a jump is taken; otherwise, RCL restores the number to its original value:

```
58 .data
59     testval BYTE 01101010b
60 .code
61     shr testval, 1 ; Shift LSB into Carry flag
62     jc exit        ; Exit if Carry flag is set
63     rcl testval, 1 ; Restore the number
```

Here, the SHR instruction shifts the LSB into the Carry flag, and if it's 1, it takes a jump;

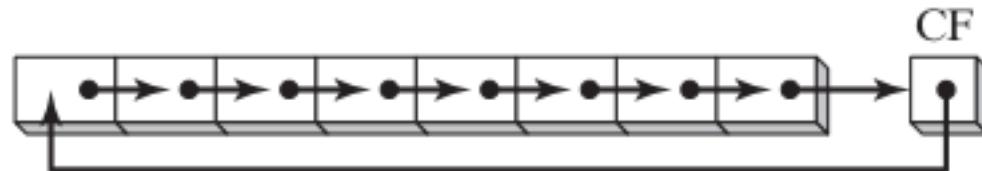
otherwise, RCL restores the number.

=====

### ***RCR (Rotate Carry Right)***

=====

The RCR instruction is similar to RCL but rotates bits to the right. Here's how it operates: Each bit is shifted to the right, and the Carry flag is copied into the **most significant bit (MSB)**.



The least significant bit (LSB) is copied into the Carry flag.

Example:

```
068 stc      ;CF = 1
069 mov ah, 10h ;AH, CF = 00010000 1
070 rcr ah, 1   ;AH, CF = 10001000 0
```

In this example, the STC instruction sets the Carry flag to 1, and then AH is rotated to the right, incorporating the Carry flag in the process.

These instructions are crucial for bit manipulation and advanced operations, allowing you to shift and rotate bits while incorporating the Carry flag for more intricate calculations in assembly language programming.

=====

### *Visualizing the integer*

=====

When working with rotate and shift instructions, visualizing the integer as a 9-bit value can be helpful.

In this representation, the Carry flag is positioned to the right of the least significant bit (LSB), acting as an extra bit during the rotation.

#### *Example: RCR (Rotate Carry Right)*

In the following code example, the STC instruction sets the Carry flag to 1, and then a rotate carry right operation is performed on the AH register:

```
076 stc    ; CF = 1
077 mov ah, 10h ; AH, CF = 00010000 1
078 rcr ah, 1 ; AH, CF = 10001000 0
```

This code demonstrates how the Carry flag is used during the right rotation, influencing the resulting bits in AH.

## *Signed Overflow*

The Overflow flag (OF) is set if the act of shifting or rotating a signed integer by one bit position generates a value outside the signed integer range of the destination operand.

In other words, this means that the sign of the number is reversed. Two examples illustrate this concept:

### Positive Integer Becoming Negative (ROL):

A positive integer (+127) stored in an 8-bit register becomes negative (-2) when rotated left.

```
mov al,+127          ; AL = 0111111b
rol al,1             ; OF = 1, AL = 11111110b
```

Initially:

**AL** = **0111111b**

After rotation:

**OF = 1, AL = 11111110b**

**Negative Integer Changing Sign (SHR):**

When -128 is shifted one position to the right, the Overflow flag is set.

The result in AL (+64) has the opposite sign.

```
mov al,-128          ; AL = 10000000b
shr al,1             ; OF = 1, AL = 01000000b
```

Initially:

**AL = 10000000b**

After shift:

**OF = 1, AL = 01000000b**

The Overflow flag is set when the signed integer range is exceeded during a shift or rotation by one bit position.

The value of the Overflow flag is undefined when the shift or rotation count is greater than 1.

It's important to note that the value of the Overflow flag is undefined when the shift or rotation count is greater than 1.

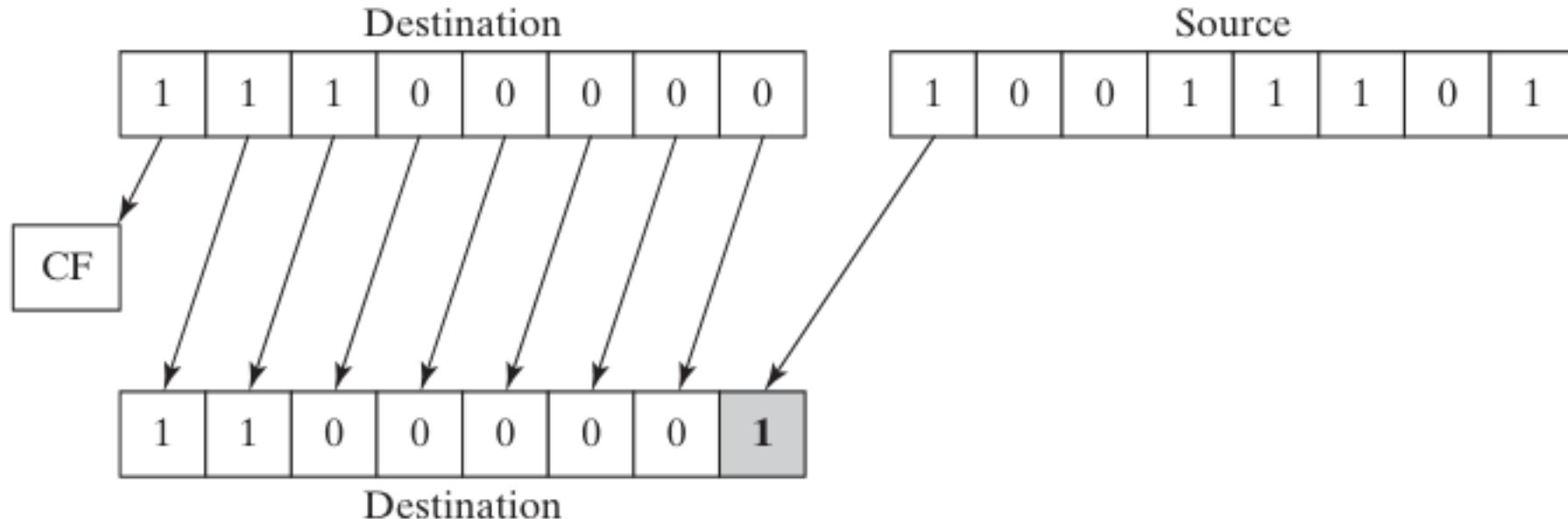
## ***Shift Left Double and Shift Right Double***

---

### ***SHLD (Shift Left Double)***

---

The **SHLD instruction** shifts a destination operand a given number of bits to the left, and fills the opened-up bit positions with the most significant bits of the source operand. The source operand is not affected.



The SHLD instruction has the following syntax:

**SHLD dest, source, count**

where:

- dest is the destination operand.
- source is the source operand.
- count is the number of bits to shift.

The count operand must be a value between 0 and 31, inclusive.

If count is 0, the destination operand is not shifted.

If count is 31, the destination operand is shifted all the way to the left, and the source operand is copied into the destination operand.

The following table shows the effects of the SHLD instruction on the Sign, Zero, Auxiliary, Parity, and Carry flags:

Flag	Before	After
Sign	Sign of the destination operand	Sign of the shifted destination operand
Zero	Zero flag set if the shifted destination operand is 0	Zero flag set if the shifted destination operand is 0
Auxiliary	Auxiliary carry flag set if the carry out of bit 3 is 1	Auxiliary carry flag set if the carry out of bit 3 is 1
Parity	Parity flag set if the shifted destination operand has an even number of 1 bits	Parity flag set if the shifted destination operand has an even number of 1 bits
Carry	Carry flag set if the carry out of bit 31 is 1	Carry flag set if the carry out of bit 31 is 1

Here is an example of how to use the SHLD instruction:

```
21 mov eax, 0x12345678  
22 mov ebx, 0xabcdef00  
23 SHLD eax, ebx, 1
```

After the SHLD instruction, eax will contain the value 0x23456780.

The SHLD instruction can be used to perform a variety of tasks, such as:

- Shifting a value to the left to multiply it by a power of two.
- Shifting a value to the left to extract the most significant bits.
- Shifting a value to the left to prepare it for a bitwise operation.

=====

### ***SHLD (Shift Right Double)***

=====

The SHRD instruction shifts a destination operand a given number of bits to the right, and fills the opened-up bit positions with the least significant bits of the source operand. The source operand is not affected.

The SHRD instruction has the following syntax:

## **SHRD dest, source, count**

where:

- dest is the destination operand.
- source is the source operand.
- count is the number of bits to shift.

The count operand must be a value between 0 and 31, inclusive.

If count is 0, the destination operand is not shifted.

If count is 31, the destination operand is shifted all the way to the right, and the source operand is copied into the destination operand.

The following table shows the effects of the SHRD instruction on the Sign, Zero, Auxiliary, Parity, and Carry flags:

Flag	Before	After
Sign	Sign of the destination operand	Sign of the shifted destination operand
Zero	Zero flag set if the shifted destination operand is 0	Zero flag set if the shifted destination operand is 0
Auxiliary	Auxiliary carry flag set if the carry out of bit 3 is 1	Auxiliary carry flag set if the carry out of bit 3 is 1
Parity	Parity flag set if the shifted destination operand has an even number of 1 bits	Parity flag set if the shifted destination operand has an even number of 1 bits
Carry	Carry flag set if the carry out of bit 0 is 1	Carry flag set if the carry out of bit 0 is 1

Here is an example of how to use the SHLD instruction:

```

41 mov eax, 0x12345678
42 mov ebx, 0xabcddef00
43 SHRD eax, ebx, 1
44
45 ;After the SHRD instruction, eax will contain the value 0x092a3c40.

```

The SHRD instruction can be used to perform a variety of tasks, such as:

- Shifting a value to the right to divide it by a power of two.
- Shifting a value to the right to extract the least significant bits.
- Shifting a value to the right to prepare it for a bitwise operation.

The SHRD instruction is a logical instruction that is used to shift a destination operand to the right by a specified number of bits, and then fills the vacated bit positions with the least significant bits of the source operand. The source operand is not affected by the instruction.

The SHRD instruction can be used to perform a variety of tasks, including:

- **Dividing a value by a power of two:** The SHRD instruction can be used to divide a value by a power of two by shifting the value to the right by the number of bits equal to the power of two. For example, to divide a value by 2, the value would be shifted to the right by 1 bit. To divide a value by 4, the value would be shifted to the right by 2 bits, and so on.
- **Extracting the least significant bits of a value:** The SHRD instruction can be used to extract the least significant bits of a value by shifting the value to the right by a number of bits equal to the number of least significant bits that need to be extracted. For example, to extract the least significant 4 bits of a value, the value would be shifted to the right by 4 bits.
- **Preparing a value for a bitwise operation:** The SHRD instruction can be used to prepare a value for a bitwise operation by shifting the value to the right by a number of bits equal to the number of bits that need to be aligned to the right. For example, to align a value to the right by 4 bits, the value would be shifted to the right by 4 bits.

The following instruction formats apply to both SHLD and SHRD. The destination operand can be a register or memory operand, and the source operand must be a register. The count operand can be the CL register or an 8-bit immediate operand:

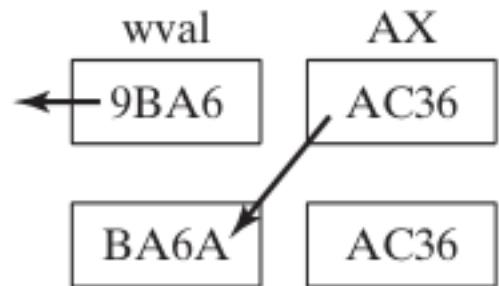
```
48 SHLD reg16,reg16, CL/imm8  
49 SHLD mem16,reg16, CL/imm8  
50 SHLD reg32,reg32, CL/imm8  
51 SHLD mem32,reg32, CL/imm8
```

### *Example 1:*

The following statements shift wval to the left 4 bits and insert the high 4 bits of AX into the low 4 bit positions of wval:

```
57 .data  
58     wval WORD 9BA6h  
59 .code  
60     mov  
61     ax,0AC36h  
62     shld  
63     wval,ax,4      ;wval = BA6Ah
```

The data movement is shown in the following figure:



### **Example 2:**

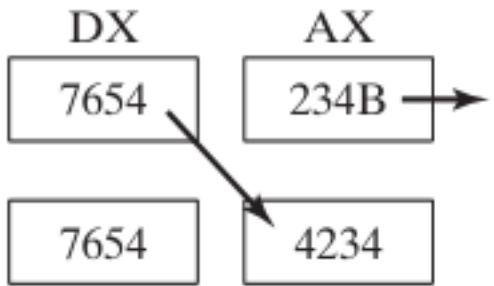
In the following example, AX is shifted to the right 4 bits, and the low 4 bits of DX are shifted into the high 4 positions of AX:

```

66 mov ax,234Bh
67 mov dx,7654h
68 shrd
69 ax,dx,4

```

The data movement is shown in the following figure:



### ***Example 3:***

The code below is an example of using the SHRD instruction to shift an array of doublewords to the right by 4 bits. This is a common operation in low-level programming when dealing with data manipulation, encryption, or even fast multiplication and division with very long integers. Let's break down the code and explain it step by step:

```
072 .data
073     array DWORD 648B2165h,8C943A29h,6DFA4B86h,91F76C04h,8BAF9857h
074 .code
075     mov bl,4
076     ; shift count
077     mov esi,OFFSET array
078     ; offset of the array
079     mov ecx,(LENGTHOF array) - 1
080     ; number of array elements
081
082 L1:
083     push ecx
084     ; save loop counter
085     mov eax,[esi + TYPE DWORD]
086     mov cl,bl
087     ; shift count
088     shrd [esi],eax,cl
089     ; shift EAX into high bits of [ESI]
090     add esi,TYPE DWORD
091     ; point to the next doubleword pair
092     pop ecx
093     ; restore loop counter
094     loop L1
095     shr DWORD PTR [esi],COUNT
096     ; shift the last doubleword
```

In the .data section, you define an array of doublewords (DWORD). Each doubleword contains a hexadecimal value. This array represents the data that you want to shift.

In the .code section, you set up some initial values:

**bl** is set to 4, which represents the shift count (you want to shift the data by 4 bits).

**esi** is loaded with the offset of the array.

**ecx** is set to the number of elements in the array minus 1 (using LENGTHOF array - 1). This **loop counter** will be used to iterate through the array.

The core of the code is a loop labeled L1. Here's what happens inside the loop:

**push ecx** saves the loop counter on the stack.

**mov eax, [esi + TYPE DWORD]** loads the doubleword at the current **esi** offset into the **eax** register.

**mov cl, bl** loads the shift count into the **cl** register.

**shrd [esi], eax, cl** performs the shift operation, shifting the value in **eax** to the right by the number of bits specified in **cl**.

This result is stored back in the memory location pointed to by **esi**.

**add esi, TYPE DWORD** moves **esi** to the next doubleword in the array.

`pop ecx` restores the loop counter.

`loop L1 decrements ecx (the loop counter)` and jumps back to `L1` as long as `ecx` is not zero. This loop processes each doubleword in the array.

Finally, after the loop, you perform a shift on the last doubleword using the `shr` instruction.

This code effectively shifts the entire array of doublewords to the right by 4 bits.

=====

## **Questions**

=====

**Which instruction shifts each bit in an operand to the left and copies the highest bit into both the Carry flag and the lowest bit position?**

Answer: The instruction that performs this operation is `SHL` (Shift Left) or `SAL` (Shift Arithmetic Left).

**Which instruction shifts each bit to the right, copies the lowest bit into the Carry flag, and copies the Carry flag into the highest bit position?**

Answer: The instruction that shifts each bit to the right, copies the lowest bit into the Carry flag, and copies the Carry flag into the highest bit position is `SHR` (Shift Right).

**Which instruction performs the following operation (CF = Carry flag)?**

Answer: The operation you described is performed by the RCL (Rotate through Carry Left) instruction. It rotates the bits to the left through the Carry flag, as indicated.

**What happens to the Carry flag when the SHR AX,1 instruction is executed?**

Answer: When the SHR AX,1 instruction is executed, the Carry flag (CF) will receive the value of the least significant bit (LSB) of the AX register before the shift, and the LSB itself will be shifted out of the AX register.

**Challenge:** Write a series of instructions that shift the lowest bit of AX into the highest bit of BX without using the SHRD instruction. Next, perform the same operation using SHRD.

Answer: Below are the series of instructions to achieve this without SHRD:

```
100 mov cx, 1      ; Set the shift count to 1
101 shl bx, 1      ; Shift left the bits in BX by 1 position
102 rcl bx, 1      ; Rotate through Carry Left (this moves the original LSB of AX to the MSB of BX)
```

To perform the same operation using SHRD:

```
shrd bx, ax, 1 ; Shift right double (moves LSB of AX to the MSB of BX)
```

**Challenge:** One way to calculate the parity of a 32-bit number in EAX is to use a loop that shifts each bit into the Carry flag and accumulates a count of the number of times the Carry flag was set. Write a code that does this, and set the Parity flag accordingly.

Answer:

```
111 xor ecx, ecx      ; Clear the counter
112 mov ebx, eax      ; Make a copy of EAX
113 parity_loop:
114 shr ebx, 1         ; Shift right by 1 bit
115 adc ecx, 0         ; Add the Carry flag to the counter
116 jnz parity_loop   ; Jump back if there's still a 1 bit in EBX
117 test ecx, 1         ; Test the least significant bit of the counter
118 setp al             ; Set the Parity flag based on the counter
```

This code calculates the parity of a 32-bit number in EAX using a loop that shifts each bit into the Carry flag and accumulates a count of the number of times the Carry flag was set. It sets the Parity flag (PF) accordingly. If the count is even, PF will be set; if it's odd, PF will be cleared.

## *Shifting Multiple Doublewords*

---

### *Shifting Multiple Doublewords*

---

In the realm of assembly programming, you can manipulate extended-precision integers that are

organized into arrays of bytes, words, or doublewords.

However, it's imperative to understand how these array elements are stored.

A prevalent method of storing these integers is referred to as "little-endian order."

**In little-endian order**, the low-order byte is placed at the array's starting address, and then, as you progress from this byte to the high-order byte, each is consecutively stored in the next memory location.

This ordering holds true regardless of whether you're working with bytes, words, or doublewords because x86 machines consistently use little-endian order for all these data formats.

Now, let's delve into the specific steps for shifting an array of bytes one bit to the right:

### **Step 1:**

To accomplish this operation, you start by shifting the highest byte located at [ESI+2] to the right.

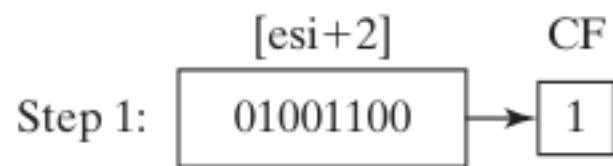
During this shift, the lowest bit of this byte is automatically copied into the Carry flag.

This behavior is a standard operation performed by instructions like SHR (Shift Right) in many assembly languages.

To visually demonstrate this, here's how you might express it in x86 assembly code, assuming that the ESI register holds the base address of the array:

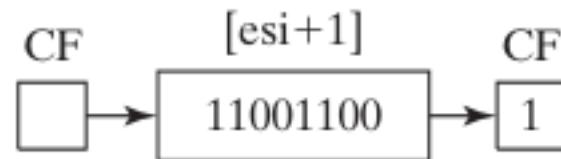
**SHR byte ptr [ESI+2], 1**

This instruction effectively shifts the byte at [ESI+2] one bit to the right, with the least significant bit being transferred to the Carry flag.



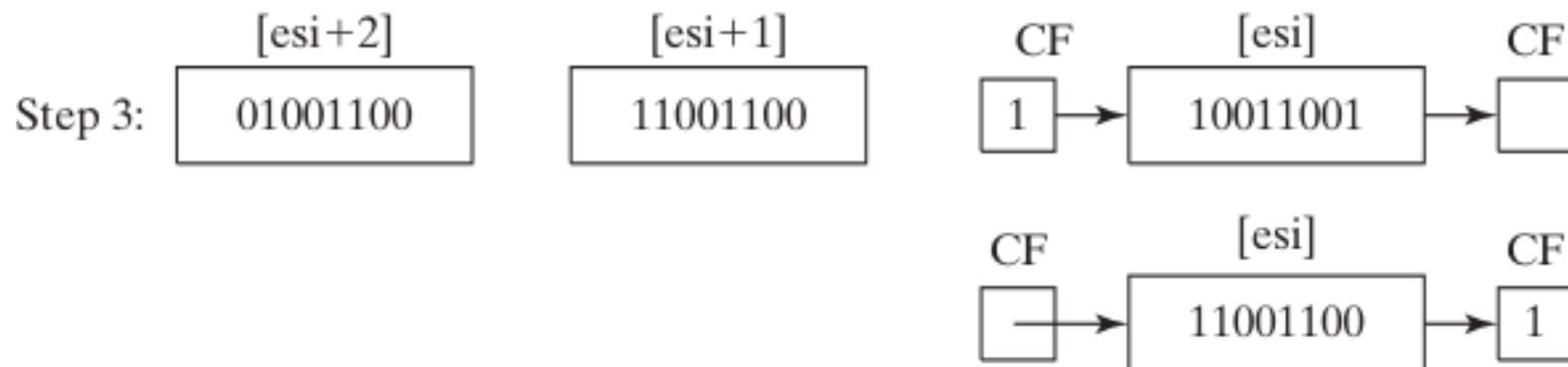
### ***Step 2:***

Rotate the value at [ESI+1] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:

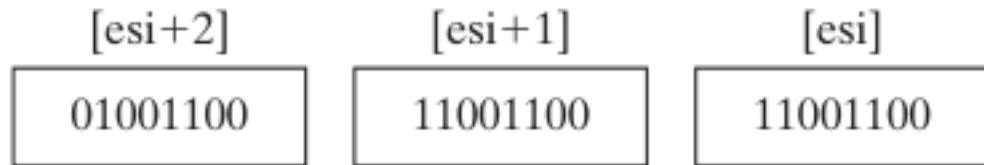


### **Step 3:**

Rotate the value at [ESI] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:



After Step 3 is complete, all bits have been shifted 1 position to the right:



```

130 .data
131 ArraySize = 3
132 array BYTE ArraySize DUP(99h) ; Initialize an array of 3 bytes with the value 99h (1001 1001 in binary).
133
134 .code
135 main PROC
136 mov esi, 0      ; Initialize the ESI register to 0, pointing to the beginning of the array.
137 shr array[esi+2], 1 ; Shift the high byte at array[2] one bit to the right.
138 rcr array[esi+1], 1 ; Rotate the middle byte at array[1] one bit to the right, including the Carry flag.
139 rcr array[esi], 1   ; Rotate the low byte at array[0] one bit to the right, including the Carry flag.

```

Here's a breakdown of what's happening in this code:

The **.data section** defines the **ArraySize** as 3 and initializes an array called **array** with three bytes, each containing the value **99h**. This means the binary representation of each byte is "1001 1001."

In the **.code section**, the **main** procedure begins.

**mov esi, 0** initializes the **ESI** register to 0, pointing to the first byte in the array.

**shr array[esi+2], 1** performs a right shift on the high byte at **array[2]**. This operation moves the most significant bit one position to the right, effectively shifting the entire byte to the right by one bit.

`rcr array[esi+1], 1` rotates the middle byte at `array[1]` one bit to the right. A rotate operation combines shifting with a rotation of the Carry flag. It effectively moves the least significant bit of the high byte into the least significant bit of the middle byte, and the Carry flag into the most significant bit of the high byte.

Similarly, `rcr array[esi], 1` rotates the low byte at `array[0]`, including the Carry flag. This rotates the bytes and carries the bit that was shifted out from the middle byte into the low byte.

As a result, this code has shifted the entire array of bytes by one bit to the right, and the Carry flag holds the value of the bit that was shifted out from the low byte.

The code can indeed be adapted to handle arrays of words or doublewords by changing the data types and adjusting the number of bytes processed. Additionally, by using a loop, you can shift arrays of arbitrary size efficiently.

## ***Binary Shifting and Multiplication***

Let's discuss the concept of binary multiplication in assembly language, specifically focusing on optimizing integer multiplication through bit shifting rather than using the MUL instruction.

### ***Bit Shifting for Multiplication:***

In assembly programming, the SHL (Shift Left) instruction is often used to perform unsigned multiplication when the multiplier is a power of 2.

Shifting an unsigned integer left by ' $n$ ' bits is equivalent to multiplying it by  $2^n$ .

This is a highly efficient way to perform multiplication when the multiplier is a power of 2 because it can be achieved through simple bit shifting operations, which are faster than a full multiplication.

### ***Representing Non-Power-of-2 Multipliers:***

For multipliers that are not powers of 2, you can express them as a sum of powers of 2. This allows you to break down the multiplication into simpler bit-shifting operations.

For example, to multiply a value in the EAX register by 36, you can represent 36 as  $2^5 + 2^2$  and apply the distributive property of multiplication as follows:

```
142 EAX * 36 = EAX * (2^5 + 2^2)
143           = EAX * (32 + 4)
144           = (EAX * 32) + (EAX * 4)
```

This decomposition enables you to perform two separate bit-shifting operations: one to multiply by 32 (shifting left by 5 bits) and another to multiply by 4 (shifting left by 2 bits). These individual results can then be added to obtain the final product.

### ***Example: 123 \* 36:***

The passage mentions an example of multiplying 123 by 36, resulting in 4428. This multiplication can be achieved using the approach described above. In this case, you would perform two separate bit-shifting operations.

$$\begin{array}{r}
 & 01111011 & 123 \\
 \times & 00100100 & \\
 \hline
 & 01111011 & 123 \text{ SHL } 2 \\
 + & 01111011 & 123 \text{ SHL } 5 \\
 \hline
 0001000101001100 & 4428
 \end{array}$$

146  $123 * 32 (2^5) = 3936$

147  $123 * 4 (2^2) = 492$

```

151 mov    eax, 123
152 mov    ebx, eax
153 shl    eax, 5      ; multiply by 2^5
154 shl    ebx, 2      ; multiply by 2^2
155 add    eax, ebx

```

This code snippet first moves the value 123 into the register EAX.

Then, it moves a copy of EAX into the register EBX.

Next, it shifts EAX left by 5 bits, which is equivalent to multiplying it by 25.

It then shifts EBX left by 2 bits, which is equivalent to multiplying it by 22.

Finally, it adds the two products together in the register EAX.

To generalize this example and create a procedure that multiplies any two 32-bit unsigned integers using shifting and addition, we can use the following algorithm:

- Initialize the product register to 0.
- For each bit in the multiplier, starting with the least significant bit:
- If the bit is set, shift the multiplicand left by the corresponding number of bits and add it to the product register.
- If the bit is not set, do nothing.
- Return the product register.

The following pseudocode shows this algorithm:

```
157 #include <stdio.h>
158
159 int multiply(int multiplicand, int multiplier) {
160     int product = 0;
161
162     for (int i = 0; i < 32; i++) {
163         if ((multiplier & (1 << i)) != 0) {
164             product += (multiplicand << i);
165         }
166     }
167
168     return product;
169 }
170
171 int main() {
172     int multiplicand = 123; // Replace with your values
173     int multiplier = 36;    // Replace with your values
174
175     int result = multiply(multiplicand, multiplier);
176
177     printf("Result: %d\n", result);
178     return 0;
179 }
```

We define a function `multiply` that takes two integers, `multiplicand` and `multiplier`, as parameters and returns the product as an integer.

Inside the function, we initialize `product` to 0, which will hold the result.

We use a `for` loop to iterate from 0 to 31 to examine each bit of the `multiplier`.

Within the loop, we use bitwise operations to check if the  $i$ -th bit of the `multiplier` is set (1).

If it is, we shift the `multiplicand` left by  $i$  bits and add the result to the `product`.

Finally, the function returns the computed `product`.

In the `main` function, you can replace the values of `multiplicand` and `multiplier` with the numbers you want to multiply. When you run the program, it will calculate the product and print the result.

The following assembly code implements this algorithm:

```
183 multiply:  
184     push    ebp  
185     mov     ebp, esp  
186     mov     eax, multiplicand  
187     mov     ebx, multiplier  
188     xor     ecx, ecx  
189     mov     ecx, 31  
190 loop:  
191     shl     eax, 1  
192     test    ebx, 1  
193     jz      next  
194     add     eax, ebx  
195 next:  
196     dec     ecx  
197     jnz     loop  
198     mov     esp, ebp  
199     pop     ebp  
200     ret
```

This assembly code appears to multiply two integers, `multiplicand` and `multiplier`, and the result is stored in the `EAX` register, which is a common convention for returning values in assembly language functions. Let's break down the code:

**push ebp:** This instruction saves the base pointer (EBP) on the stack to establish a new stack frame. This is a common practice at the beginning of a function.

**mov ebp, esp:** It sets the base pointer (EBP) to the current stack pointer (ESP). This establishes a new stack frame for the function.

**mov eax, multiplicand:** This instruction loads the value of multiplicand into the EAX register.

**mov ebx, multiplier:** It loads the value of multiplier into the EBX register.

**xor ecx, ecx:** This clears the ECX register to zero. It will be used as a loop counter.

**mov ecx, 31:** This sets ECX to 31, which is the loop iteration count. The loop will iterate for 32 bits (0 to 31).

**loop::** This is a label for the loop.

**shl eax, 1:** It shifts the EAX register left by 1 bit, effectively multiplying it by 2. This is performed in each iteration to handle the next bit of the multiplier.

**test ebx, 1:** This instruction tests the least significant bit of EBX (the multiplier) to check if it's set (1).

**jz next:** If the least significant bit of the multiplier is not set (i.e., it's zero), the code jumps to the next label without adding EBX to the result in EAX.

**add eax, ebx:** If the least significant bit of the multiplier is set, it adds EBX to EAX, effectively performing an addition in each iteration.

**next::** This label is used to continue with the next iteration of the loop.

**dec ecx:** It decrements the loop counter in ECX.

**jnz loop:** This instruction checks if ECX is not zero (meaning there are more bits to process in the multiplier). If it's not zero, the code jumps back to the loop label, continuing the multiplication process.

**mov esp, ebp:** This restores the stack pointer (ESP) to its previous value, effectively cleaning up the stack frame.

**pop ebp:** It restores the base pointer (EBP) to its previous value, completing the stack frame cleanup.

**ret:** This is the return instruction, and it returns the result in the EAX register.

Overall, the code is an assembly implementation of integer multiplication using bit-shifting and addition, and it follows a loop-based approach to handle each bit of the multiplier.

## ***BinToAsc***

```
206 BinToAsc:  
207     push    ebp  
208     mov     ebp, esp  
209     mov     eax, binary_integer  
210     xor     ecx, ecx  
211     mov     ecx, 31  
212 loop:  
213     shl     eax, 1  
214     adc     ecx, ecx  
215     mov     edx, eax  
216     cmp     dl, 32  
217     jb      ascii_zero  
218     mov     dl, dl - 32  
219 ascii_zero:  
220     mov     [edi], dl  
221     inc     edi  
222     dec     ecx  
223     jnz     loop  
224     pop     ebp  
225     ret
```

This procedure works by iterating over the bits in the binary integer, starting with the most significant bit.

For each bit, the procedure shifts the binary integer left by 1 bit and adds the carry flag to the

counter register (ECX).

The carry flag is used to keep track of whether the previous iteration resulted in a carry-out.

If the binary integer is less than 32, then the least significant bit will be 0 and the carry flag will be 0. In this case, the procedure will move the ASCII character '0' (0x30) to the buffer at the address specified by the register EDI.

If the binary integer is greater than or equal to 32, then the least significant bit will be 1 and the carry flag will be 1.

In this case, the procedure will move the ASCII character '1' (0x31) to the buffer at the address specified by the register EDI.

After the procedure has finished iterating over the bits in the binary integer, the buffer at the address specified by the register EDI will contain the ASCII binary string representation of the binary integer.

### ***Example 2 Usage:***

The following code snippet shows how to use the BinToAsc procedure to convert the binary integer 123 (01111011) to an ASCII binary string:

```
229 mov      eax, 123
230 call     BinToAsc
231
232 ;The buffer at the address specified by the register `EDI`
233 ;will now contain the ASCII binary string "01111011".
```

The BinToAsc procedure is a simple and efficient way to convert a binary integer to an ASCII binary string. It is useful for displaying binary data on the console or in a file.

## *Extracting File Date Fields*

**Shifting and masking:** The two most important operations used to extract bit strings are shifting and masking.

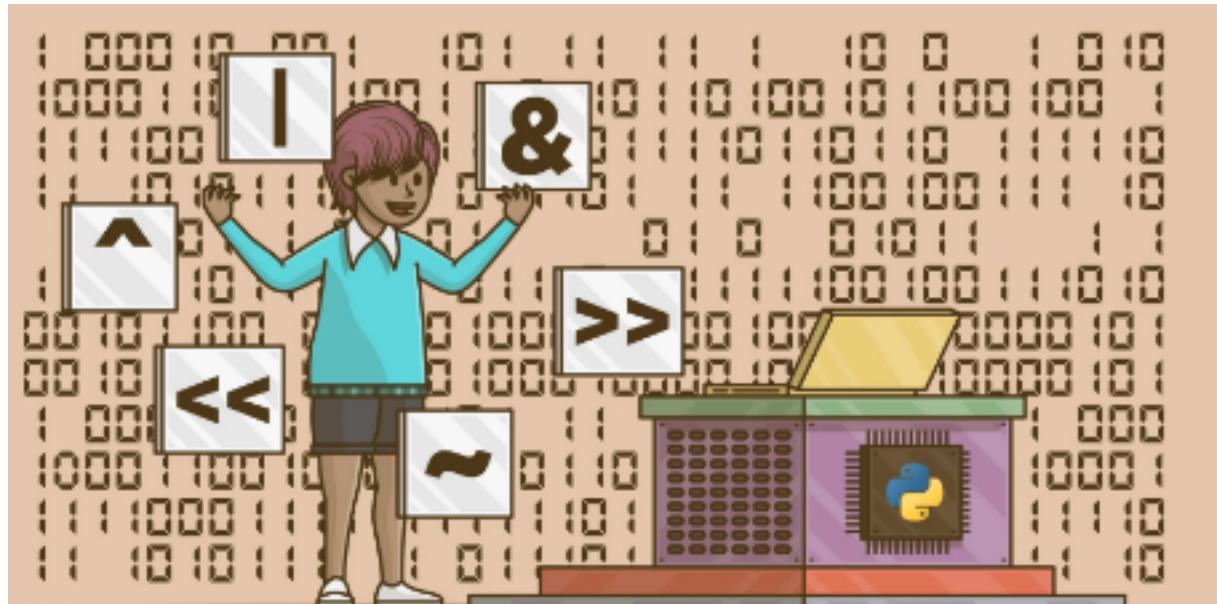
# BIT MASKING

32	
12 =	000000000000000000000000000000001100
1 =	000000000000000000000000000000000001
1 << 2 =	00000000000000000000000000000000000100
12 & (1 << 2) =	00000000000000000000000000000000000100 = 4

Shifting allows you to move the bit string to the desired position within a register, while masking allows you to clear any unwanted bits.



**Using the AX register:** The AX register is a convenient register to use for extracting bit strings, as it is 16 bits wide. This means that it can hold two 8-bit byte values.



This can be useful for extracting bit strings that are spread across two bytes, such as the month and day fields of a date stamp.

**Storing the extracted bit strings:** Once you have extracted the bit strings, you need to store them somewhere.

This can be done by copying them to other registers or to memory.

The following code snippet shows how to extract the day, month, and year fields of a date stamp integer stored in the DX register:

```
236 ;Make a copy of DL and mask off bits not belonging to the day field.  
237 mov al, dl  
238 and al, 00011111b  
239 mov day, al  
240  
241 ;Shift bits 5 through 8 into the low part of AL before masking off all other bits.  
242 mov ax, dx  
243 shr ax, 5  
244 and al, 00001111b  
245 mov month, al  
246  
247 ;Copy the year field from DH to AL and shift right by 1 bit to clear AH.  
248 mov al, dh  
249 shr al, 1  
250 mov ah, 0  
251 add ax, 1980  
252 mov year, ax
```

This code snippet first makes a copy of the DL register to the AL register. Then, it masks off all bits except for the day field (bits 0 through 4). Finally, it copies the masked value to the day variable.

Next, the code snippet shifts bits 5 through 8 of the DX register into the low part of the AX register. Then, it masks off all bits except for the month field (bits 5 through 8). Finally, it copies the masked value to the month variable.

Finally, the code snippet copies the year field (bits 9 through 15) from the DH register to the AL register. Then, it shifts the value right by 1 bit to clear the AH register.

Finally, it adds 1980 to the value to account for the fact that the year field is relative to 1980. The code snippet then copies the final value to the year variable.

Once the day, month, and year fields have been extracted, they can be used for any purpose, such as displaying the date or calculating the number of days since the file was last modified.

---

### 1. Write assembly language instructions that calculate EAX \* 24 using binary multiplication.

Here's how you can calculate EAX \* 24 in assembly language using binary multiplication:

```
254 mov ecx, 4      ; Initialize a counter for the number of bits to shift  
255 mov ebx, eax    ; Make a copy of the original value in EAX  
256 shl eax, 3      ; Multiply EAX by 2^3 (which is 8)  
257 add eax, ebx    ; Multiply the result by 3 (24 = 8 * 3)
```

### 2. Write assembly language instructions that calculate EAX \* 21 using binary multiplication.

Hint:  $21 = 24 - 22 - 20$ .

To calculate EAX \* 21, you can use binary multiplication based on the hint provided:

```
263 mov ebx, eax ; Copy the original value to EBX
264 shl eax, 3 ; Multiply EAX by 8 ( $2^3$ )
265 sub ebx, eax ; Subtract the original value by the result (EBX - EAX)
266 shl eax, 1 ; Multiply EAX by 2 ( $2^1$ )
267 add eax, ebx ; Add the result to the previous result (EAX + EBX)
```

3. What change would you make to the BinToAsc procedure in Section 7.2.3 in order to display the binary bits in reverse order?

To display the binary bits in reverse order in the BinToAsc procedure, you can modify the loop that processes the bits. Instead of starting from the most significant bit (bit 31) and moving towards the least significant bit (bit 0), you can reverse the loop to start from the least significant bit and move towards the most significant bit. Here's a modified version of the BinToAsc procedure:

```

271 BinToAsc PROC
272     pushad          ; Preserve registers
273     mov    edi, 31      ; Start from the least significant bit
274     mov    ecx, 32      ; Loop through all 32 bits
275     mov    esi, OFFSET outputStr ; Address of the output buffer
276
277 ConvertLoop:
278     mov    al, [ebx + edi/8]   ; Load a byte from the binary data
279     shl    al, cl           ; Shift the bit of interest to the lowest position
280     and    al, 1             ; Mask all bits except the lowest one
281     add    al, '0'           ; Convert the bit to its ASCII representation
282     stosb                  ; Store the character in the output buffer
283     loop   ConvertLoop
284     mov    byte ptr [esi], 0  ; Null-terminate the output string
285     popad                  ; Restore registers
286     ret
287 BinToAsc ENDP

```

In this modified version, we start with the least significant bit (bit 0) and iterate through the bits in reverse order, which will display the binary bits in reverse.

4. The time stamp field of a file directory entry uses bits 0 through 4 for the seconds, bits 5 through 10 for the minutes, and bits 11 through 15 for the hours. Write instructions that extract the minutes and copy the value to a byte variable named bMinutes.

Here are the assembly instructions to extract the minutes from the time stamp and store the value in a byte variable named bMinutes:

```
292 mov    edx, [DirectoryEntryTime] ; Load the directory entry time stamp (assuming it's in edx)
293 and    edx, 0x07E0           ; Mask out the bits for minutes (5 through 10)
294 shr    edx, 5              ; Shift the extracted minutes to the least significant bits
295 mov    byte ptr [bMinutes], dl ; Store the extracted minutes in bMinutes
```

In this code, we use the and and shr instructions to isolate and shift the bits representing the minutes in the directory entry time stamp.

Finally, we store the extracted minutes in the bMinutes byte variable. Please replace [DirectoryEntryTime] with the actual address of the time stamp in your program.

## **MUL Operator**

The MUL instruction performs unsigned integer multiplication.

It has three versions, which multiply an 8-bit operand by AL, a 16-bit operand by AX, or a 32-bit operand by EAX.

The multiplicand and multiplier must always be the same size, and the product is twice their size.

The following table shows the default multiplicand and product, depending on the size of the multiplier:

Multiplier size	Multiplicand	Product	Default destination operand	Register/memory operands
8 bits	AL	AX	AX	AL, reg/mem8
16 bits	AX	DX:AX	DX:AX	AX, reg/mem16
32 bits	EAX	EDX:EAX	EDX:EAX	EAX, reg/mem32
64 bits	RAX	RDX:RAX	RDX:RAX	RAX, reg/mem64

Because the destination operand is twice the size of the multiplicand and multiplier, overflow cannot occur.

However, the MUL instruction sets the Carry and Overflow flags if the upper half of the product is not equal to zero.

The Carry flag is ordinarily used for unsigned arithmetic, so it can be used to detect overflow in the MUL instruction.

For example, if AX is multiplied by a 16-bit operand, the product is stored in the combined DX and AX registers.

If DX is not equal to zero after the multiplication operation, then the product will not fit into

the lower half of the implied destination operand, and the Carry flag will be set.

Here is an example of how to use the MUL instruction to multiply two 16-bit operands:

```
298 mov ax, 1000h ; load first operand into AX  
299 mov bx, 2000h ; load second operand into BX  
300 mul bx ; multiply AX by BX
```

After the multiplication operation, the product will be stored in the combined DX and AX registers. If DX is not equal to zero, then the product will not fit into the lower half of the AX register, and the Carry flag will be set.

As you can see, the MUL instruction supports register and memory operands for all multiplier sizes. This gives you a lot of flexibility in how you use the instruction.

For example, the following assembly code multiplies the AL register by the 8-bit operand in memory location MY\_DATA:

```
303 mov al, 100h ; load first operand into AL  
304 mul MY_DATA ; multiply AL by the operand in MY_DATA
```

The following assembly code multiplies the EAX register by the 32-bit operand in memory location MY\_DATA:

```
306 mov eax, 10000000h ; load first operand into EAX  
307 mul MY_DATA ; multiply EAX by the operand in MY_DATA
```

The MUL instruction is a powerful tool for performing unsigned integer multiplication on the x86 architecture. It is important to understand the different versions of the instruction and how to use the Carry flag to detect overflow.

---

A good reason for checking the Carry flag after executing MUL is to know whether the upper half of the product can safely be ignored.

The MUL instruction multiplies two operands and stores the product in two registers. If the product is too large to fit in the destination registers, the Carry flag is set.

For example, if you multiply two 16-bit operands, the product will be 32 bits.

The MUL instruction will store the lower 16 bits of the product in the AX register and the upper 16 bits of the product in the DX register.

If the upper 16 bits of the product are zero, then you can safely ignore them. However, if the upper 16 bits of the product are non-zero, then you will need to use the DX register to store the entire product.

You can check the Carry flag to determine whether the upper half of the product is zero.

If the Carry flag is clear, then the upper half of the product is zero and you can safely ignore it.

However, if the Carry flag is set, then the upper half of the product is non-zero and you will need to use the DX register to store the entire product.

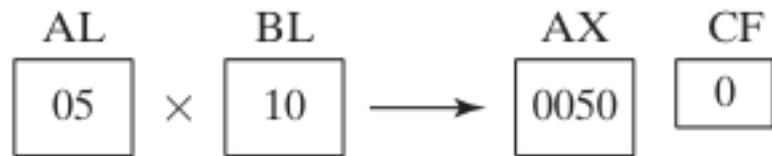
Here is an example of how to check the Carry flag after executing MUL:

```
311 mov ax, 1000h ;load first operand into AX
312 mov bx, 2000h ;load second operand into BX
313 mul bx ; multiply AX by BX
314
315 ;check the Carry flag
316 jc overflow ;jump to overflow handler if the Carry flag is set
317
318 ;the upper half of the product is zero, so we can ignore it
319 ;use the AX register to store the lower half of the product
```

*Example 1:*

```
324 ; 8-bit multiplication
325 mov al, 5h
326 mov bl, 10h
327 mul bl
328 ; AX = 0050h, CF = 0
329
330 ; 16-bit multiplication
331 mov ax, 2000h
332 mul val2
333 ; DX:AX = 00200000h, CF = 1
```

For this code, this is the data flow:



The following statements multiply 12345h by 1000h, producing a 64-bit product in the combined EDX and EAX registers:

```
339 mov eax, 12345h  
340 mov ebx, 1000h  
341 mul ebx  
342 ; EDX:EAX = 0000000012345000h, CF = 0
```

The MUL instruction multiplies two unsigned integers and stores the product in two registers: the low-order half of the product is stored in the EAX register, and the high-order half of the product is stored in the EDX register.

The Carry flag is set if the product is too large to fit in the destination registers.

In this case, the product of 12345h and 1000h is 12345000h, which is a 64-bit value. The product fits in the EDX and EAX registers, so the Carry flag is clear.

The following diagram illustrates the movement between registers:

```
347 Before:  
348 EAX = 12345h  
349 EBX = 1000h  
350 EDX = 0  
351  
352 After:  
353 EAX = 0000h  
354 EBX = 1000h  
355 EDX = 12345h
```



*Here is a summary of the MUL instruction:*

- The MUL instruction multiplies two unsigned integers and stores the product in two registers.
- The multiplicand (the first operand) is stored in the AL register (for 8-bit multiplication) or the AX register (for 16-bit multiplication).

- The multiplier (the second operand) is stored in another register or in memory.
- The product is stored in two registers: the low-order half of the product is stored in the AL register (or the AX register for 16-bit multiplication), and the high-order half of the product is stored in the AH register (or the DX register for 16-bit multiplication).
- The Carry flag is set if the product is too large to fit in the destination registers.

=====

### ***MUL in 64-bit mode***

=====

In 64-bit mode, the MUL instruction can be used to multiply two 64-bit operands. The result is a 128-bit product, which is stored in the RDX:RAX register pair.

The following example shows how to use the MUL instruction to multiply RAX by 2:

```
359 mov rax, 0FFFF0000FFFF0000h
360 mov rbx, 2
361 mul rbx
362
363 ; RDX:RAX = 0000000000000001FFFE0001FFFE0000h
```

In this example, the highest bit of RAX spills over into the RDX register because the product is too large to fit in a 64-bit register.

The following example shows how to use the MUL instruction to multiply RAX by a 64-bit memory operand:

```
367 .data
368     multiplier QWORD 10h
369
370 .code
371     mov rax, 0AABBBCCCDDDDh
372     mul multiplier
373
374 ;RDX:RAX = 0000000000000000AABBBCCCDDDD0h
```

In this example, the product is a **128-bit value**, but both halves of the product fit in RAX and RDX because the product is less than  $2^{128}$ .

Here is a more in-depth explanation of what happens when the MUL instruction is used in 64-bit mode:

The RAX and RDX registers are multiplied together. The low-order 64 bits of the product are stored in RAX. The high-order 64 bits of the product are stored in RDX. The Carry flag is set if the product is too large to fit in RAX and RDX.

## ***IMUL Operation***

The IMUL instruction performs signed integer multiplication.

This means that it multiplies two integers and takes into account their signs. Unlike the MUL instruction, the IMUL instruction preserves the sign of the product.

How does the IMUL instruction work?

The IMUL instruction works by first sign extending the highest bit of the lower half of the product into the upper bits of the product.

This ensures that the sign of the product is the same as the sign of the multiplicand (the number being multiplied).

What are the different formats of the IMUL instruction?

The IMUL instruction has three formats:

[378 One-operand format](#)

[379 Two-operand format](#)

[380 Three-operand format](#)

The **one-operand format of the IMUL instruction** multiplies the operand by itself and stores the product in the same operand. This is equivalent to squaring the operand.

The **two-operand format of the IMUL instruction** multiplies the two operands and stores the product in the first operand. The second operand can be a register, a memory operand, or an immediate value.

The **three-operand format of the IMUL instruction** multiplies the first and third operands and stores the product in the second operand. The first and third operands can be registers or memory operands, and the second operand must be a register.

### ***When should I use the IMUL instruction?***

You should use the IMUL instruction whenever you need to perform signed integer multiplication. This is especially important when you need to preserve the sign of the product.

### ***Here are some examples of how to use the IMUL instruction:***

```
382 ; One-operand format
383 imul eax
384 ; eax = eax * eax
385
386 ; Two-operand format
387 imul eax, ebx
388 ; eax = eax * ebx
389
390 ; Three-operand format
391 imul eax, ebx, ecx
392 ; eax = ebx * ecx
```

### ***Important things to keep in mind:***

- The IMUL instruction can generate overflow. If the product of the two operands is too large to fit in the destination operand, the Overflow flag is set.
- The IMUL instruction can also generate a carry. If the highest bit of the product is set, the Carry flag is set.
- It is important to check the Overflow and Carry flags after performing an IMUL operation to ensure that the result is correct.

```
396 ;Multiply two 16-bit integers and store the product in AX
397 mov ax, 1000h
398 mov bx, 2000h
399 imul bx
400
401 ;Multiply two 32-bit integers and store the product in EDX:EAX
402 mov eax, 10000000h
403 mov ebx, 20000000h
404 imul ebx
405
406 ;Multiply a 32-bit integer by an 8-bit immediate value and store the product in AX
407 mov ax, 1000h
408 imul ax, 2
409
410 ;Multiply a 32-bit integer by a 32-bit immediate value and store the product in EDX:EAX
411 mov eax, 10000000h
412 imul eax, 20000000h
413
414 ;Multiply two 16-bit integers and store the product in memory
415 mov ax, 1000h
416 mov bx, 2000h
417 imul bx
418 mov [word_variable], ax
419
420 ;Multiply two 32-bit integers and store the product in memory
421 mov eax, 10000000h
422 mov ebx, 20000000h
423 imul ebx
424 mov [dword_variable], eax
```

The first two lines of code move the values 1000h and 2000h into the AX and BX registers, respectively. The third line then uses the IMUL instruction to multiply the two values and store the product in the AX register.

The next two lines of code do the same thing, but with 32-bit integers. The IMUL instruction in this case stores the product in the EDX:EAX register pair.

The next two lines of code multiply a 32-bit integer by an 8-bit and 32-bit immediate value, respectively. The IMUL instruction in these cases stores the product in the AX and EDX:EAX register pair, respectively.

The last two lines of code multiply two 16-bit and 32-bit integers and store the product in memory. The IMUL instruction in these cases stores the product in the word and dword variable, respectively.

It is important to note that the IMUL instruction can generate overflow. This means that if the product of the two operands is too large to fit in the destination operand, the Overflow flag is set. It is important to check the Overflow flag after performing an IMUL operation to ensure that the result is correct.

-----

```
428 ;Multiply two 32-bit integers and store the product in EDX:EAX
429 mov eax, 80000000h
430 mov ebx, 80000000h
431 imul ebx
432 jc overflow_label      ; Check the Overflow flag
433 jc carry_label        ; Check the Carry flag
434 ;No overflow or carry occurred, so the result is correct
435 ;Exit the program
436 int 3
437 overflow_label:
438 ;Overflow occurred, so the result is incorrect
439 ;Handle the overflow error
440 ;Exit the program
441 int 3
442 carry_label:
443 ;Carry occurred, but the result may still be correct
444 ;Check the highest bit of the product
445 test edx, edx
446 jz carry_ok
447 ;The highest bit of the product is set, so the result is incorrect
448 ;Handle the carry error
449 ;Exit the program
450 int 3
451 carry_ok:
452 ;The highest bit of the product is not set, so the result is correct
453 ;Continue with the program
```

The first two lines of code move the values 1000h and 2000h into the AX and BX registers, respectively. The third line then uses the IMUL instruction to multiply the two values and store the product in the AX register.

The next two lines of code do the same thing, but with 32-bit integers. The IMUL instruction in this case stores the product in the EDX:EAX register pair.

The next two lines of code multiply a 32-bit integer by an 8-bit and 32-bit immediate value, respectively. The IMUL instruction in these cases stores the product in the AX and EDX:EAX register pair, respectively.

The last two lines of code multiply two 16-bit and 32-bit integers and store the product in memory. The IMUL instruction in these cases stores the product in the word and dword variable, respectively.

It is important to note that the IMUL instruction can generate overflow. This means that if the product of the two operands is too large to fit in the destination operand, the Overflow flag is set. It is important to check the Overflow flag after performing an IMUL operation to ensure that the result is correct.

-----

```
457 ;Multiply two 32-bit integers and store the product in EDX:EAX
458 mov     eax, 80000000h      ;Load the first integer into EAX
459 mov     ebx, 80000000h      ;Load the second integer into EBX
460 imul    ebx              ;Multiply EAX by EBX; result in EDX:EAX
461 ;Check for overflow:
462 jc      overflow_label   ;Jump if the Overflow flag is set
463 ;Check for carry:
464 jc      carry_label       ;Jump if the Carry flag is set
465 ;No overflow or carry occurred, so the result is correct
466 ;Exit the program
467 int    3
468 overflow_label:
469 ;Overflow occurred, so the result is incorrect
470 ;Handle the overflow error
471 ;Exit the program
472 int    3
473 carry_label:
474 ;Carry occurred, but the result may still be correct
475 ;Check the highest bit of the product
476 test    edx, edx          ;Test the value in EDX
477 jz      carry_ok          ;Jump if the highest bit of the product is not set
478 ;The highest bit of the product is set, so the result is incorrect
479 ;Handle the carry error
480 ;Exit the program
481 int    3
482 carry_ok:
483 ;The highest bit of the product is not set, so the result is correct
484 ;Continue with the program
```

In this example, we multiply two 32-bit integers, `8000000h` and `8000000h`. The product of these two integers is `640000000h`, which is too large to fit in a 32-bit register. Therefore, the Overflow flag is set.

We then check the Carry flag. The Carry flag is set if the highest bit of the product is set. In this case, the highest bit of the product is not set, so the Carry flag is not set.

We then check the Overflow flag to see if overflow occurred. If overflow occurred, the result of the multiplication is incorrect. In this case, overflow occurred, so the result is incorrect.

We could handle the overflow error in a number of ways. For example, we could print an error message and exit the program. Or, we could try to scale the result down so that it fits in a 32-bit register.

If overflow did not occur, we need to check the Carry flag. If the Carry flag is set, the highest bit of the product is set. This may or may not indicate that the result is incorrect.

In this example, the highest bit of the product is not set, so the result is correct. We can then continue with the program.

It is important to note that this is just a simple example of how to illustrate the IMUL instruction overflow and carry flag in MASM. There are many other ways to handle overflow and carry errors.

=====

### ***IMUL in 64-Bit Mode***

=====

```
488 ;64-bit mode IMUL example  
489 mov rax,-4  
490 mov rbx,4  
491 imul rbx    ;RDX = 0xFFFFFFFFFFFFFFFh, RAX = -16
```

In this example, we multiply the 64-bit register RBX by the 64-bit register RAX. The product of these two integers is -64, which is a 128-bit value. The IMUL instruction in this case stores the product in the RDX:RAX register pair.

The RDX register stores the high-order 64 bits of the product, and the RAX register stores the low-order 64 bits of the product.

In this case, the high-order 64 bits of the product are all ones, so the RDX register is filled with the value 0xFFFFFFFF...FFFFh. The low-order 64 bits of the product are equal to -64, so the RAX register is filled with the value FFFFFFFFFFFFFFC0h.

```
495 ;64-bit mode IMUL example  
496 .data  
497     multiplicand QWORD -16  
498 .code  
499     imul rax, multiplicand, 4 ;RAX = FFFFFFFFFFFFFFC0 (-64)
```

In this example, we multiply the 64-bit memory operand multiplicand by the immediate value 4. The product of these two operands is -64, which is a 64-bit value. The IMUL instruction in this case

stores the product in the RAX register.

The multiplicand memory operand is defined in the data section of the program. It is a QWORD variable, which means that it is 64 bits wide. The immediate value 4 is a 32-bit value, but it is automatically promoted to 64 bits before the multiplication operation is performed.

The IMUL instruction in this case stores the product in the RAX register, which is a 64-bit register. Therefore, the RAX register will be filled with the value FFFFFFFFFFFFFC0h after the IMUL instruction is executed.

### ***Unsigned multiplication***

The IMUL instruction can also be used to perform unsigned multiplication. However, there is a small disadvantage to doing so: the Carry and Overflow flags will not indicate whether the upper half of the product is equal to zero.

This is because the IMUL instruction always sign-extends the product, even if the operands are unsigned. This means that the Carry and Overflow flags will be set if the high-order bit of the product is set, even if the product is a valid unsigned integer.

The IMUL instruction is a powerful instruction that can be used to perform signed and unsigned multiplication in 64-bit mode. However, it is important to be aware of the fact that the Carry and Overflow flags will not indicate whether the upper half of the product is equal to zero when performing unsigned multiplication.

### ***MUL Examples in Depth***

#### ***MUL Overflow***

The MUL instruction can generate overflow if the product of the two operands is too large to fit in the destination operand. For example, the following code will generate overflow:

```
mov ax, -32000  
imul ax, 2
```

The product of -32000 and 2 is -64000, which is too large to fit in a 16-bit register. Therefore, the Overflow flag will be set after the IMUL instruction is executed.

#### *MUL Signed and Unsigned Examples*

```
511 ;Multiply 48 by 4 and store the product in AX.  
512 mov al, 48  
513 mov bl, 4  
514 mul bl  
515 ;AX = 00C0h, OF = 1  
516  
517 ;Multiply -4 by 4 and store the product in AX.  
518 mov al, -4  
519 mov bl, 4  
520 mul bl  
521 ;AX = FFF0h, OF = 0  
522  
523 ;Multiply 48 by 4 and store the product in DX:AX.  
524 mov ax, 48  
525 mov bx, 4  
526 mul bx  
527 ;DX:AX = 000000C0h, OF = 0  
528  
529 ;Multiply 4,823,424 by -423 and store the product in EDX:EAX.  
530 mov eax, 4823424  
531 mov ebx, -423  
532 mul ebx  
533 ;EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

## ***Two-Operand IMUL Instructions***

The two-operand IMUL instruction uses a destination operand that is the same size as the multiplier. Therefore, it is possible for signed overflow to occur. Always check the Overflow flag after executing these types of IMUL instructions. The following code examples demonstrate two-operand IMUL instructions:

```
537 ;Multiply -16 by 2 and store the product in AX.  
538 mov ax, -16  
539 imul ax, 2  
540 ; AX = -32  
541  
542 ;Multiply -32 by 2 and store the product in AX.  
543 imul ax, 2  
544 ;AX = -64  
545  
546 ;Multiply -64 by word1 and store the product in BX.  
547 ;word1 is a 16-bit variable.  
548 mov bx, word1  
549 imul bx, word1, -16  
550 ;BX = word1 * -16  
551  
552 ;Multiply -64 by dword1 and store the product in BX.  
553 ;dword1 is a 32-bit variable.  
554 mov bx, dword1  
555 imul bx, dword1, -16  
556 ;BX = dword1 * -16  
557  
558 ;Multiply dword1 by -2000000000 and store the product in BX.  
559 ;This instruction will generate overflow because the product is too large to fit in a 32-bit register.  
560 imul bx, dword1, -2000000000  
561 ;signed overflow!
```

### *Three-Operand IMUL Instructions*

The three-operand IMUL instruction uses a destination operand that is the same size as the first operand. The first operand is multiplied by the second operand and the product is stored in the third operand. The following code examples demonstrate three-operand IMUL instructions:

```
568 ;Multiply word1 by -16 and store the product in BX.  
569 mov bx, word1  
570 imul bx, word1, -16  
571 ;BX = word1 * -16  
572  
573 ;Multiply dword1 by -16 and store the product in BX.  
574 mov bx, dword1  
575 imul bx, dword1, -16  
576 ;BX = dword1 * -16  
577  
578 ;Multiply dword1 by -2000000000 and store the product in BX.  
579 ;This instruction will generate overflow because the product is too large to fit in a 32-bit register.  
580 mov bx, dword1  
581 imul bx, dword1, -2000000000  
582 ;signed overflow!
```

The MUL and IMUL instructions are powerful instructions that can be used to perform signed and unsigned multiplication.

However, it is important to be aware of the fact that these instructions can generate overflow.

Always check the Overflow flag after executing these types of instructions to ensure that the result is correct.

## ***Measuring Execution Times***

The code example you provided shows how to use the GetMseconds procedure in the Irvine32 library to

measure the execution time of a program.

The **GetMseconds** procedure returns the number of system milliseconds that have elapsed since midnight.

To measure the execution time of a program, you would first call the GetMseconds procedure to record the start time.

Then, you would call the program whose execution time you wish to measure. Finally, you would call the GetMseconds procedure again to record the end time.

The difference between the end time and the start time is the execution time of the program.

The following code example shows how to use the GetMseconds procedure to measure the execution time of a simple program:

```
585 .data
586     startTime DWORD ?
587     procTime DWORD ?
588
589 .code
590     call GetMseconds
591     ; get start time
592     mov startTime, eax
593
594     ; call the program whose execution time you wish to measure
595     ; ...
596
597     call GetMseconds
598     ; get end time
599     sub eax, startTime
600     ; calculate the elapsed time
601     mov procTime, eax
602     ; save the elapsed time
```

The **variable procTime** will now contain the execution time of the program, in milliseconds.

You can use this technique to measure the execution time of any program, regardless of its complexity.

However, it is important to note that the overhead of calling the GetMseconds procedure twice is

insignificant when compared to the execution time of most programs.

---

## ***Relative Performance***

You can also use the GetMseconds procedure to measure the relative performance of two different code implementations.

To do this, you would measure the execution time of each implementation and then divide the execution time of the first implementation by the execution time of the second implementation.

The result will be a number that indicates the relative performance of the two implementations.

For example, the following code example shows how to measure the relative performance of two different sorting algorithms:

```
627 .data
628     startTime1 DWORD ?
629     procTime1 DWORD ?
630     startTime2 DWORD ?
631     procTime2 DWORD ?
632 .code
633     ;measure the execution time of the first sorting algorithm
634     call GetMseconds      ;get start time
635     mov startTime1, eax
636     ; call the first sorting algorithm ...
637     call GetMseconds
638     ; get end time
639     sub eax, startTime1
640     ; calculate the elapsed time
641     mov procTime1, eax
642     ; save the elapsed time
643     ; measure the execution time of the second sorting algorithm
644     call GetMseconds
645     ; get start time
646     mov startTime2, eax
647     ; call the second sorting algorithm
648     ; ...
649     call GetMseconds
650     ; get end time
651     sub eax, startTime2
652     ; calculate the elapsed time
653     mov procTime2, eax
654     ; save the elapsed time
655     ; calculate the relative performance of the two sorting algorithms
656     div procTime1, procTime2
657     ; the result is now in EAX
```

The EAX register will now contain the relative performance of the two sorting algorithms. A value of 1.0 indicates that the two sorting algorithms have the same performance.

A value greater than 1.0 indicates that the first sorting algorithm is faster than the second sorting algorithm. A value less than 1.0 indicates that the first sorting algorithm is slower than the second sorting algorithm.

You can use this technique to measure the relative performance of any two code implementations, regardless of their complexity.

-----

### ***Comparing MUL and IMUL to Bit Shifting in Depth***

In older x86 processors, there was a significant difference in performance between multiplication by bit shifting and multiplication using the MUL and IMUL instructions.

However, in recent processors, Intel has managed to greatly optimize the MUL and IMUL instructions, so that they now have the same performance as bit shifting for multiplication by powers of two.

The following code shows two procedures for multiplying a number by 36 using bit shifting and the MUL instruction:

```
661 ;Multiplies EAX by 36 using SHL, LOOP_COUNT times.
662 mult_by_shifting PROC
663 mov ecx, LOOP_COUNT
664 L1: push eax
665 ; save original EAX
666 mov ebx, eax
667 shl eax, 5
668 shl ebx, 2
669 add eax, ebx
670 pop eax
671 ; restore EAX
672 loop L1
673 ret
674 mult_by_shifting ENDP
675
676 ; Multiplies EAX by 36 using MUL, LOOP_COUNT times.
677 mult_by_MUL PROC
678 mov ecx, LOOP_COUNT
679 L1:
680 push eax
681 ; save original EAX
682 mov ebx, 36
683 mul ebx
684 pop eax
685 ; restore EAX
686 loop L1
687 ret
688 mult_by_MUL ENDP
```

The following code calls the `mult_by_shifting` procedure and displays the timing results:

```
692 .data
693     LOOP_COUNT = 0xFFFFFFFFh
694 .data
695     intval DWORD 5
696 .code
697     call
698     GetMseconds
699     ; get start time
700     mov
701     startTime,eax
702     mov
703     eax,intval
704     ; multiply now
705     call
706     mult_by_shifting
707     call
708     GetMseconds
709     ; get stop time
710     sub
711     eax,startTime
712     call WriteDec
713     ; display elapsed time
```

The code above, is a simple example of how to measure the execution time of a program using the GetMseconds procedure in the Irvine32 library. The program multiplies the integer 5 by 36 using the mult\_by\_shifting procedure, and then displays the execution time.

The two .data segments in the program are used to define two variables: LOOP\_COUNT and intval. LOOP\_COUNT is a constant that specifies the number of times to repeat the multiplication operation. intval is the integer that is multiplied by 36.

The reason for having two .data segments is not entirely clear. It is possible that the original author of the code was simply trying to organize the data in a logical way.

However, it is also possible that the author was trying to take advantage of some optimization in the Irvine32 library.

Regardless of the reason, it is not necessary to have two .data segments in this program. The two variables could be defined in the same .data segment without any problems.

Here is a revised version of the program with the two .data segments combined into one:

```
717 .data
718     LOOP_COUNT = 0xFFFFFFFFh
719     intval DWORD 5
720 .code
721     call
722     GetMseconds
723     ; get start time
724     mov
725     startTime,eax
726     mov
727     eax,intval
728     ; multiply now
729     call
730     mult_by_shifting
731     call
732     GetMseconds
733     ; get stop time
734     sub
735     eax,startTime
736     call WriteDec
737     ; display elapsed time
```

This revised version of the program works just as well as the original version, and it is more concise and easier to read.

- You can have as many segments for .data, .code, .bss/text.
  - Use segments wisely, grouping related data and code.
  - Avoid excessive segments for clarity and performance.
- 

On a legacy 4-GHz Pentium 4 processor, the `mult_by_shifting` procedure executed in 6.078 seconds, while the `mult_by_MUL` procedure executed in 20.718 seconds.

This means that using the MUL instruction was 241 percent slower. However, when running the same program on a more recent processor, the timings of both function calls were exactly the same.

This example shows that Intel has managed to greatly optimize the MUL and IMUL instructions in recent processors.

Therefore, there is no longer any need to use bit shifting for multiplication by powers of two.

In fact, using the MUL and IMUL instructions is generally preferred, as they are more readable and easier to maintain.

## ***DIV Instruction***

The following table shows the relationship between the dividend, divisor, quotient, and remainder for the DIV instruction:

Operand Size	Dividend	Divisor	Quotient	Remainder
8-bit	AX	reg/mem8	AL	AH
16-bit	DX:AX	reg/mem16	AX	DX
32-bit	EDX:EAX	reg/mem32	EAX	EDX

In 64-bit mode, the DIV instruction uses RDX:RAX as the dividend and permits the divisor to be a 64-bit register or memory operand. The quotient is stored in RAX, and the remainder is stored in RDX.

The table above shows the relationship between the dividend, divisor, quotient, and remainder for the DIV instruction.

- **Dividend** is the number being divided.
- **Divisor** is the number by which the dividend is being divided.
- **Quotient** is the result of dividing the dividend by the divisor.
- **Remainder** is the number that is left over after the dividend is divided by the divisor.

The table shows that the operand size of the dividend and divisor determines the operand size of the quotient and remainder.

For example, if the dividend and divisor are 8-bit integers, then the quotient and remainder will

also be 8-bit integers.

The table also shows that the dividend and divisor can be stored in registers or memory.

For example, the dividend can be stored in the AX register, and the divisor can be stored in the BL register.

Here is an example of how to use the DIV instruction to perform 8-bit unsigned division:

### **DIV Examples**

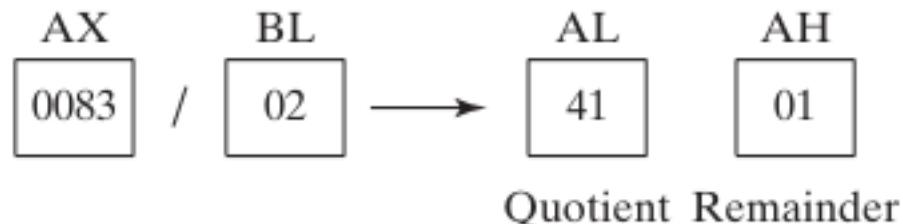
The following instructions perform 8-bit unsigned division (83h/2), producing a quotient of 41h and a remainder of 1:

```
746 ;dividend  
747 mov ax, 0083h  
748 ;divisor  
749 mov bl, 2  
750 ;divide  
751 div bl  
752 ;AL = 41h, AH = 01h
```

In this example, the dividend is stored in the AX register, and the divisor is stored in the BL register.

The DIV instruction divides the dividend by the divisor and stores the quotient in the AL register and the remainder in the AH register.

The following diagram illustrates the movement between registers:



### DIV Example 2:

The following instructions perform 32-bit unsigned division using a memory operand as the divisor:

```
769 .data
770     dividend QWORD 0000000800300020h
771     divisor DWORD 00000100h
772 .code
773     mov edx, DWORD PTR dividend + 4      ;high doubleword
774     mov eax, DWORD PTR dividend          ;low doubleword
775     div divisor
776     ;EAX = 08003000h, EDX = 00000020h
```

Explanation:

The **.data** directive defines two variables: dividend and divisor. The dividend variable is a 64-bit integer (QWORD) that contains the dividend.

The **divisor** variable is a 32-bit integer (DWORD) that contains the divisor. The .code directive marks the beginning of the code section.

The **mov edx, DWORD PTR dividend + 4** instruction loads the high doubleword of the dividend into the EDX register.

The **mov eax, DWORD PTR dividend** instruction loads the low doubleword of the dividend into the EAX register.

The **div divisor** instruction divides the dividend in the EAX:EDX registers by the divisor in the divisor variable and stores the quotient in the EAX register and the remainder in the EDX register.

After the DIV instruction executes, the EAX register will contain the quotient (08003000h) and the EDX register will contain the remainder (00000020h).

In other words, the above code performs the following operation:

$$\text{EAX:EDX} = 0000000800300020h / 00000100h$$

The result is stored in the EAX:EDX registers, with the quotient in EAX and the remainder in EDX.

The **EAX:EDX registers** are a pair of 32-bit registers that can be used to store a 64-bit value. The EAX register stores the lower 32 bits of the value, and the EDX register stores the higher 32 bits of the value.

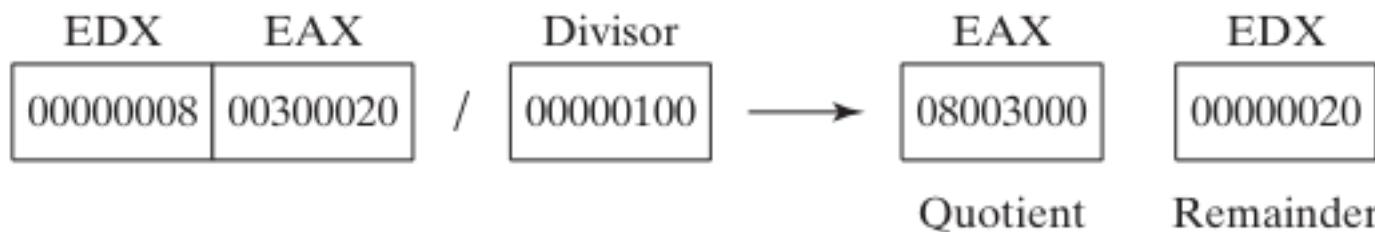
When you say that the result of a division operation is stored in the EAX:EDX registers, it means that the quotient of the division is stored in the EAX register and the remainder of the division is

stored in the EDX register.

For example, if you divide the number 100 by the number 10, the quotient is 10 and the remainder is 0. The EAX register would contain the value 10 and the EDX register would contain the value 0.

Another way to think about it is that the EAX:EDX registers can be used to store a 64-bit integer. When you perform a division operation, the result is a 64-bit integer, which is then stored in the EAX:EDX registers.

The following diagram illustrates the movement between registers:



This image is related to the text you provided. The image shows a diagram of a sequence of numbers, with the following arrows and labels:

This diagram illustrates the **32-bit unsigned division operation** that is described in the text.

The dividend is 00300020h, the divisor is 00000100h, the quotient is 08003000h, and the remainder is 00000020h.

The EAX register contains the low doubleword of the dividend and the EDX register contains the high doubleword of the dividend.

The DIV instruction divides the dividend in the **EAX:EDX registers** by the divisor in the divisor variable. The quotient is stored in the EAX register and the remainder is stored in the EDX register.

**EAX:EDX = 000000800300020h / 00000100h**

This equation represents the division operation that is being performed. The dividend is 000000800300020h and the divisor is 00000100h. The result of the division is stored in the EAX:EDX registers.

### *DIV Example 3:*

The following 64-bit division produces the quotient (010800000003330h) in RAX and the remainder (000000000000020h) in RDX:

```
787 .data
788     dividend_hi QWORD 000000000000108h
789     dividend_lo QWORD 0000000033300020h
790     divisor QWORD 000000000010000h
791 .code
792     mov rdx, dividend_hi
793     mov rax, dividend_lo
794     div divisor ;RAX = 010800000003330
795     ;RDX = 000000000000020
```

*Explanation:*

The .data directive defines three variables: dividend\_hi, dividend\_lo, and divisor.

The dividend\_hi and dividend\_lo variables contain the high and low doublewords of the dividend, respectively.

The divisor variable contains the divisor. The .code directive marks the beginning of the code section.

The mov rdx, dividend\_hi instruction loads the high doubleword of the dividend into the RDX register.

The mov rax, dividend\_lo instruction loads the low doubleword of the dividend into the RAX register.

The div divisor instruction divides the dividend in the RAX:RDX registers by the divisor in the divisor variable and stores the quotient in the RAX register and the remainder in the RDX register.

After the DIV instruction executes, the RAX register will contain the quotient (0108000000003330h) and the RDX register will contain the remainder (0000000000000020h).

***Why is each hexadecimal digit in the dividend shifted 4 positions to the right?***

This is because the dividend is being divided by 64. In other words, the dividend is being shifted 6 bits to the right.

Each hexadecimal digit represents 4 bits, so each hexadecimal digit in the dividend will be shifted 4 positions to the right.

For example, the high doubleword of the dividend (000000000000108h) is shifted 4 positions to the right to produce the following result:

```
000000000000108h >> 4 = 000000000000010h
```

The low doubleword of the dividend (000000033300020h) is also shifted 4 positions to the right to produce the following result:

```
000000033300020h >> 4 = 000000000003330h
```

The quotient of the division operation is then stored in the RAX register and the remainder is stored in the RDX register.

## *Signed DIV Instructions*

Signed integer division in MASM is similar to unsigned integer division, with one key difference: the dividend must be sign-extended before the division takes place. This is because the IDIV instruction (signed integer division) treats the dividend as a signed integer, and the result of the division is also a signed integer.

To sign-extend a number means to copy the sign bit of the number to all of the higher bits of the number.

This can be done using the:

- **CWD instruction (convert word to doubleword)**

- CBW instruction (convert byte to word).

The following MASM code shows how to sign-extend a 16-bit integer and then perform signed integer division:

```
799 .data
800     wordVal SWORD -101      ;009Bh
801 .code
802     mov eax, 0              ;EAX = 00000000h
803     mov ax, wordVal         ;EAX = 0000009Bh (+155)
804     cwd                      ;EAX = FFFFFFF9Bh (-101)
805     mov bx, 2                ;EBX is the divisor
806     idiv bx                 ;divide EAX by BX
```

In this code, the CWD instruction is used to sign-extend the AX register into the EAX register.

This ensures that the EAX register contains the correct signed value of -101 before the division operation is performed.

The IDIV instruction then divides the EAX register by the EBX register and stores the result in the EAX register.

The following table shows the results of the division operation:

Dividend	Divisor	Quotient	Remainder
-101	2	-50	1

The quotient of the division operation is -50 and the remainder is 1.

It is important to note that the **IDIV instruction** can also be used to perform unsigned integer division.

However, in this case, the dividend does not need to be sign-extended.

=====

### ***Sign Extension Instructions (CBW, CWD, CDQ)***

=====

The CBW, CWD, and CDQ instructions are sign extension instructions that are used to extend the sign bit of a smaller integer to a larger integer.

#### ***CBW***

The CBW instruction (convert byte to word) extends the sign bit of the AL register into the AH register.

This means that if the AL register contains a negative byte value, the AH register will be set to FFh. Otherwise, the AH register will be set to 00h.

The following MASM code shows how to use the CBW instruction:

```
810 .data  
811     byteVal SBYTE -101  
812     ;9Bh  
813 .code  
814     mov al, byteVal      ;AL = 9Bh  
815     cbw                  ;AX = FF9Bh
```

In this code, the CBW instruction is used to extend the sign bit of the AL register into the AH register. After the CBW instruction is executed, the AX register will contain the value FF9Bh, which is the signed representation of the number -101.

---

## CWD

The CWD instruction (convert word to doubleword) extends the sign bit of the AX register into the DX register.

This means that if the AX register contains a negative word value, the DX register will be set to FFh. Otherwise, the DX register will be set to 00h.

The following MASM code shows how to use the CWD instruction:

```
819 .data  
820     wordVal SWORD -101  
821 ; FF9Bh  
822 .code  
823     mov ax, wordVal ; AX = FF9Bh  
824     cwd ; DX:AX = FFFFFFF9Bh
```

In this code, the CWD instruction is used to extend the sign bit of the AX register into the DX register. After the CWD instruction is executed, the DX:AX registers will contain the value FFFFFFF9Bh, which is the signed representation of the number -101.

---

## CDQ

The CDQ instruction (convert doubleword to quadword) extends the sign bit of the EAX register into the EDX register.

This means that if the EAX register contains a negative doubleword value, the EDX register will be set to FFh. Otherwise, the EDX register will be set to 00h.

The following MASM code shows how to use the CDQ instruction:

```
827 .data  
828     dwordVal SDWORD -101  
829     ;FFFFF9Bh  
830 .code  
831     mov eax, dwordVal ;EAX = FFFFF9Bh  
832     cdq                 ;EDX:EAX = FFFFFFFFFFFFF9Bh
```

---

In this code, the CDQ instruction is used to extend the sign bit of the EAX register into the EDX register. After the CDQ instruction is executed, the EDX:EAX registers will contain the value FFFFFFFFFFFFF9Bh, which is the signed representation of the number -101.

### ***When to use sign extension instructions***

Sign extension instructions are typically used in the following situations:

When performing signed integer arithmetic operations. When converting a signed integer from a smaller type to a larger type.

When passing a signed integer to a function that expects a signed integer parameter.

For example, if you are writing a function that calculates the average of two signed integers, you would need to use a sign extension instruction to ensure that the two integers are converted to the same type before the division operation is performed.

### ***Conclusion***

Sign extension instructions are a powerful tool that can be used to ensure that signed integers are handled correctly. By understanding how to use these instructions, you can write more efficient and reliable code.

## ***IDIV Instruction***

The **IDIV (signed divide) instruction** performs signed integer division, using the same operands as DIV.

However, before executing 8-bit division, the dividend (AX) must be completely sign-extended. The remainder always has the same sign as the dividend.

### **Syntax:**

```
840 IDIV reg/mem8  
841 IDIV reg/mem16  
842 IDIV reg/mem32
```

### **Operands:**

**reg/mem8:** An 8-bit register or memory location containing the divisor.

**reg/mem16:** A 16-bit register or memory location containing the divisor.

**reg/mem32:** A 32-bit register or memory location containing the divisor.

### ***Operation:***

The IDIV instruction divides the signed integer dividend in the AX register by the signed integer divisor in the operand.

The quotient is stored in the AL register and the remainder is stored in the AH register.

### **Example 1:**

The following instructions divide -48 by 5. After IDIV executes, the quotient in AL is -9 and the remainder in AH is -3:

```
847 .data
848     byteVal SBYTE -48
849     ;D0 hexadecimal
850 .code
851     mov al,byteVal
852     ;lower half of dividend
853     cbw
854     ;extend AL into AH
855     mov bl,+5
856     ;divisor
857     idiv bl
858     ;AL = -9, AH = -3
```

### ***Explanation:***

The CBW instruction sign-extends the AL register into the AX register.

This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the AX register by the BL register and stores the quotient in the AL register and the remainder in the AH register.

### ***Example 2:***

The following instructions divide -5000 by 256:

```
863 .data
864     wordVal SWORD -5000
865 .code
866     mov ax,wordVal
867     ;dividend, low
868     cwd
869     ;extend AX into DX
870     mov bx,+256
871     ;divisor
872     idiv bx
873     ;quotient AX = -19, rem DX = -136
```

### *Explanation:*

The CWD instruction sign-extends the AX register into the DX register. This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the DX:AX registers by the BX register and stores the quotient in the AX register and the remainder in the DX register.

### Example 3:

The following instructions divide 50,000 by -256:

```
878 .data
879     dwordVal SDWORD + 50000
880 .code
881     mov eax,dwordVal
882     ;dividend, low
883     cdq
884     ;extend EAX into EDX
885     mov ebx,-256
886     ;divisor
887     idiv ebx
888     ;quotient EAX = -195, rem EDX = +80
```

### *Explanation:*

The CDQ instruction sign-extends the EAX register into the EDX register.

This is necessary because the IDIV instruction divides signed integers.

The IDIV instruction then divides the EDX:EAX registers by the EBX register and stores the quotient in the EAX register and the remainder in the EDX register.

Important:

The IDIV instruction undefines all arithmetic status flag values.

The IDIV instruction can also be used to perform unsigned integer division.

However, in this case, the dividend does not need to be sign-extended.

=====

### *Divide Overflow*

=====

A divide overflow condition occurs when the result of a division operation is too large to fit into the destination operand. This causes a processor exception and halts the current program.

The following instructions generate a divide overflow because the quotient (100h) is too large for the 8-bit AL destination register:

```
892 mov ax,1000h  
893 mov bl,10h  
894 div bl  
895 ; AL cannot hold 100h
```

### *Avoiding Divide Overflow:*

**Use a larger destination operand.** For example, instead of using the AL register, you could use the AX register or the EAX register.

**Use a smaller divisor.** For example, instead of dividing by 10h, you could divide by 2h. Use a combination of the above two approaches. For example, you could use the AX register as the destination operand and divide by 2h. Test the divisor before dividing to avoid division by zero.

The following code uses a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition:

```
901 mov eax,1000h  
902 cdq  
903 mov ebx,10h  
904 div ebx  
905 ; EAX = 00000100h
```

### *Explanation:*

The CDQ instruction sign-extends the EAX register into the EDX register. This creates a 64-bit

dividend in the EDX:EAX registers. The DIV instruction then divides the EDX:EAX registers by the EBX register and stores the quotient in the EAX register.

---

The following code uses a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition and tests the divisor before dividing to avoid division by zero:

```
909 mov eax, dividend
910 mov bl, divisor
911 cmp bl, 0
912 je NoDivideZero
913
914 ; Not zero: continue
915 div bl
916
917 ; ...
918
919 NoDivideZero:
920 ; Display "Attempt to divide by zero"
```

#### *Explanation:*

The **MOV** instructions load the dividend and divisor into the EAX and BL registers, respectively. The CMP instruction compares the BL register to zero. If the BL register is equal to zero, the JE instruction jumps to the NoDivideZero label.

If the **BL** register is not equal to zero, the DIV instruction divides the EAX register by the BL register and stores the quotient in the EAX register.

The **NoDivideZero** label is where the code will jump if the divisor is zero. At this point, the code could display an error message or take other appropriate action.

=====

### ***Implementing Arithmetic Expressions(ASM)***

=====

To implement arithmetic expressions in assembly language, we need to break them down into their constituent operations. For example, the following C++ statement:

```
var4 = (var1 + var2) * var3;
```

can be broken down into the following assembly language instructions:

```
928 mov eax, var1  
929 add eax, var2  
930 mul var3  
931 mov var4, eax
```

The first instruction loads the value of var1 into the EAX register. The second instruction adds the value of var2 to the EAX register.

value of var2 to the EAX register.

The third instruction multiplies the value of var3 by the value in the EAX register and stores the result in the EAX register.

The fourth instruction stores the value in the EAX register into the var4 variable.

### Handling Overflow

When performing arithmetic operations in assembly language, it is important to be aware of the possibility of overflow.

Overflow occurs when the result of an operation is too large to fit into the destination operand.

For example, the following assembly language instruction:

```
mul var3
```

will multiply the value in the EAX register by the value of var3 and store the result in the EAX register. If the product of the multiplication is too large to fit into the EAX register, overflow will occur.

To handle overflow, we can use the JC (jump on carry) instruction. The JC instruction will jump to a specified label if the carry flag is set.

The carry flag is set if there was an overflow when performing the previous arithmetic operation.

The following assembly language code shows how to handle overflow when multiplying two unsigned 32-bit integers:

```
945 mov eax, var1
946 add eax, var2
947 mul var3
948 jc tooBig      ;jump if overflow
949 mov var4, eax
950 jmp next
951 tooBig:
952     ;display error message
```

If the MUL instruction generates a product larger than 32 bits, the JC instruction will jump to the tooBig label. The tooBig label can then display an error message or take other appropriate action.

### Handling Signed Integers

When performing arithmetic operations on signed integers, it is important to be aware of the possibility of sign extension.

Sign extension is the process of copying the sign bit of an integer to all of the higher bits of the integer.

For example, the following assembly language instruction:

**idiv var3**

will divide the value in the EDX:EAX registers by the value of var3 and store the quotient in the EAX register and the remainder in the EDX register.

If the dividend is a signed integer, it is important to sign-extend the dividend into EDX before performing the division.

The following assembly language code shows how to divide two signed 32-bit integers:

```
960 mov eax, var2
961 neg eax
962 cdq          ;sign-extend dividend
963 idiv var3
964 mov ebx, edx ;EBX = right side
```

The CDQ instruction sign-extends the EAX register into the EDX register. This ensures that the EDX:EAX registers contain the correct signed value of the dividend before the division operation is performed.

### **Questions:**

**Explain why overflow cannot occur when the MUL and one-operand IMUL instructions execute.**

Overflow cannot occur because these instructions ensure that the destination operand is twice the size of the multiplicand and multiplier. This means there is always enough space to hold the result

without overflowing.

**How is the one-operand IMUL instruction different from MUL in the way it generates a multiplication product?**

The one-operand IMUL instruction, unlike MUL, can perform signed integer multiplication. It generates a product that can be positive or negative, depending on the signs of the multiplicand and multiplier.

**What has to happen for the one-operand IMUL to set the Carry and Overflow flags?**

The Carry and Overflow flags are set when the product of one-operand IMUL is too large to fit into the destination operand size, signifying an overflow condition. This occurs when the result is outside the representable range for the given operand size.

**When EBX is the operand in a DIV instruction, which register holds the quotient?**

When EBX is the operand in a DIV instruction, the EAX register holds the quotient.

**When BX is the operand in a DIV instruction, which register holds the quotient?**

When BX is the operand in a DIV instruction, the AX register holds the quotient.

**When BL is the operand in a MUL instruction, which registers hold the product?**

When BL is the operand in a MUL instruction, the AX and DX registers hold the product. AX contains the low 16 bits, and DX contains the high 16 bits of the 32-bit product.

**Show an example of sign extension before calling the IDIV instruction with a 16-bit operand.**

Sign extension is necessary when working with signed integers. Here's an example of sign extension before calling IDIV with a 16-bit operand:

```
970 movsx eax, word ptr [your_16_bit_variable] ; Sign extend 16-bit value to 32 bits  
971 idiv ebx ; Perform signed division with the extended value
```

This code first sign-extends the 16-bit value to a 32-bit value in the EAX register before performing a signed division with the IDIV instruction.

## ***Extended Addition and Subtraction***

The **ADC (add with carry) instruction** in assembly language is used to add two operands, taking into account the Carry flag.

The Carry flag is set when the result of a previous addition or subtraction operation overflows.

The ADC instruction is typically used to perform multi-byte or multi-word addition and subtraction operations.

```
0976 ; Load the first operand into the AL register  
0977 mov al, 0FFh  
0978  
0979 ; Add the second operand to the AL register, setting the Carry flag if necessary  
0980 add al, 0FFh  
0981  
0982 ; Save the result in the DL register, adding the Carry flag if necessary  
0983 adc dl, 0
```

After the first instruction, the AL register contains the value FEh and the Carry flag is set.

The second instruction then adds the Carry flag to the DL register, resulting in a final value of 01FEh in the DL:AL register pair.

Here is another example of using the ADC instruction:

```
0988 ; Load the first operand into the EAX register  
0989 mov eax, 0xFFFFFFFFh  
0990  
0991 ; Add the second operand to the EAX register, setting the Carry flag if necessary  
0992 add eax, 0xFFFFFFFFh  
0993  
0994 ; Save the result in the EDX register, adding the Carry flag if necessary  
0995 adc edx, 0
```

After the first instruction, the EAX register contains the value FFFFFFFFh and the Carry flag is set.

The second instruction then adds the Carry flag to the EDX register, resulting in a final value of 00000001FFFFFEh in the EDX:EAX register pair.

The ADC instruction can be used to add operands of any size, including 1024-bit integers.

To do this, multiple ADC instructions would be used in sequence, carrying the Carry flag from one instruction to the next.

Here is an example of how to add two 1024-bit integers using the ADC instruction:

```
1000 ; Load the first operand into the [EAX:EBX:ECX:EDX] register quad
1001 mov eax, [operand1]
1002 mov ebx, [operand1 + 4]
1003 mov ecx, [operand1 + 8]
1004 mov edx, [operand1 + 12]
1005
1006 ; Load the second operand into the [ESI:EDI:ESI:EDI] register quad
1007 mov esi, [operand2]
1008 mov edi, [operand2 + 4]
1009
1010 ; Add the two operands, carrying the Carry flag from each instruction to the next
1011 adc eax, esi
1012 adc ebx, edi
1013 adc ecx, edi
1014 adc edx, esi
1015
1016 ; Save the result in the [EAX:EBX:ECX:EDX] register quad
1017 mov [result], eax
1018 mov [result + 4], ebx
1019 mov [result + 8], ecx
1020 mov [result + 12], edx
```

This code will add the two 1024-bit operands stored in the `operand1` and `operand2` arrays and store the result in the `result` array. Let's break down the code step by step:

*Loading the First Operand (`operand1`):*

The code begins by loading the first operand, which is a 1024-bit value, into the [EAX:EBX:ECX:EDX] register quad. It does this in four 32-bit chunks ( $4 * 32 = 128$  bits):

**mov eax, [operand1]**: Loads the first 32 bits of operand1 into the EAX register.

**mov ebx, [operand1 + 4]**: Loads the next 32 bits (bits 32-63) of operand1 into the EBX register.

**mov ecx, [operand1 + 8]**: Loads the following 32 bits (bits 64-95) of operand1 into the ECX register.

**mov edx, [operand1 + 12]**: Loads the last 32 bits (bits 96-127) of operand1 into the EDX register.

Loading the Second Operand (operand2):

The code then loads the second operand, which is also a 1024-bit value, into the [ESI:EDI:ESI:EDI] register quad. Like the first operand, it does this in four 32-bit chunks:

**mov esi, [operand2]**: Loads the first 32 bits of operand2 into the ESI register.

**mov edi, [operand2 + 4]**: Loads the next 32 bits (bits 32-63) of operand2 into the EDI register.

### ***Adding the Two Operands with Carry Propagation:***

The actual addition of the two operands is performed in this step. It uses the adc (add with carry) instruction, which allows for carry propagation.

**adc eax, esi**: Adds the first 32 bits of the first operand (EAX) and the first 32 bits of the second operand (ESI) along with any carry from the previous addition. The result is stored in EAX.

**adc ebx, edi**: Adds the next 32 bits of the first operand (EBX) and the next 32 bits of the second operand (EDI) along with any carry from the previous addition. The result is stored in EBX.

operand (EDI) along with any carry from the previous addition. The result is stored in EBX.

**adc ecx, edi:** Adds the following 32 bits of the first operand (ECX) and the next 32 bits of the second operand (EDI) along with any carry from the previous addition. The result is stored in ECX.

**adc edx, esi:** Adds the last 32 bits of the first operand (EDX) and the first 32 bits of the second operand (ESI) along with any carry from the previous addition. The result is stored in EDX.

### ***Storing the Result in the result Array:***

Finally, the result of the addition is saved back into the result array in four 32-bit chunks.

**mov [result], eax:** Stores the first 32 bits of the result (in EAX) in the result array.

**mov [result + 4], ebx:** Stores the next 32 bits (in EBX) of the result in the result array.

**mov [result + 8], ecx:** Stores the following 32 bits (in ECX) of the result in the result array.

**mov [result + 12], edx:** Stores the last 32 bits (in EDX) of the result in the result array.

This code essentially performs a multi-precision addition for 1024-bit operands, taking care of carry propagation between 32-bit chunks. It's a low-level operation that handles large integers by breaking them into manageable pieces.

The ADC instruction is a powerful tool for performing multi-byte and multi-word addition and subtraction operations. It can be used to add and subtract operands of any size, including very large integers.

=====

The Extended\_Add procedure below is an example of how to add two extended integers of the same size using assembly language.

It works by iterating through the two integers, adding each corresponding byte and carrying over any carry from the previous iteration. The procedure takes four arguments:

**ESI** and **EDI**: Pointers to the two integers to be added.

**EBX**: A pointer to a buffer in which the sum will be stored. The buffer must be one byte longer than the two integers.

**ECX**: The length of the longest integer in bytes. The procedure assumes that the integers are stored in little-endian order, with the least significant byte at the lowest offset.

Here is a more detailed explanation of the code:

```
1025 ;-----  
1026 ;Extended_Add PROC  
1027 ;-----  
1028 pushad ; Save all registers on the stack.  
1029 clc ; Clear the Carry flag.  
1030  
1031 L1: mov al,[esi] ; Get the next byte from the first integer.  
1032 ; Add the next byte from the second integer, including any carry from the previous iteration.  
1033 adc al,[edi]  
1034 pushfd ; Save the Carry flag.  
1035 mov [ebx],al ; Store the partial sum.  
1036 add esi,1 ; Advance the pointers to the next bytes in the integers.  
1037 add edi,1  
1038 add ebx,1  
1039 popfd ; Restore the Carry flag.  
1040 loop L1 ; Repeat the loop until all bytes have been added.  
1041  
1042 ; Clear the high byte of the sum, since it may contain a carry.  
1043 mov byte ptr [ebx],0  
1044 adc byte ptr [ebx],0 ; Add any leftover carry.  
1045  
1046 popad ; Restore all registers from the stack.  
1047 ret ; Return from the procedure.  
1048 Extended_Add ENDP
```

The loop at L1 iterates through the two integers, adding each corresponding byte and carrying over any carry from the previous iteration.

The Carry flag is saved and restored on each iteration so that it is always in the correct state

when the next addition is performed.

After the loop has finished iterating, the high byte of the sum is cleared.

This is necessary because the high byte may contain a carry from the addition of the two highest bytes of the integers.

The ADC instruction is then used to add any leftover carry to the high byte of the sum.

Finally, the registers are restored from the stack and the procedure returns.

The Extended\_Add procedure is a useful example of how to perform extended precision arithmetic in assembly language.

It can be used to add two integers of any size, regardless of whether they fit within the registers of the CPU.

-----

The following sample code calls the Extended\_Add procedure to add two 8-byte integers:

```
1053 .data
1054     op1 BYTE 34h, 12h, 98h, 74h, 06h, 0A4h, 0B2h, 0A2h
1055     op2 BYTE 02h, 45h, 23h, 00h, 00h, 87h, 10h, 80h
1056     sum BYTE 9 dup(0)
1057
1058 .code
1059 main PROC
1060     ; Load addresses of operands and result
1061     mov esi, OFFSET op1    ; First operand
1062     mov edi, OFFSET op2    ; Second operand
1063     mov ebx, OFFSET sum    ; Result operand
1064
1065     ; Determine the number of bytes to process
1066     mov ecx, LENGTHOF op1
1067
1068     ; Call the Extended_Add function
1069     call Extended_Add
1070
1071     ; Display the sum.
1072     mov esi, OFFSET sum
1073     mov ecx, LENGTHOF sum
1074     call Display_Sum
1075
1076     ; Call a function to output a newline.
1077     call Crlf
1078
1079     ; Exit the program
1080     invoke ExitProcess, 0
1081
1082 main ENDP
```

The .data section of the code defines three byte arrays: op1, op2, and sum. The op1 and op2 arrays store the two integers to be added, and the sum array will store the result of the addition.

The sum array is one byte longer than the other two arrays to accommodate any carry that may be generated.

The .code section of the code contains the main procedure.

The main procedure first moves the pointers to the two operand arrays and the sum array into the ESI, EDI, and EBX registers, respectively. It then moves the length of the operands into the ECX register.

Next, the main procedure calls the Extended\_Add procedure.

The Extended\_Add procedure will add the two operands and store the result in the sum array.

After the Extended\_Add procedure has finished executing, the main procedure moves the pointer to the sum array into the ESI register and the length of the sum array into the ECX register.

It then calls the Display\_Sum procedure to display the sum to the console.

The Display\_Sum procedure is a simple procedure that iterates through the sum array and prints each byte to the console.

After the Display\_Sum procedure has finished executing, the main procedure calls the Crlf procedure to print a newline character to the console.

The output of the program is the following:

1087

0122C32B0674BB5736

This is the correct sum of the two operands, even though the addition produced a carry. The Extended\_Add procedure handles the carry correctly and stores the correct result in the sum array.

---

The Display\_Sum procedure below is related to the Extended\_Add procedure you provided in the previous question.

The Display\_Sum procedure is used to display the sum of two integers that have been added using the Extended\_Add procedure.

It works by iterating through the sum array in reverse order, starting with the high-order byte and working its way down to the low-order byte.

For each byte in the sum array, the Display\_Sum procedure calls the WriteHexB procedure to display the byte in hexadecimal format.

Here is a more detailed explanation of the Display\_Sum procedure:

```
1092 Display_Sum PROC
1093     pushad          ; Save registers
1094     ; Point to the last array element
1095     add esi, ecx
1096     sub esi, TYPE BYTE
1097     mov ebx, TYPE BYTE
1098
1099 L1:
1100     mov al, [esi]    ; Get a byte from the array
1101     call WriteHexB ; Display it in hexadecimal
1102     sub esi, TYPE BYTE ; Move to the previous byte
1103     loop L1
1104
1105     popad          ; Restore registers
1106     ret
1107 Display_Sum ENDP
```

The `pushad` instruction saves all of the registers on the stack. The `add esi,ecx` instruction moves the value of the ECX register into the ESI register.

The `sub esi,TYPE BYTE` instruction subtracts the size of a byte from the ESI register. This moves the pointer to the last element of the sum array.

The `mov ebx,TYPE BYTE` instruction moves the size of a byte into the EBX register. This will be used to loop through the sum array.

The L1: label marks the beginning of the loop.

The mov al,[esi] instruction moves the byte at the current position in the sum array into the AL register. The call WriteHexB instruction calls the WriteHexB procedure to display the byte in hexadecimal format.

The sub esi,TYPE BYTE instruction subtracts the size of a byte from the ESI register. This moves the pointer to the previous byte in the sum array.

The loop L1 instruction loops back to the beginning of the loop if the EBX register is not zero.

The popad instruction restores all of the registers from the stack. The ret instruction returns from the procedure.

The Display\_Sum procedure is a good example of how to iterate through an array in reverse order. It is also a good example of how to call another procedure from within a procedure.

=====

### ***Subtract with Borrow***

=====

The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.

The possible operands are the same as for the ADC instruction, which means it can be used to

subtract operands of any size, including 32-bit, 64-bit, and even 128-bit operands.

The following example code carries out 64-bit subtraction with 32-bit operands:

```
1111 mov edx, 7  
1112 ; upper half  
1113 mov eax, 1  
1114 ; lower half  
1115 sub eax, 2  
1116 ; subtract 2  
1117 sbb edx, 0  
1118 ; subtract upper half
```

This code will subtract the value 2 from the 64-bit integer stored in the EDX:EAX register pair. The subtraction is done in two steps:

The value 2 is subtracted from the lower 32 bits of the integer, which are stored in the EAX register. This subtraction may set the Carry flag if a borrow is required.

The SBB instruction subtracts both 0 and the value of the Carry flag from the upper 32 bits of the integer, which are stored in the EDX register.

Here is a more detailed explanation of the code:

```
1123 mov edx, 7  
1124 ; upper half
```

This instruction moves the value 7 into the EDX register. This is the upper 32 bits of the 64-bit integer that we will be subtracting from.

```
1128 mov eax, 1  
1129 ; lower half
```

This instruction moves the value 1 into the EAX register. This is the lower 32 bits of the 64-bit integer that we will be subtracting from.

```
01 sub eax, 2  
02 ; subtract 2
```

This instruction subtracts the value 2 from the lower 32 bits of the integer, which are stored in the EAX register. This may set the Carry flag if a borrow is required.

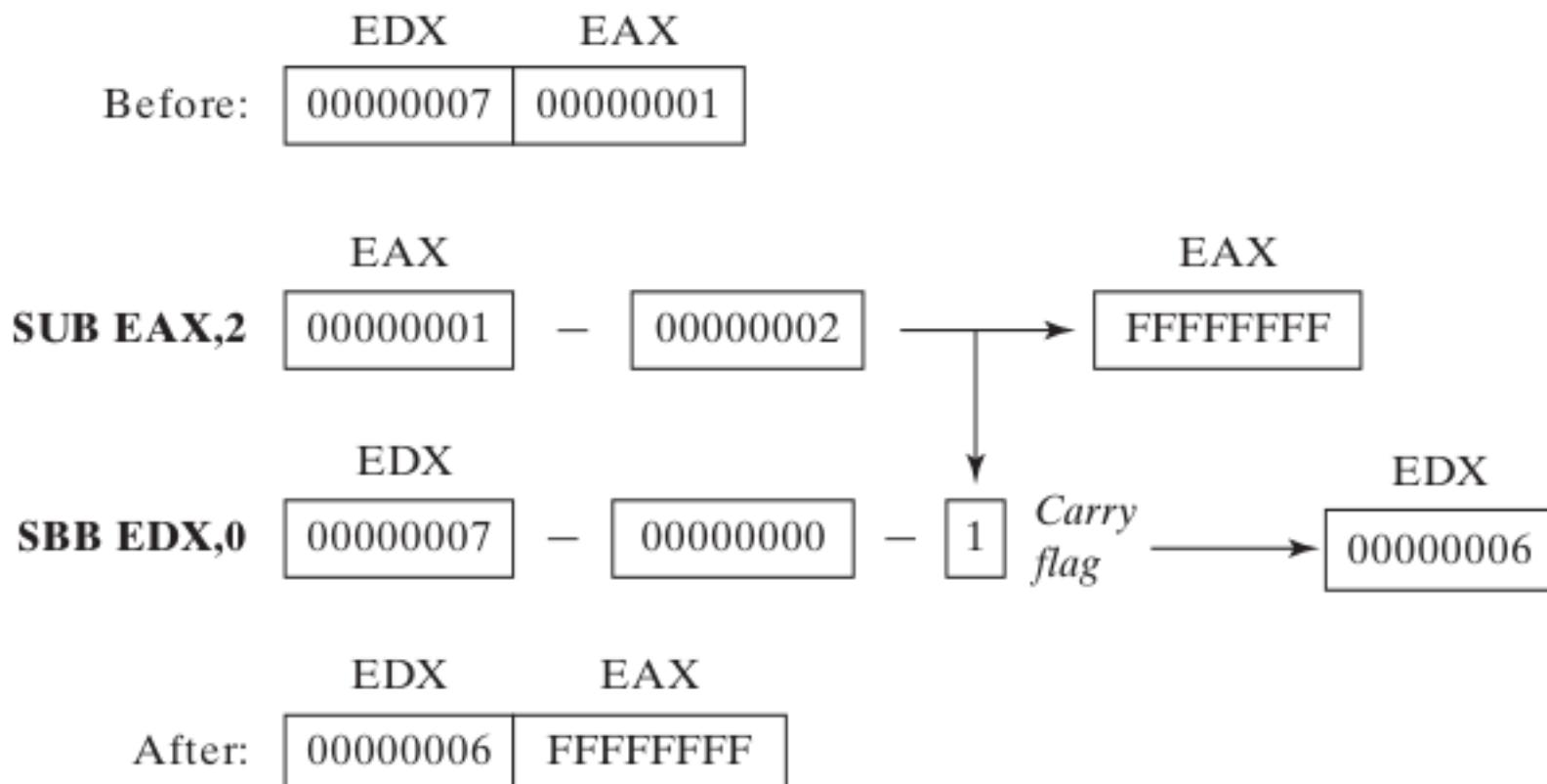
```
10 sbb edx, 0  
11 ; subtract upper half
```

This instruction subtracts both 0 and the value of the Carry flag from the upper 32 bits of the integer, which are stored in the EDX register.

After the code has executed, the EDX:EAX register pair will contain the result of the subtraction, which is the value 0000000700000001h.

The SBB instruction is a powerful tool for performing multi-byte and multi-word subtraction operations. It can be used to subtract operands of any size, including very large integers.

FIGURE 7–2 Subtracting from a 64-bit integer using SBB.



#### Describe the ADC instruction:

The ADC (Add with Carry) instruction is used for addition in assembly language. It adds two operands, along with the value of the Carry flag, and stores the result in the destination operand. If there is a carry from the addition, it sets the Carry flag; otherwise, it clears it. It is particularly useful for multi-precision arithmetic, where you need to handle carry from previous

operations.

#### **Describe the SBB instruction:**

The SBB (Subtract with Borrow) instruction is used for subtraction in assembly language. It subtracts the source operand from the destination operand, along with the Borrow flag (Carry flag treated as borrow), and stores the result in the destination operand. If a borrow is generated from the subtraction, it sets the Carry flag; otherwise, it clears it. SBB is often used for multi-precision arithmetic to handle borrows from previous operations.

#### **Values of EDX:EAX after the given instructions execute:**

`mov edx, 10h` loads 16 into EDX.

`mov eax, 0A0000000h` loads A0000000h into EAX. `add eax, 20000000h` adds 20000000h to EAX without carry. `adc edx, 0` adds 0 to EDX along with any carry. Result: EDX = 0 (no carry), EAX = C0000000h.

#### **Values of EDX:EAX after the given instructions execute:**

`mov edx, 100h` loads 256 into EDX.

`mov eax, 80000000h` loads 80000000h into EAX.

`sub eax, 90000000h` subtracts 90000000h from EAX without borrow. `sbb edx, 0` subtracts 0 from EDX along with any borrow.

Result: EDX = FFFFFFFF (due to borrow), EAX = FFFFFFFF.

#### **Contents of DX after the given instructions execute:**

`mov dx, 5` loads 5 into DX.

`stc` sets the Carry flag to 1.

`mov ax, 10h` loads 16 into AX.

adc dx, ax adds AX to DX along with the carry. Result: DX = 1 (due to carry).

## ***ASCII and unpacked decimal arithmetic***

This is a type of arithmetic that can be performed on ASCII decimal strings, without requiring them to be converted to binary.

There are two advantages to using ASCII arithmetic:

Conversion from string format before performing arithmetic is not necessary.

Using an assumed decimal point permits operations on real numbers without danger of the roundoff errors that occur with floating-point numbers.

However, ASCII arithmetic does execute more slowly than binary arithmetic.

There are four instructions that deal with ASCII addition, subtraction, multiplication, and division:

- AAA (ASCII adjust after addition)

- AAS (ASCII adjust after subtraction)
- AAM (ASCII adjust after multiplication)
- AAD (ASCII adjust before division)

These instructions are used to adjust the sum, difference, product, or quotient, respectively, to ensure that it is in a valid ASCII decimal format.

Here is an example of how to use ASCII addition to add the numbers 3402 and 1256:

```
19 ; Load the first operand into the AL register.  
20 mov al, 3  
21 ; Load the second operand into the AH register.  
22 mov ah, 4  
23 ; Add the two operands together.  
24 adc al, 0  
25 ; ASCII adjust the sum.  
26 aaa  
27 ; Store the sum in the AX register.  
28 mov ax, al
```

This code will add the two numbers together and store the sum in the AX register.

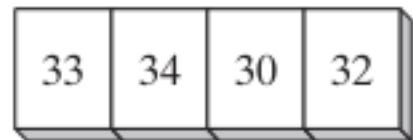
The AAA instruction is used to adjust the sum to ensure that it is in a valid ASCII decimal format.

ASCII subtraction can be performed in a similar way, using the AAS instruction to adjust the difference.

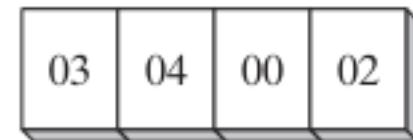
ASCII multiplication and division are also possible, but they are more complex and require the use of the AAM and AAD instructions, respectively.

ASCII arithmetic can be a useful tool for performing arithmetic on ASCII decimal strings. It is important to note, however, that ASCII arithmetic is slower than binary arithmetic.

ASCII format:



Unpacked:



(All values are in hexadecimal)

This means that the numbers in the block diagram are represented in two different formats: ASCII and unpacked decimal.

ASCII is a character encoding standard that assigns a unique code to each letter, number, and symbol. The ASCII codes for the digits 0 through 9 are 30 through 39, respectively.

Unpacked decimal is a binary representation of decimal numbers, where one byte is used to represent each digit. The unpacked decimal representation of the number 12 is 000000010010, or 0x0302 in hexadecimal.

The image shows that the four numbers in the block diagram are the same in both ASCII and unpacked decimal formats. This is because the ASCII codes for the digits 0 through 9 are the same as the unpacked decimal representations of those digits.

Digit	ASCII code	Unpacked decimal
0	30	00
1	31	01
2	32	02
3	33	03
4	34	04
5	35	05
6	36	06
7	37	07
8	38	08
9	39	09

Here is a table showing the ASCII codes and unpacked decimal representations of the four numbers in the image:

Number	ASCII code	Unpacked decimal
3	33	03
4	34	04
0	30	00
2	32	02

This image is useful for understanding the relationship between ASCII and unpacked decimal formats. It is also a reminder that all numbers are ultimately represented in binary form, regardless of how they are displayed or stored.

---

As you can see, the ASCII codes for the digits 0 through 9 are simply the decimal values of the digits shifted by 4 bits. This makes it easy to convert between ASCII and unpacked decimal representations of numbers.

For example, to convert the ASCII code 34 to an unpacked decimal representation, we simply shift the

ASCII code right by 4 bits. This gives us the unpacked decimal representation 04, which is the decimal value of 4.

To convert the unpacked decimal representation 05 to an ASCII code, we simply shift the unpacked decimal representation left by 4 bits. This gives us the ASCII code 35, which is the ASCII code for the digit 5.

The fact that the ASCII codes for the digits 0 through 9 are the same as their unpacked decimal representations makes it easy to perform arithmetic on ASCII decimal strings.

For example, we can add two ASCII decimal strings by simply adding the corresponding ASCII codes for each digit.

## ***AAA (ASCII adjust after addition)***

The ASCII addition procedure below is used to add ASCII decimal values with implied decimal points.

It works by iterating through the two operands, adding each corresponding digit and carrying over any carry from the previous iteration.

The procedure uses the AAA instruction to adjust the sum after each addition to ensure that it is in a valid ASCII decimal format.

Here is a more detailed explanation of the procedure:

```
36 mov esi, SIZEOF decimal_one - 1
37 mov edi, SIZEOF decimal_one
38 mov ecx, SIZEOF decimal_one
39 mov bh, 0
40
41 L1:
42 mov ah, 0
43 mov al, decimal_one[esi]
44 add al, bh
45 aaa
46 mov bh, ah
47 or bh, 30h
48 add al, decimal_two[esi]
49 aaa
50 or bh, ah
51 or bh, 30h
52 or al, 30h
53 mov sum[edi], al
54 dec esi
55 dec edi
56 loop L1
57
58 mov sum[edi], bh
```

The esi register points to the last digit position of the first operand, and the edi register points to the last digit position of the sum. The ecx register contains the length of the first operand.

The loop at L1 iterates through the two operands, adding each corresponding digit and carrying over any carry from the previous iteration.

The AAA instruction is used to adjust the sum after each addition to ensure that it is in a valid ASCII decimal format.

The carry digit is saved in the bh register and then converted to ASCII. The ASCII carry digit is then added to the sum.

After the loop has finished iterating, the last carry digit is saved in the sum.

The following example shows how to use the ASCII addition procedure to add the numbers **1001234567.89** **765** and **9004020765.02015**:

```
63 mov
64 esi,OFFSET decimal_one
65 mov
66 edi,OFFSET sum
67 mov
68 ecx,SIZEOF decimal_one
69
70 call
71 ASCII_add
72
73 mov
74 edx,OFFSET sum
75 call
76 WriteString
77 call
78 Crlf
```

This code will produce the following output:

**1000525533291780.**

As you can see, the ASCII addition procedure correctly adds the two numbers, even though they have implied decimal points.

## **AAS, AAM, AAD**

The AAS (ASCII adjust after subtraction) instruction is used to adjust the result of a subtraction operation when the result is negative.

It is typically used after a SUB or SBB instruction that has subtracted one unpacked decimal value from another and stored the result in the AL register.

The AAS instruction works by first checking the Carry flag. If the Carry flag is set, it means that the subtraction resulted in a negative number.

In this case, the AAS instruction subtracts 1 from the AH register and sets the Carry flag again.

It also sets the AL register to the ASCII representation of the negative number.

If the Carry flag is not set, it means that the subtraction resulted in a positive number.

In this case, the AAS instruction simply sets the AL register to the ASCII representation of the positive number.

Here is an example of how to use the AAS instruction:

```
83 mov ah, 0 ; clear AH before subtraction
84 mov al, 8
85 sub al, 9 ; subtract 9 from 8
86 aas ; adjust the result
87 or al, 30h ; convert AL to ASCII
```

After the above code has executed, the AL register will contain the ASCII representation of the number -1, which is 45h.

The AAS instruction can be useful for performing arithmetic operations on ASCII decimal strings.

For example, it can be used to subtract two ASCII decimal strings, even if they have different lengths.

Here is an example of how to use the AAS instruction to subtract two ASCII decimal strings:

```
095 mov esi, offset first_number
096 mov edi, offset second_number
097 mov ecx, length_of_first_number
098
099 loop:
100 mov ah, 0 ; clear AH before subtraction
101 mov al, [esi]
102 sub al, [edi]
103 aas ; adjust the result
104 or al, 30h ; convert AL to ASCII
105 mov [esi], al
106
107 inc esi ; increment the first number pointer
108 inc edi ; increment the second number pointer
109 dec ecx ; decrement the loop counter
110
111 cmp ecx, 0
112 jne loop ; continue looping if the loop counter is not zero
```

This code will subtract the two ASCII decimal strings starting at the least significant digits and working their way up to the most significant digits.

The AAS instruction is used to adjust the result of each subtraction operation to ensure that it is in a valid ASCII decimal format.

The AAS instruction is a powerful tool for performing arithmetic operations on ASCII decimal strings. It is easy to use and can be used to implement a variety of arithmetic algorithms.

=====

### ***AAM (ASCII adjust after multiplication)***

=====

The AAM (ASCII adjust after multiplication) instruction is used to convert the binary product produced by the MUL instruction to unpacked decimal format.

The MUL instruction must be used to multiply two unpacked decimal values.

The AAM instruction works by dividing the product by 100 and storing the quotient in the AH register and the remainder in the AL register.

The quotient represents the most significant digit of the unpacked decimal result, and the remainder represents the least significant digit.

Here is an example of how to use the AAM instruction:

```
116 mov bl, 5 ; first operand  
117 mov al, 6 ; second operand  
118 mul bl ; AX = 001Eh (binary product)  
119 aam ; AX = 0300h (unpacked decimal result)
```

After the above code has executed, the AX register will contain the unpacked decimal representation of the product of 5 and 6, which is 30.

The AAM instruction can be useful for performing arithmetic operations on ASCII decimal strings. For example, it can be used to multiply two ASCII decimal strings, even if they have different lengths.

Here is an example of how to use the AAM instruction to multiply two ASCII decimal strings:

```
124 mov esi, offset first_number
125 mov edi, offset second_number
126 mov ecx, length_of_first_number
127
128 loop:
129 mov bl, [esi]
130 mov al, [edi]
131 mul bl ; AX = binary product of two digits
132 aam ; AX = unpacked decimal representation of product
133
134 mov [esi], al ; store the least significant digit of the product
135 mov [edi], ah ; store the most significant digit of the product
136
137 inc esi ; increment the first number pointer
138 inc edi ; increment the second number pointer
139 dec ecx ; decrement the loop counter
140
141 cmp ecx, 0
142 jne loop ; continue looping if the loop counter is not zero
```

This code will multiply the two ASCII decimal strings starting at the least significant digits and working their way up to the most significant digits.

The AAM instruction is used to convert the binary product of each multiplication operation to unpacked decimal format.

The AAM instruction is a powerful tool for performing arithmetic operations on ASCII decimal strings. It is easy to use and can be used to implement a variety of arithmetic algorithms.

=====

### ***AAD (ASCII adjust before division)***

=====

The AAD (ASCII adjust before division) instruction is used to convert an unpacked decimal dividend in AX to binary in preparation for executing the DIV instruction. This is necessary because the DIV instruction can only divide binary numbers.

The AAD instruction works by multiplying the AL register by 100 and adding the result to the AH register.

This ensures that the AH register contains the most significant digit of the unpacked decimal dividend and the AL register contains the least significant digit.

Here is an example of how to use the AAD instruction:

```
149 mov ax, 0307h ; dividend
150 aad ; AX = 0025h
151 mov bl, 5 ; divisor
152 div bl ; AX = 0207h
```

After the above code has executed, the AX register will contain the quotient and remainder of the

division operation, respectively. The quotient is stored in the AL register, and the remainder is stored in the AH register.

The AAD instruction can be useful for performing arithmetic operations on ASCII decimal strings. For example, it can be used to divide two ASCII decimal strings, even if they have different lengths.

Here is an example of how to use the AAD instruction to divide two ASCII decimal strings:

```
162 mov esi, offset first_number
163 mov edi, offset second_number
164 mov ecx, length_of_first_number
165
166 loop:
167 mov bl, [esi]
168 aad ; AX = unpacked decimal representation of first number
169
170 mov ah, 0 ; clear AH before division
171 mov al, [edi]
172 div bl ; AX = quotient and remainder of division
173
174 mov [esi], al ; store the quotient
175 mov [edi], ah ; store the remainder
176
177 inc esi ; increment the first number pointer
178 inc edi ; increment the second number pointer
179 dec ecx ; decrement the loop counter
180
181 cmp ecx, 0
182 jne loop ; continue looping if the loop counter is not zero
```

This code will divide the two ASCII decimal strings starting at the most significant digits and working their way down to the least significant digits.

The AAD instruction is used to convert the unpacked decimal representation of the first number to binary before each division operation.

The AAD instruction is a powerful tool for performing arithmetic operations on ASCII decimal strings. It is easy to use and can be used to implement a variety of arithmetic algorithms.

### **Questions:**

**Question:** Write a single instruction that converts a two-digit unpacked decimal integer in AX to ASCII decimal.

Answer: To convert a two-digit unpacked decimal integer in AX to ASCII decimal, you can use the AAM (ASCII Adjust AX After Multiplication) instruction:

aam

**Question:** Write a single instruction that converts a two-digit ASCII decimal integer in AX to unpacked decimal format.

Answer: To convert a two-digit ASCII decimal integer in AX to unpacked decimal format, you can use the AAD (ASCII Adjust AX Before Division) instruction:

aad

**Question:** Write a two-instruction sequence that converts a two-digit ASCII decimal number in AX to

**binary.**

Answer: To convert a two-digit ASCII decimal number in AX to binary, you can use the following two-instruction sequence:

```
186 sub al, 30h ; Convert the tens digit to binary  
187 aam           ; Adjust AX to form the binary value
```

**Question:** Write a single instruction that converts an unsigned binary integer in AX to unpacked decimal.

Answer: To convert an unsigned binary integer in AX to unpacked decimal, you can use the AAD (ASCII Adjust AX Before Division) instruction:

```
aad
```

## ***Packed Decimal Arithmetic, DAA, DAS***

Packed decimal arithmetic is a way of representing and performing arithmetic on decimal numbers using a binary format.

Packed decimal numbers store two decimal digits per byte, with each digit represented by 4 bits.

This makes packed decimal arithmetic a more efficient way to store and manipulate decimal numbers than traditional binary arithmetic, which stores each decimal digit as a separate byte.

Packed decimal arithmetic is often used in financial and business applications, where it is important to be able to perform accurate calculations on large numbers.

It is also used in some scientific and engineering applications, where it is necessary to perform calculations on numbers with a high degree of precision.

*There are two main advantages to using packed decimal arithmetic:*

**Efficiency:** Packed decimal numbers require less storage space than traditional binary numbers. This is because packed decimal numbers store two decimal digits per byte, while traditional binary numbers store each decimal digit as a separate byte.



**Accuracy:** Packed decimal arithmetic can be used to perform calculations with a high degree of

accuracy. This is because packed decimal numbers store two decimal digits per byte, which allows for more precise calculations than traditional binary arithmetic.



*However, there is one main disadvantage to using packed decimal arithmetic:*

**Performance:** Packed decimal arithmetic can be slower than traditional binary arithmetic. This is because packed decimal arithmetic requires additional instructions to convert packed decimal numbers to and from binary numbers, which is necessary for performing arithmetic operations. Overall, packed decimal arithmetic is a powerful tool for performing arithmetic on decimal numbers. It is especially useful for financial and business applications, where it is important to be able to perform accurate calculations on large numbers.



*Here are some examples of packed decimal numbers:*

```
193 bcd1 QWORD 2345673928737285h ; 2,345,673,928,737,285 decimal
194
195 bcd2 DWORD 12345678h ; 12,345,678 decimal
196
197 bcd3 DWORD 08723654h ; 8,723,654 decimal
198
199 bcd4 WORD 9345h ; 9,345 decimal
200
201 bcd5 WORD 0237h ; 237 decimal
202
203 bcd6 BYTE 34h ; 34 decimal
```

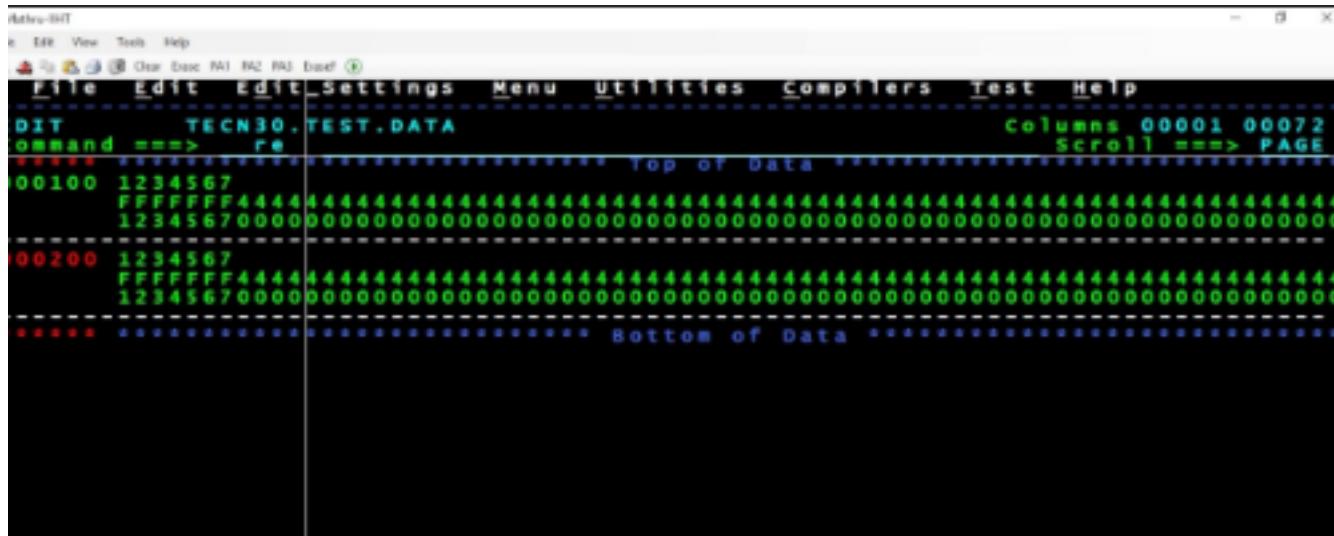
The following two instructions are used to adjust the result of an addition or subtraction operation on packed decimals:

- **DAA (decimal adjust after addition)**
- **DAS (decimal adjust after subtraction)**

These instructions are necessary because packed decimal numbers store two decimal digits per byte. After an addition or subtraction operation, it is possible that the result will be a three-digit number. The DAA and DAS instructions adjust the result to ensure that it is a valid two-digit packed decimal number.

Packed decimal arithmetic can be used to perform all of the basic arithmetic operations, including addition, subtraction, multiplication, and division. However, there are no specific instructions for multiplication and division of packed decimal numbers. This means that packed decimal numbers must be unpacked before these operations can be performed, and then repacked after the operations are

complete.



Despite this disadvantage, packed decimal arithmetic is a powerful tool for performing arithmetic on decimal numbers. It is especially useful for financial and business applications, where it is important to be able to perform accurate calculations on large numbers.

## **DAA (decimal adjust after addition)**

The DAA (decimal adjust after addition) instruction is used to convert the binary sum produced by the ADD or ADC instruction in the AL register to packed decimal format.

This is necessary because packed decimal numbers store two decimal digits per byte. After an

addition operation, it is possible that the result will be a three-digit number.

The DAA instruction adjusts the result to ensure that it is a valid two-digit packed decimal number.

Here is an example of how to use the DAA instruction:

```
208 mov al, 35h  
209 add al, 48h ; AL = 7Dh  
210 daa ; AL = 83h (adjusted result)
```

In this example, the ADD instruction adds the packed decimal numbers 35 and 48. The result is 7Dh, which is a three-digit binary number.

The DAA instruction adjusts the result to 83h, which is the packed decimal representation of the sum of 35 and 48.

The DAA instruction can be used to perform packed decimal addition on any number of digits.

However, it is important to note that the sum variable must contain space for one more digit than the operands. This is because the DAA instruction can generate a carry digit.

The following program adds two 16-bit packed decimal integers and stores the sum in a packed doubleword:

```
; Packed Decimal Example  
(AddPacked.asm)  
; Demonstrate packed decimal addition.
```

```
INCLUDE Irvine32.inc
.data
packed_1 WORD 4536h
packed_2 WORD 7207h
sum DWORD ?
.code
main PROC
    ; Initialize sum and index.
    mov sum, 0
    xor esi, esi

    ; Add low bytes and handle carry.
    add al, BYTE PTR packed_1[esi]
    daa
    mov BYTE PTR sum[esi], al

    ; Add high bytes and include carry.
    inc esi
    add al, BYTE PTR packed_1[esi]
    adc al, BYTE PTR packed_2[esi]
    daa
    mov BYTE PTR sum[esi], al

    ; Add final carry, if any.
    inc esi
    adc al, 0
    mov BYTE PTR sum[esi], al

    ; Display the sum in hexadecimal.
    mov eax, sum
    call WriteHex
    call Crlf
    exit
main ENDP
END main
```

This program uses a loop to add the two packed decimal integers one digit at a time. The DAA instruction is used to adjust the result of each addition operation.

The sum variable is a packed doubleword, which is large enough to store the sum of two 16-bit packed decimal integers.

The DAA instruction is a powerful tool for performing packed decimal arithmetic. It is easy to use and can be used to implement a variety of arithmetic algorithms.

### **DAS (decimal adjust after subtraction)**

The DAS (decimal adjust after subtraction) instruction is used to convert the binary result of a SUB or SBB instruction in the AL register to packed decimal format.

This is necessary because packed decimal numbers store two decimal digits per byte.

After a subtraction operation, it is possible that the result will be a negative three-digit number.

The DAS instruction adjusts the result to ensure that it is a valid two-digit packed decimal number.

Here is an example of how to use the DAS instruction:

```
256 mov bl, 48h  
257 mov al, 85h  
258 sub al, bl ; AL = 3Dh  
259 das ; AL = 37h (adjusted result)
```

In this example, the SUB instruction subtracts the packed decimal numbers 85 and 48. The result is 3Dh, which is a negative three-digit binary number.

The DAS instruction adjusts the result to 37h, which is the packed decimal representation of the difference of 85 and 48.

The DAS instruction can be used to perform packed decimal subtraction on any number of digits.

However, it is important to note that the result variable must contain space for one more digit than the operands.

This is because the DAS instruction can generate a borrow digit.

Here is a pseudocode implementation of the DAS instruction:

```
264 DAS(AL):  
265     if AL < 10:  
266         return AL  
267     else:  
268         AL -= 10  
269         AH += 1  
270         if AH >= 10:  
271             AH -= 10  
272             CF = 1  
273         else:  
274             CF = 0  
275         return AL
```

Explanation:

The DAS instruction begins by checking if the value in the AL register (the low decimal digit) is less than 10. If it is, it means there's no need for adjustment, and it returns AL as it is.

If AL is greater than or equal to 10, it means there's a carry or overflow in the low digit. To correct this:

Subtract 10 from AL, effectively "borrowing" from the low digit. Increment AH (the high digit) to account for the borrow from AL.

Check if AH itself requires adjustment. If AH is now greater than or equal to 10, it means there's a carry in the high digit as well.

If AH needs adjustment, subtract 10 from AH to bring it within the valid range.

Finally, set the Carry Flag (CF) to 1 to indicate that there was a carry or borrow operation.

If AH does not require adjustment, set CF to 0 to indicate that no carry occurred.

In summary, the DAS instruction ensures that after a subtraction operation, the AL and AH registers contain valid packed decimal digits, taking into account any borrows or carries to maintain the integrity of the packed decimal representation.

It is a crucial instruction in packed decimal arithmetic, commonly used in financial and decimal data processing applications.

## ***ADVANCED PROCEDURES***

Skip...

### ***SubRoutine Calls***

#### ***Introduction to Subroutine Calls***

This chapter covers the fundamental structure of subroutine calls, with a focus on the runtime stack. Subroutine calls are common in C and C++ programming, and debugging these calls can require an understanding of the runtime stack.

In C and C++, subroutines are referred to as **functions**, while in Java, they are known as **methods**. In MASM, they are termed **procedures**.

Values passed to a subroutine by a calling program are termed arguments. However, once these values are received by the called **subroutine**, they become **parameters**.

**Stack frames** are used to manage subroutine calls. A **stack frame** is a region of memory on the runtime stack that is used to store the subroutine's local variables and parameters.

- Subroutine calls are a fundamental part of low-level programming.
  - The runtime stack is used to manage subroutine calls.
  - **Arguments** passed to a subroutine **become parameters** within the subroutine.
  - Stack frames are used to store local variables and parameters for subroutines.
- =====

## ***Stack Frames***

=====

In this section, we'll delve into the concept of stack frames, specifically focusing on stack parameters.

## **Stack Parameters**

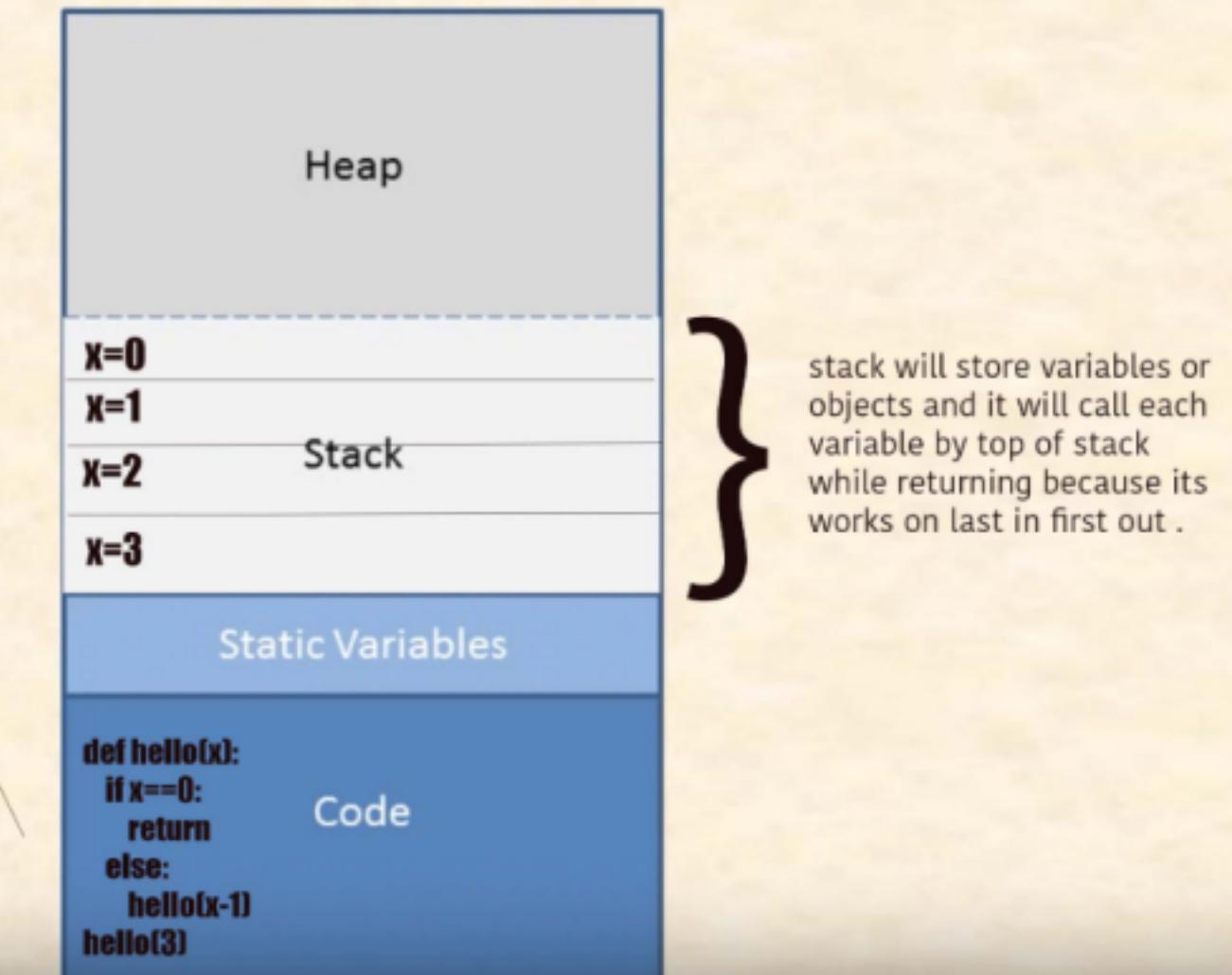
In 32-bit mode, stack parameters are the norm for Windows API functions. In 64-bit mode, Windows functions receive a combination of both register and stack parameters.

To pass a parameter to a subroutine on the stack, the **caller function pushes** the parameter onto the stack before calling the subroutine. The subroutine then accesses the parameter by using the stack pointer register.

### *The Anatomy of a Stack Frame*

A **stack frame**, often referred to as an **activation record**, is a designated area on the stack used for various purposes.

# Memory



It serves as the container for passed arguments, the subroutine return address, local variables, and saved registers. The construction of a stack frame typically involves the following sequential steps:

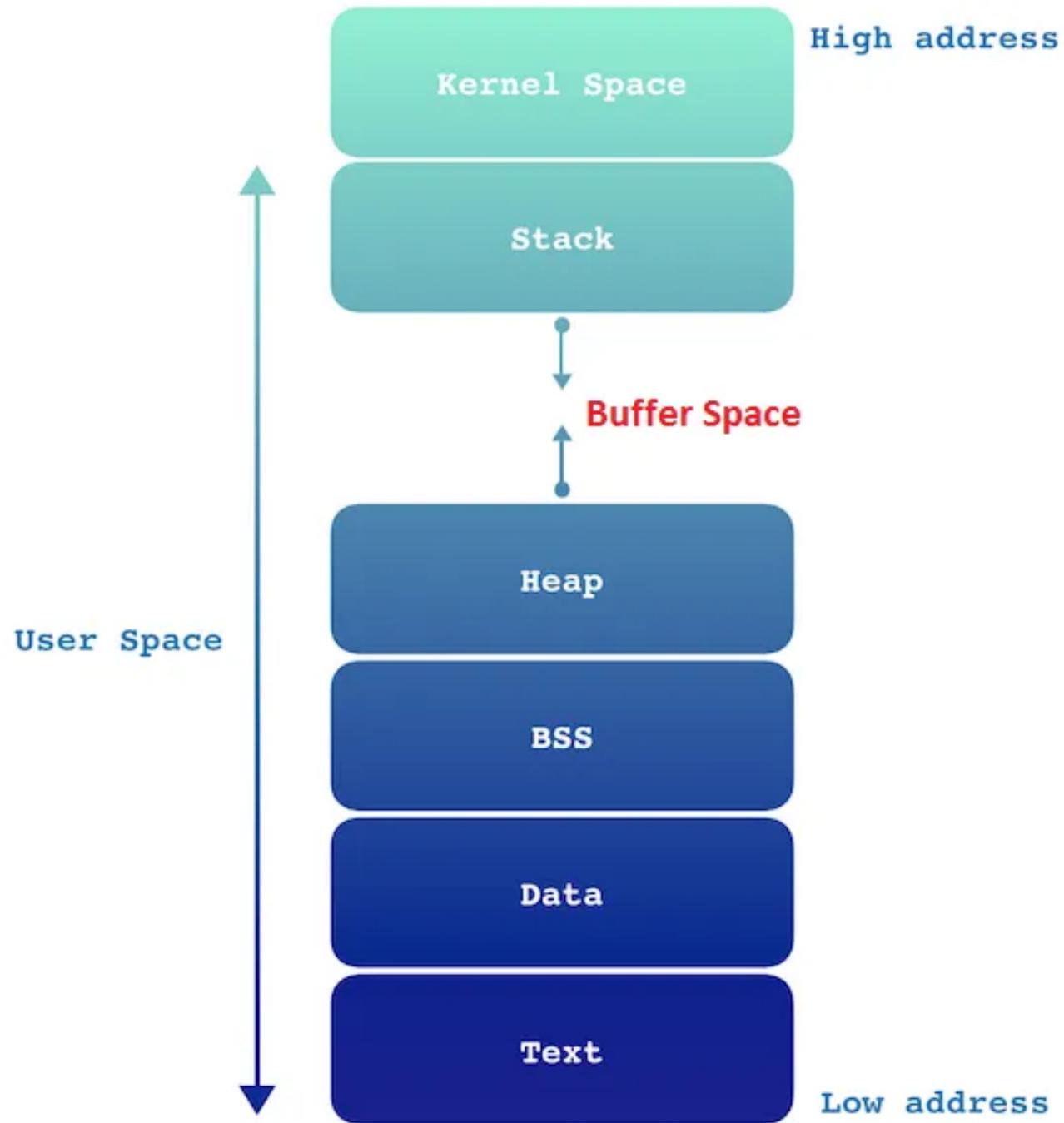
**Passed arguments**, if any, are pushed onto the stack.

As the subroutine begins its execution, the **Extended Base Pointer (EBP)** is pushed onto the stack.

**EBP** is set equal to the value of the **Stack Pointer (ESP)**.

From this point onward, **EBP** acts as a fundamental **reference point** for all the subroutine parameters.

If there are local variables, the **Stack Pointer (ESP)** is decremented to allocate space for these variables on the stack. We said stack pointer starts from highest memory addresses, getting decremented as long as parameters and local variables are being pushed onto the stack. Pops usually increment the stack pointer. So, stack grows downwards.



If any registers need to be preserved, they are pushed onto the stack. The structure and organization of a stack frame can be heavily influenced by a program's memory model and its chosen argument passing convention.

Understanding the concept of passing arguments on the stack is of paramount importance. This is because nearly all high-level programming languages rely on this method.

For instance, when calling functions in the 32-bit Windows Application Programming Interface (API), you'll find it essential to pass arguments on the stack.

However, as you delve into 64-bit programming, you'll encounter a different parameter passing convention, which we will explore in detail in Chapters ahead.

=====

### ***Calls and stack:***

When "Jackie" (an external procedure) calls "Rennex" (an internal procedure) in MASM:

- When "Jackie" calls "Rennex," it's Jackie who pushes Rennex's return address onto the stack.
- This return address points to the location in "Jackie" where execution should resume after "Rennex" completes its tasks.
- So, it is "Jackie" who takes care of preserving the return address for "Rennex."
- When "Rennex" finishes executing and reaches the point where it needs to return, it uses this saved return address to determine where it should return to, which, in this case, is the location

within "Jackie" where the call to "Rennex" occurred.

- So, Jackie, as the calling procedure, takes responsibility for saving and restoring the return address when it calls Rennex.
- Rennex uses this saved return address to correctly return control to Jackie once its execution is complete.
- After the execution of "Rennex" is complete, it's typically the responsibility of the calling procedure, in this case, "Jackie," to manage the stack. Specifically, "Jackie" needs to issue a POP instruction to remove the return address from the stack.

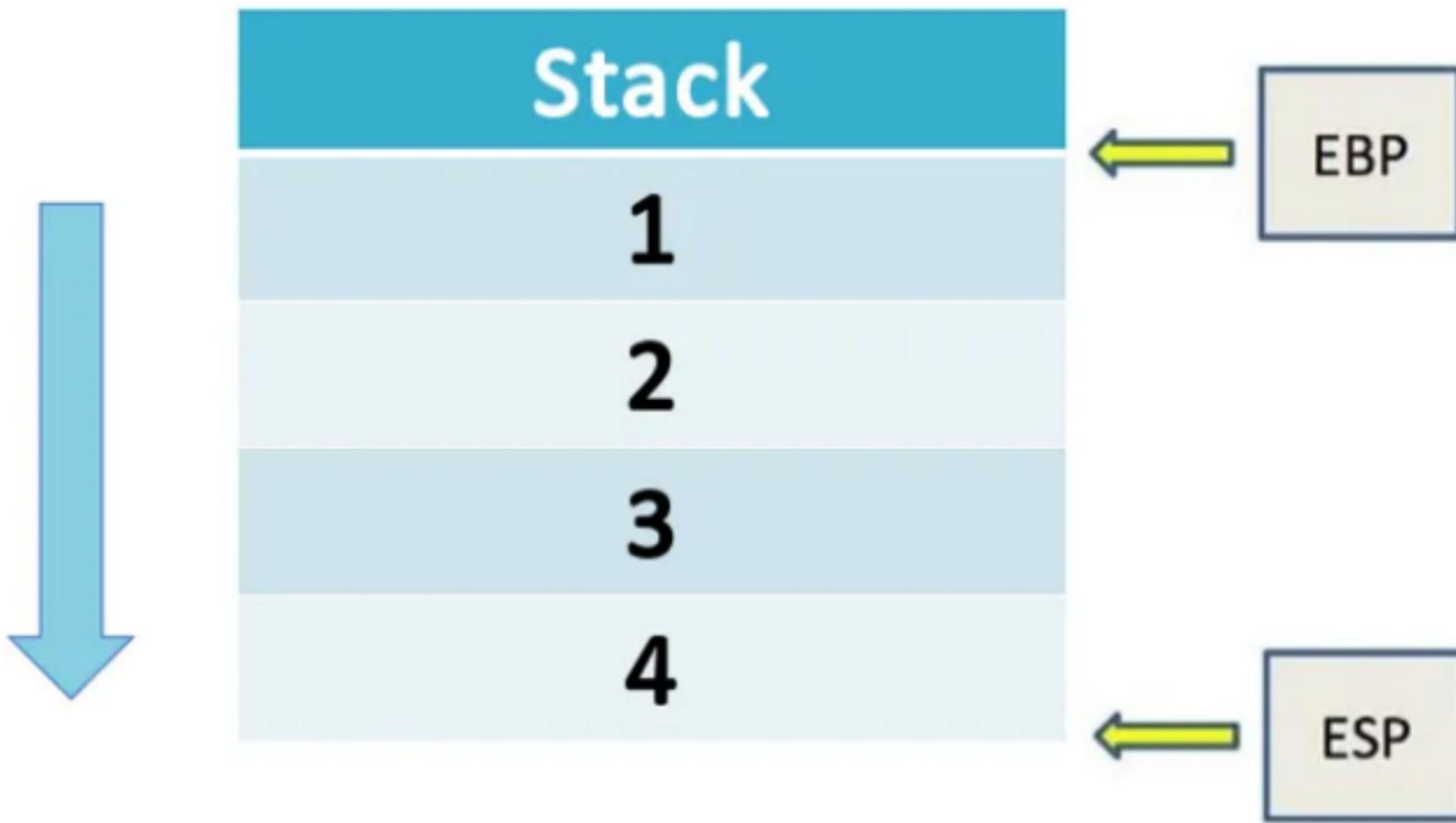
### ***Here's how it works:***

- "Jackie" calls "Rennex" and pushes Rennex's return address onto the stack.
- "Rennex" executes and reaches its return point(**ret instruction**). At this stage, Rennex uses the saved return address to determine where to transfer control back, which is the location within "Jackie" where the call to "Rennex" occurred.
- After Rennex's execution, control returns to "Jackie." Now, it's "Jackie's" responsibility to pop the return address from the stack.
- This is done using the POP instruction, which retrieves the value from the top of the stack and adjusts the stack pointer (ESP) accordingly.
- By executing this POP instruction, "Jackie" effectively removes **Rennex's return address** from the stack, ensuring that the stack is correctly managed and that the program's flow is maintained.

=====

This image can confuse you once you meet something like it.

It is still the same, stack is still pointing to the top of the stack.



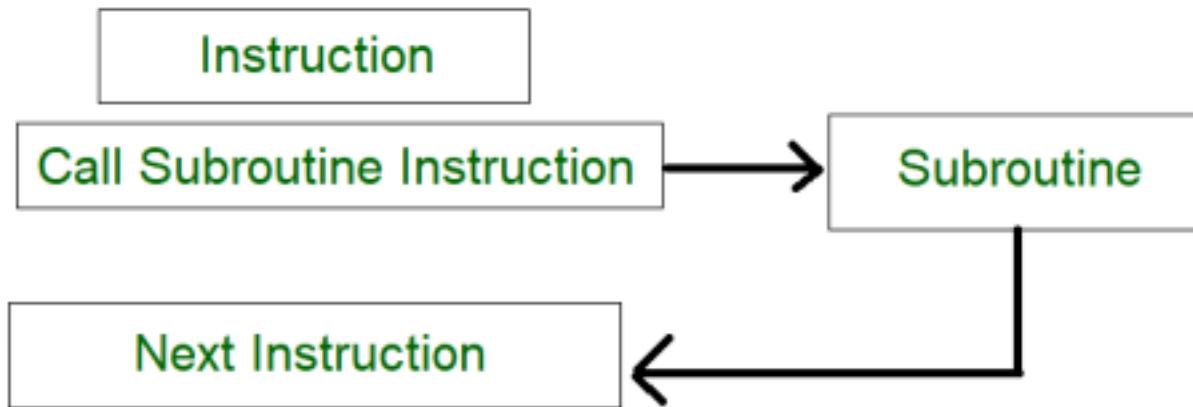
In this example, the stack pointer (ESP) points to the top of the stack frame, which contains the value 4. The extended base pointer (EBP) points to the bottom of the stack frame, which contains the value 1.

This may seem counterintuitive, since stack pointers typically start from higher memory addresses to lower memory addresses.

However, it is important to remember that the stack grows downwards. This means that when a new item is pushed onto the stack, its address is lower than the address of the previous item on the stack.

In the example image, the value 1 was pushed onto the stack first, followed by the value 4. Therefore, the value 1 is at the bottom of the stack frame, and the value 4 is at the top of the stack frame.

EBP is used to keep track of the bottom of the stack frame. This is useful for subroutines, which need to be able to access their local variables and parameters, even if the caller function has pushed new items onto the stack since the subroutine was called.



When a subroutine is called, it pushes the EBP register onto the stack. It then sets the EBP register to the current value of the ESP register. This effectively creates a new stack frame for the subroutine.

The subroutine can then access its local variables and parameters by using the EBP register as a reference point. For example, to access the first local variable, the subroutine would subtract 4 from the EBP register.

To access the second local variable, the subroutine would subtract 8 from the EBP register, and so on.

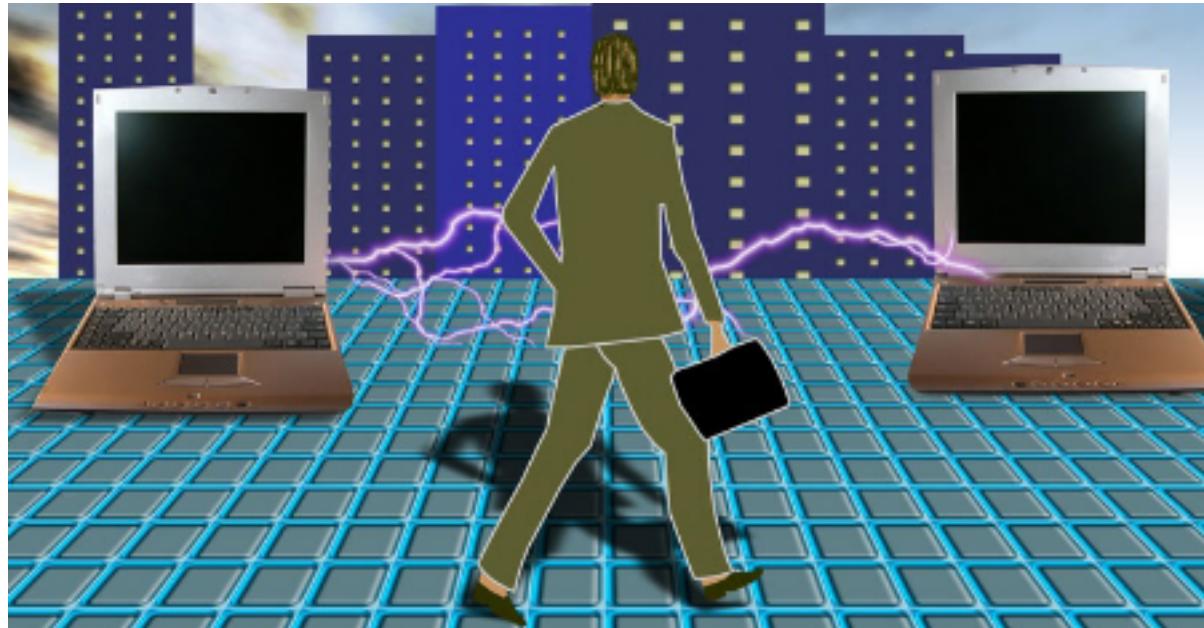
When the subroutine returns, it pops the EBP register from the stack. This restores the stack frame to the state it was in before the subroutine was called.

## *Disadvantages of Register Parameters*

**Register parameters** can be used to pass arguments to subroutines in 32-bit programs using the fastcall calling convention. This can be more efficient than passing arguments on the stack, but it has a number of disadvantages:



**Registers** are used for other purposes. The registers used for parameters are also used for other purposes, such as holding loop counters and operands in calculations. Therefore, any registers used as parameters must be pushed on the stack and restored after the subroutine returns.



**Code clutter.** The extra pushes and pops can create code clutter and make it difficult to maintain. Potential for errors. Programmers must be careful to match every PUSH with a POP, even when there are multiple execution pathways through the code. Otherwise, registers may be left on the stack, which can lead to unexpected behavior.



The following code shows an example of how register parameters can be used to call the DumpMem subroutine from the Irvine32 library:

```
01 push ebx
02 ; save register values
03 push ecx
04 push esi
05 mov esi, OFFSET array
06 ; starting OFFSET
07 mov ecx, LENGTHOF array
08 ; size, in units
09 mov ebx, TYPE array
10 ; doubleword format
11 call DumpMem
12 ; display memory
13 pop esi
14 ; restore register values
15 pop ecx
16 pop ebx
```

Note the order of popping, LIFO - Last In First Out.

This code saves the values of the EAX, EBX, and ECX registers before calling DumpMem.

The DumpMem subroutine then uses these registers to access the memory to be displayed.

After the DumpMem subroutine returns, the values of the EAX, EBX, and ECX registers are restored.

However, this code is also susceptible to errors.

For example, if the eax register equals 1 on line 8, the procedure will not return to its caller on line 17 because three register values were left on the runtime stack.

## ***Stack parameters***

**Stack parameters** offer a more flexible and reliable approach to passing arguments to subroutines. To pass an argument to a subroutine using stack parameters, the argument is simply pushed onto the stack before calling the subroutine.

For example, the following code shows how to call the DumpMem subroutine using stack parameters:

```
54 push
55 TYPE array
56 push
57 LENGTHOF array
58 push
59 OFFSET array
60 call
61 DumpMem
```

This code pushes the type, length, and offset of the array to be displayed onto the stack before calling the DumpMem subroutine. The DumpMem subroutine then uses these values to access the memory to be displayed.

## ***Advantages of Stack Parameters***

Stack parameters have a number of advantages over register parameters:

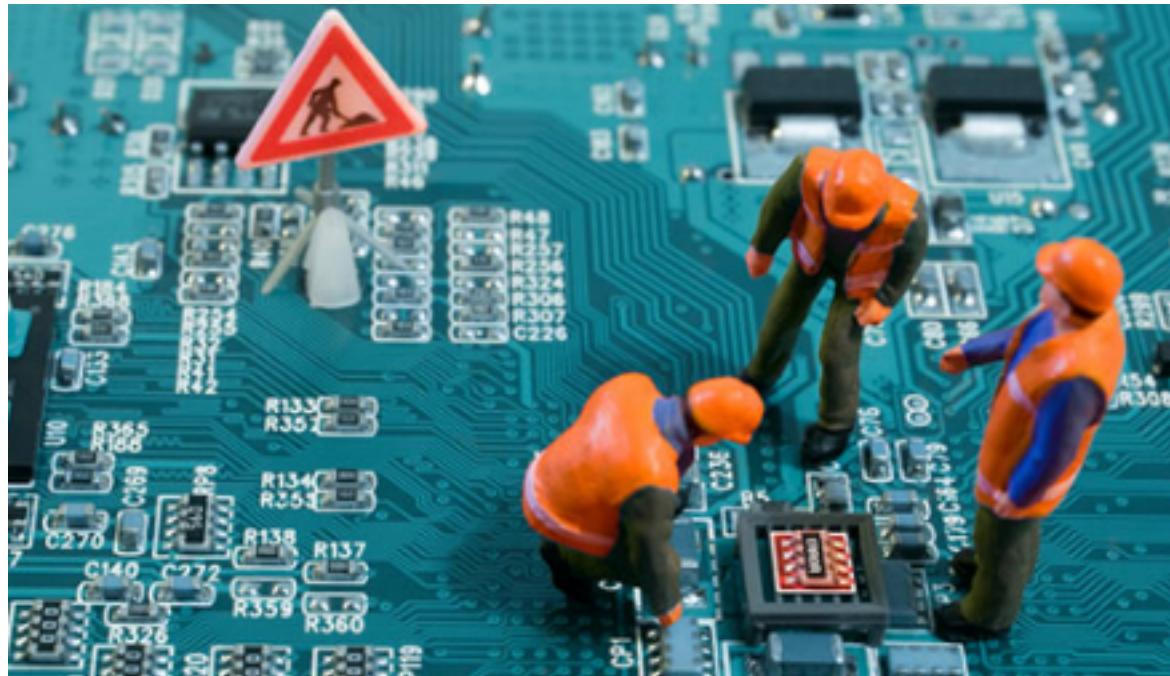
**More flexible.** Stack parameters can be used to pass any number of arguments to a subroutine, regardless of the number of registers available.



**More reliable.** Stack parameters are less susceptible to errors than register parameters. For example, there is no need to worry about matching every PUSH with a POP.



**Easier to maintain.** Code that uses stack parameters is typically easier to read and maintain than code that uses register parameters.



Stack parameters are the preferred way to pass arguments to subroutines in most cases. They offer a more flexible, reliable, and maintainable approach than register parameters.

---

=====

## ***Pass by Value***

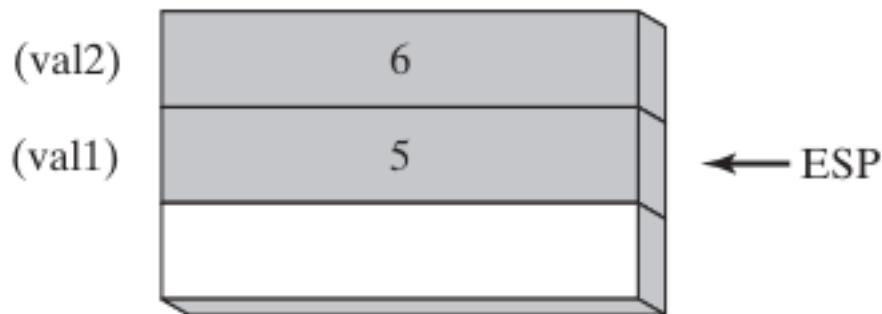
=====

When an argument is passed by value in MASM, a copy of the value is pushed onto the stack. The calling convention used by MASM pushes the arguments in reverse order, meaning that the last argument is pushed onto the stack first.

The following MASM code shows how to call a subroutine named AddTwo, passing it two 32-bit integers by value:

```
73 .data  
74     val1 DWORD 5  
75     val2 DWORD 6  
76 .code  
77     push val2  
78     push val1  
79     call AddTwo
```

After the push instructions have been executed, the stack will look like this:



The diagram above shows the stack just prior to the CALL instruction for the AddTwo subroutine, which is passed two 32-bit integers by value. The arguments are pushed on the stack in reverse order, with val2 on top and val1 below.

The equivalent function call in C++ would be:

```
int sum = AddTwo(val1, val2);
```

In the image, the ESP register is pointing to val1 because it was the last value to be pushed onto the stack.

The stack grows downwards in MASM, meaning that when you push a value onto the stack, the ESP register is decremented by the size of the value.

So, if you push val2 onto the stack, the ESP register will be decremented by 4 bytes. Then, if you push val1 onto the stack, the ESP register will be decremented by another 4 bytes.

As a result, the ESP register will now be pointing to val1, which is the most recently pushed value.

As you can see, the ESP register is pointing to val1, which is the most recently pushed value. When the subroutine AddTwo is called, it will read the arguments from the stack at offsets 0 and 4 bytes, respectively.

This means that it will read val2 from offset 0 and val1 from offset 4 bytes. After the subroutine AddTwo has finished executing, it will pop the arguments off the stack.

This will increment the ESP register by 8 bytes, so that it points to the next value on the stack.

=====

## *Pass by Reference*

---

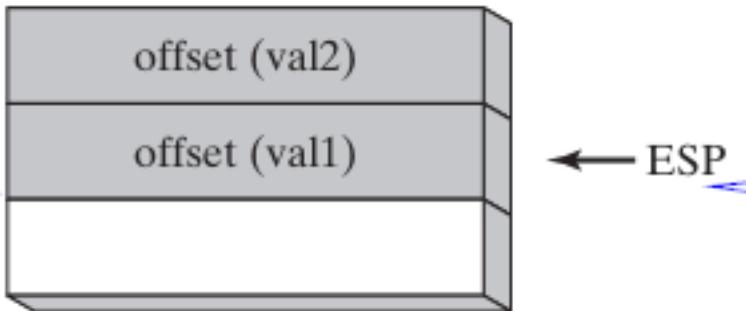
When an argument is passed by reference in MASM, the **address of the argument is pushed onto the stack.**

This allows the subroutine to modify the value of the original variable. The following MASM code shows how to call a subroutine named Swap, passing it two arguments by reference:

```
084 .data
085     val1 DWORD 5
086     val2 DWORD 6
087 .code
088     push OFFSET val2 ; Push the address of the second argument onto the stack.
089     push OFFSET val1 ; Push the address of the first argument onto the stack.
090     call Swap ; Call the Swap subroutine.
```

After the push instructions have been executed, the stack will look like this:

So, ESP is correctly pointing to the latest item that was pushed



Don't confuse, this is the top of the stack, top of the stack is on bottom

Stack grows downwards!!! I won't stop repeating that.

The ESP register points to the top of the stack, so the subroutine Swap can access the arguments by reading from the stack at offsets 0 and 4 bytes, respectively.

The subroutine Swap can then use these addresses to modify the values of the original variables, val1 and val2.

The following C code is equivalent to the MASM code above:

```
094 ;this is wrong code
095 int *val1;
096 int *val2;
097 Swap(&val1, &val2);
098
099 -----
100
101 ;this is the correct code
102 int val1;
103 int val2;
104 int *ptr1 = &val1;
105 int *ptr2 = &val2;
106 Swap(&val1, &val2);
```

The first code is wrong because the pointer variables val1 and val2 are not initialized. This means that they contain garbage values.

- When the Swap function is called, it will expect to receive pointers to integers as arguments. However, the pointer variables val1 and val2 will not contain pointers to integers, so the Swap function will not be able to work correctly.
- Additionally, the Swap function will try to modify the values of the variables pointed to by val1 and val2. However, the pointer variables val1 and val2 will contain garbage values, so the Swap function will try to modify the values of garbage memory. This can lead to undefined behavior.

The second code is correct because the pointer variables ptr1 and ptr2 are initialized to the

addresses of the variables val1 and val2, respectively.

- When the Swap function is called, it will receive pointers to integers as arguments, because the pointer variables ptr1 and ptr2 contain pointers to the variables val1 and val2.
- The Swap function will then be able to modify the values of the variables val1 and val2, because it has pointers to those variables.
- Therefore, the second code is correct, and the first code is wrong.
- It is important to initialize pointer variables before using them. This will ensure that they contain valid addresses, and that they will be able to point to the correct variables.

## ***Conclusion***

Passing by value and passing by reference are two different ways to pass arguments to a subroutine in MASM.

Passing by value is the default, and it is the safest way to pass arguments, because it prevents the subroutine from modifying the original variables.

=====

## ***Passing Arrays***

=====

High-level languages always pass arrays to subroutines by reference because it is more efficient and safer than passing them by value.

When **passing an array by reference**, the address of the array is pushed onto the stack. The subroutine can then get the address from the stack and use it to access the array.

This is much more efficient than passing the array by value, because it does not require each array element to be pushed onto the stack separately.

Additionally, **passing an array by reference is safer** than passing it by value. If the array is large, passing it by value can quickly **overflow the stack**.

When passing an array by reference, the **stack only needs to store the address of the array**, which is much smaller than the size of the array itself.

The following MASM code shows how to pass an array to a subroutine by reference:

```
118 .data
119     array DWORD 50 DUP(?)
120 .code
121     push OFFSET array
122     call ArrayFill
```

The **OFFSET** array operator returns the address of the array variable. So, the **push OFFSET array** instruction pushes the address of the array variable onto the stack.

The **ArrayFill** subroutine can then access the array by reading from the stack at offset 0 bytes. The

following C code is equivalent to the MASM code above:

```
128 void ArrayFill(int *array) {  
129     for (int i = 0; i < 50; i++) {  
130         array[i] = i;  
131     }  
132 }
```

The array parameter is a pointer to the array. The subroutine can then access the array elements by dereferencing the array pointer.

Passing arrays by reference is the standard way to pass arrays to subroutines in both MASM and C. It is more efficient and safer than passing arrays by value.

## ***Accessing Stack Parameters***

The register 'ESP' is used to point to the next item on the stack and is referred to as the 'stack pointer'. Is this false??

Yes, the statement "The register 'ESP' is used to point to the next item on the stack and is referred to as the 'stack pointer'" is wrong. The ESP register points to the top of the stack, which is the most recently pushed item. It only points to the next item after that item has been pushed.

The ESP register points to the top of the stack, which is the most recently pushed item.

The stack grows downwards, so when you push a value onto the stack, the ESP register is decremented

by the size of the value. When you pop a value off the stack, the ESP register is incremented by the size of the value.

So, to answer your question, the ESP register points to the item that was last pushed onto the stack. For example, if you push the values 1, 2, and 3 onto the stack, the ESP register will point to the value 3.

This is because the value 3 was the last value to be pushed onto the stack. If you then pop a value off the stack, the ESP register will point to the value 2. This is because the value 2 is now the most recently pushed item on the stack.

=====

EBP aka the '**stack frame pointer**' serves as an unchanging reference point for data on the stack. This allows the program to work out how far away something in the stack is from this point.

A **stack frame** is a region of memory that is used to store the local variables and parameters of a function. Each function has its own stack frame, which is created when the function is called and destroyed when the function returns.

The **number of stack frames that can be active at any given time depends** on the depth of the call stack. The call stack is a list of all the functions that are currently executing. The depth of the call stack is the number of functions in the call stack.

For example, if the following function calls are made:

```
136 functionA()  
137 functionB()  
138 functionC()
```

Then the call stack will be as follows:

```
140 functionC()  
141 functionB()  
142 functionA()
```

There will be three stack frames active, one for each function in the call stack.

The stack frames are nested, with the stack frame for the most recently called function at the top of the stack.

When a function returns, its stack frame is destroyed and the stack pointer is moved back to the previous stack frame.

The maximum **number of stack frames** that can be active is limited by the size of the stack. The stack is a region of memory, so it has a finite size. If the stack overflows, the program will crash.

Most operating systems have a default stack size, but this can be changed. The stack size can be increased to allow for deeper call stacks, but this will reduce the amount of memory available for other purposes.

The number of stack frames that are typically active in a program depends on the type of program. **Programs that use recursion** can have very deep call stacks. Programs that use a lot of functions can

also have deep call stacks.

In general, it is best to avoid having very deep call stacks. Deep call stacks can lead to stack overflows and can also make programs more difficult to debug.

=====

## ***Accessing Stack Parameters***

=====

High-level languages have various ways of initializing and accessing parameters during function calls. In C and C++, this is done through the use of a stack frame.

A stack frame is a region of memory that is allocated on the stack when a function is called. It contains the function's parameters, local variables, and saved registers.

The following is an example of a simple C function:

```
146 int AddTwo(int x, int y) {  
147     return x + y;  
148 }
```

When this function is called, the compiler will generate a prolog and an epilog. The prolog saves the EBP register and points EBP to the top of the stack. The epilog restores the EBP register and returns to the caller.

The following is an example of an assembly language implementation of the AddTwo function:

```
152 AddTwo PROC  
153     push ebp  
154     mov ebp, esp  
155     sub esp, 8  
156     mov [ebp-4], edi  
157     mov [ebp-8], esi  
158     add eax, [ebp-4]  
159     add eax, [ebp-8]  
160     pop ebp  
161     ret  
162 AddTwo ENDP
```

The first instruction, push ebp, saves the EBP register on the stack. The second instruction, mov ebp, esp, points EBP to the top of the stack. This creates a new stack frame for the AddTwo function.

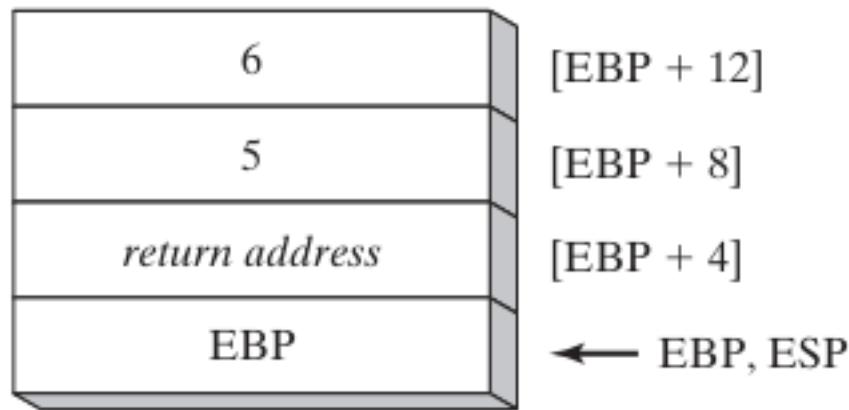
The next instruction, sub esp, 8, reserves 8 bytes of space on the stack for the two parameters. The two following instructions, mov [ebp-4], edi and mov [ebp-8], esi, store the parameters in the stack frame.

The next two instructions, add eax, [ebp-4] and add eax, [ebp-8], add the two parameters together and store the result in the EAX register.

The final two instructions, pop ebp and ret, restore the EBP register and return to the caller. When the AddTwo function is called, the compiler will generate code to push the two parameters on the stack in reverse order.

The first parameter will be pushed last, and the second parameter will be pushed first. This is because the stack grows downwards.

*The following figure shows the contents of the stack frame after the function call AddTwo(5, 6):*



AddTwo could push additional registers on the stack without altering the offsets of the stack parameters from EBP. ESP would change value, but EBP would not.

Here is a rewritten explanation of the image, based on the description you gave me:

The image shows a stack frame for a function with two parameters. The stack frame is a region of memory that is allocated on the stack when a function is called. It contains the function's parameters, local variables, and saved registers.

The stack frame grows downwards, so the parameters are pushed onto the stack in reverse order. The first parameter is pushed last, and the second parameter is pushed first.

The image shows the following:

The function we are dealing with is AddTwo(5,6): 6 is the last parameter, be pushed first into the stack.

When a function is called, the **parameters are pushed onto the stack in reverse order**. This means that the last parameter is pushed first, and the first parameter is pushed last.

6: The second parameter for the function AddTwo.

5: The first parameter for the function AddTwo.

[EBP + 12]: The address of the second parameter for the function AddTwo.

[EBP + 8]: The address of the first parameter for the function AddTwo.

[EBP + 4]: is the address of the EBP register for the calling function. This means that it stores the address of the stack frame for the calling function.

When a function is called, the compiler generates code to save the EBP register and point EBP to the top of the stack. This creates a new stack frame for the function.

The address of the EBP register for the calling function is stored at [EBP + 4]. This ensures that the function can return to the calling function when it is finished executing.

When the function is ready to return, it pops the return address off the stack and restores the EBP register. This restores the stack frame for the calling function, and the function returns.

**EBP:** The base pointer for the stack frame. This is the address of the **start of the stack frame**.

**EBP, ESP:** The EBP and ESP registers.

-----

Here is an example of how EBP is used to access the parameters and local variables for a function:

```
167 ; Function prologue
168 push ebp
169 mov ebp, esp
170 sub esp, 8 ; Reserve space for two parameters
171 mov [ebp-4], edi ; Store the first parameter
172 mov [ebp-8], esi ; Store the second parameter
173
174 ; Function body
175 ; ...
176 ; Function epilogue
177 mov esp, ebp
178 pop ebp
179 ret
180
181 ; Access the first parameter
182 mov eax, [ebp-4]
183
184 ; Access the second parameter
185 mov eax, [ebp-8]
```

## Base Offset Addressing

The following code is a rewritten and explained implementation of AddTwo using base-offset

addressing to access stack parameters:

```
189 ; AddTwo - Add two parameters and return their sum in EAX
190 AddTwo PROC
191     ; Push the base register (EBP) onto the stack
192     push ebp
193     ; Move the stack pointer (ESP) to the base register (EBP)
194     mov ebp, esp
195     ; Calculate the offset of the second parameter (12 bytes from the base of the stack frame)
196     mov eax, 12
197     ; Add the offset to the base register to get the address of the second parameter
198     add eax, ebp
199     ; Load the second parameter into the accumulator (EAX)
200     mov eax, [eax]
201     ; Calculate the offset of the first parameter (8 bytes from the base of the stack frame)
202     mov eax, 8
203     ; Add the offset to the base register to get the address of the first parameter
204     add eax, ebp
205     ; Load the first parameter into the accumulator (EAX)
206     mov eax, [eax]
207     ; Add the first and second parameters
208     add eax, [eax]
209     ; Restore the base register (EBP) from the stack
210     pop ebp
211     ; Return the result in EAX
212     ret
213 AddTwo ENDP
```

## **Explanation:**

The first instruction, **push ebp**, saves the base register (EBP) onto the stack. This is important because EBP will be used as the base register for accessing stack parameters.



The next instruction, **mov ebp, esp**, moves the stack pointer (ESP) to the base register (EBP). This effectively sets the base of the stack frame.

The next two instructions, **mov eax, 12** and **add eax, ebp**, calculate the offset of the second parameter. The second parameter is located 12 bytes from the base of the stack frame.

The next instruction, **mov eax, [eax]**, loads the second parameter into the accumulator (EAX).



The next two instructions, **mov eax, 8** and **add eax, ebp**, calculate the offset of the first parameter. The first parameter is located 8 bytes from the base of the stack frame.

The next instruction, **mov eax, [eax]**, loads the first parameter into the accumulator (EAX).

The next instruction, **add eax, [eax]**, adds the first and second parameters.

The next instruction, **pop ebp**, restores the base register (EBP) from the stack.



This is important because we need to restore the base register before returning from the function. The final instruction, `ret`, returns from the function. Example usage:

```
217 ;Call the AddTwo function
218 call AddTwo
219
220 ;The sum of the two parameters is now in EAX
```

### Benefits of using base-offset addressing:

Base-offset addressing is efficient because it allows us to access stack parameters without having to calculate their absolute addresses.

Base-offset addressing is also **flexible** because it allows us to access stack parameters relative to the base of the stack frame.



This means that we can easily move the stack frame around without having to update the code that accesses stack parameters.

Image 1:

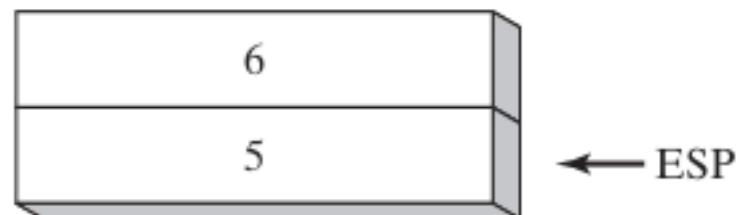
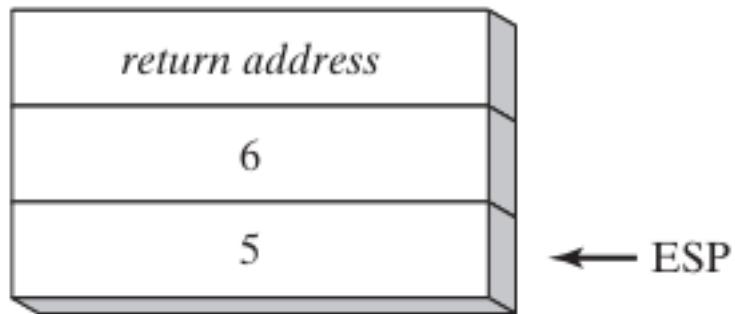


Image 2:



## ***Explicit Stack Parameters***

**Explicit stack parameters** are stack parameters that are referenced by their offset from the base pointer register (EBP).

This is in contrast to **implicit stack parameters**, which are referenced by their position in the stack frame.

The following code shows an example of how to use explicit stack parameters in an assembly language procedure:

```
227 AddTwo PROC  
228     push ebp  
229     mov ebp, esp  
230     mov eax, [ebp + 12] ; y_param  
231     add eax, [ebp + 8] ; x_param  
232     pop ebp  
233     ret  
234 AddTwo ENDP
```

This procedure takes two arguments, x and y, and returns the sum of those two arguments. The arguments are passed to the procedure on the stack.

The procedure begins by pushing the EBP register onto the stack. This is done to preserve the value of EBP, which is used as the base pointer for the stack frame.

The procedure then moves the ESP register to EBP. This sets EBP to point to the top of the stack frame.

Next, the procedure loads the first argument, y, from the stack.

This is done by using the mov instruction to load the value from the address [ebp + 12].

The [ebp + 12] offset is the offset of the first argument from the base pointer register.

The procedure then adds the second argument, x, to the first argument.

This is done by using the add instruction to add the value from the address [ebp + 8] to the value

in the EAX register.

The [ebp + 8] offset is the offset of the second argument from the base pointer register.

Finally, the procedure pops the EBP register from the stack.

This restores the original value of EBP.

The procedure then returns by using the ret instruction.

***The following is a more in-depth explanation of the code:***

**push ebp:** This instruction pushes the EBP register onto the stack. This is done to preserve the value of EBP, which is used as the base pointer for the stack frame.

**mov ebp, esp:** This instruction moves the ESP register to EBP. This sets EBP to point to the top of the stack frame.

**mov eax, [ebp + 12]:** This instruction loads the first argument, y, from the stack. This is done by using the mov instruction to load the value from the address [ebp + 12]. The [ebp + 12] offset is the offset of the first argument from the base pointer register.

**add eax, [ebp + 8]:** This instruction adds the second argument, x, to the first argument. This is done by using the add instruction to add the value from the address [ebp + 8] to the value in the EAX register. The [ebp + 8] offset is the offset of the second argument from the base pointer register.

**pop ebp:** This instruction pops the EBP register from the stack. This restores the original value of EBP.

`ret`: This instruction returns from the procedure. Explicit stack parameters can be useful for making code more readable and maintainable.

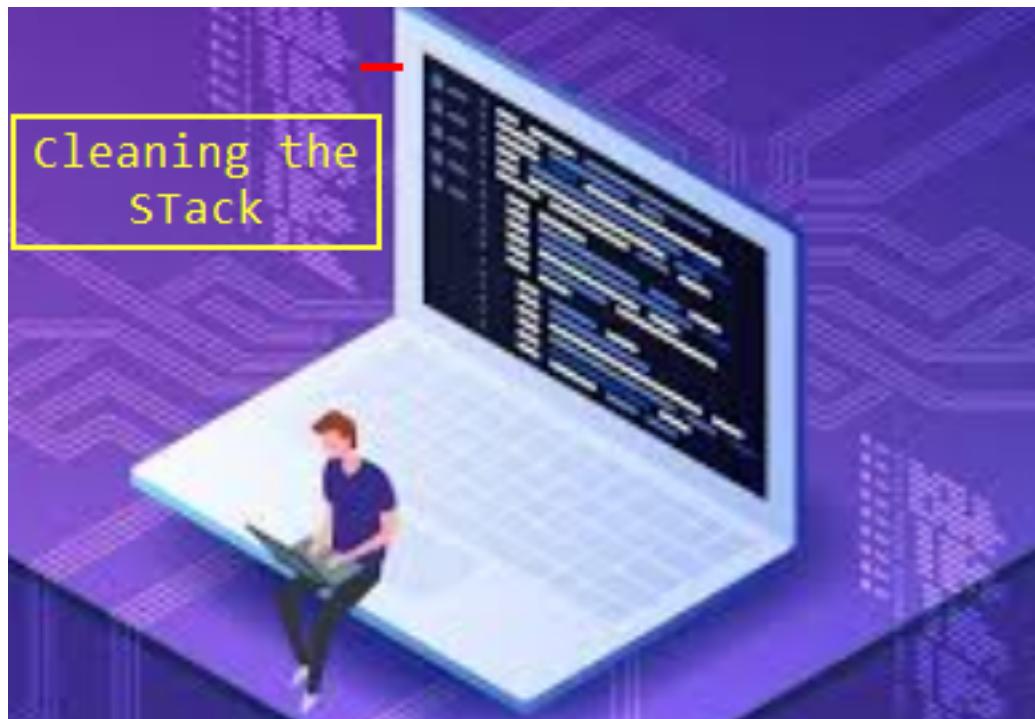
For example, the following code uses symbolic constants to represent the explicit stack parameters:

```
239 y_param EQU [ebp + 12]
240 x_param EQU [ebp + 8]
241
242 AddTwo PROC
243     push ebp
244     mov ebp, esp
245     mov eax, y_param
246     add eax, x_param
247     pop ebp
248     ret
249 AddTwo ENDP
```

This code is more readable and maintainable because it **uses symbolic constants** to represent the explicit stack parameters. This makes it easier to understand what the code is doing and to make changes to the code in the future.

## *Cleaning up the Stack*

To remove parameters from the stack when a subroutine returns, the subroutine must perform stack cleanup. Stack cleanup is the process of removing the subroutine's parameters from the stack so that the stack can be used by other subroutines.



There are two ways to perform stack cleanup:

**Explicit stack cleanup:** The subroutine explicitly removes its parameters from the stack using the POP instruction. This is done by popping the parameters off the stack in reverse order from which they were pushed.



# Explicit

**Implicit stack cleanup:** The subroutine leaves the stack cleanup to the caller. This is done by using a CALL instruction that specifies the number of bytes to be removed from the stack when the subroutine returns.



# Implicit

The following example shows how to perform **explicit stack cleanup** in the AddTwo subroutine:

```
273 AddTwo PROC  
274     push ebp  
275     mov ebp, esp  
276     ; ...  
277     ; Calculate the sum of the two parameters.  
278     ; ...  
279     pop ebp  
280     ret  
281 AddTwo ENDP
```

The `pop ebp` instruction at the end of the subroutine removes the base pointer register (EBP) from the stack.

This is done to restore the original value of EBP, which was pushed onto the stack at the beginning of the subroutine.

The following example shows how to use **implicit stack cleanup** in the AddTwo subroutine:

```
289 AddTwo PROC  
290     push ebp  
291     mov ebp, esp  
292     ; ...  
293     ; Calculate the sum of the two parameters.  
294     ; ...  
295     ret 8  
296 AddTwo ENDP
```

The `ret 8` instruction at the end of the subroutine tells the caller to remove 8 bytes from the stack when the subroutine returns.

This is the same as the size of the two parameters that were pushed onto the stack at the beginning of the subroutine.

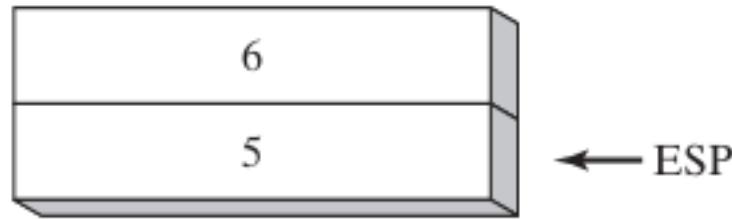
=====

### *Stack OverFlow*

=====

Assuming that `AddTwo` leaves the two parameters on the stack, the following illustration shows the stack after returning from the call:

This image shows the stack after the call `AddTwo` instruction in `main` has been executed:



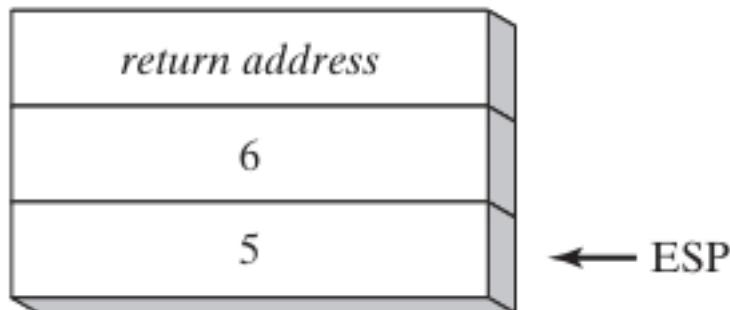
Inside main, we might try to ignore the problem and hope that the program terminates normally. But if we were to call AddTwo from a loop, the stack could overflow.

Each call uses 12 bytes of stack space - 4 bytes for each parameter, plus 4 bytes for the CALL instruction's return address. A more serious problem could result if we called Example1 from main, which in turn calls AddTwo:

```
319 main PROC
320     call Example1
321     exit
322 main ENDP
323
324 Example1 PROC
325     push 6
326     push 5
327     call AddTwo
328     ret          ;stack is corrupted
329 Example1 ENDP
```

In the image below, the return address for the `call AddTwo` instruction is still on the stack. This is because the `AddTwo` subroutine did not perform any stack cleanup.

When the `RET` instruction in Example1 is about to execute, `ESP` points to the integer 5 rather than the return address that would take it back to `main`:



The `RET` instruction loads the value 5 into the instruction pointer and attempts to transfer control to memory address of 5.

Assuming that this address is outside the program's code boundary, the processor issues a **runtime exception**, which tells the OS to terminate the program.

### ***Are these notes well organized??***

Yes, your notes are in good order and well explained. You have clearly explained the problem of stack corruption and how to prevent it. You have also provided a good example of how stack corruption can occur.

**Stack overflow:** Stack overflow occurs when the stack uses more memory than is available. This can happen if a subroutine leaves its parameters on the stack when it returns.



**Stack corruption:** Stack corruption occurs when the stack is not used correctly. This can happen if a subroutine does not perform stack cleanup or if a subroutine accesses the stack incorrectly.



**Preventing stack corruption:** To prevent stack corruption, subroutines must perform stack cleanup when they return. This can be done explicitly or implicitly. Your example of Example1 is a good example of how stack corruption can occur.



If the Example1 subroutine does not perform stack cleanup, the stack will be corrupted when the `ret` instruction is executed. This will cause the program to terminate with a runtime exception.

## *Calling Conventions*

Here is a simplified explanation of the **C** and **STDCALL** calling conventions.

=====

**C calling convention:**

=====

**Parameter passing:** Parameters are pushed onto the stack in reverse order.

## Passing Parameters

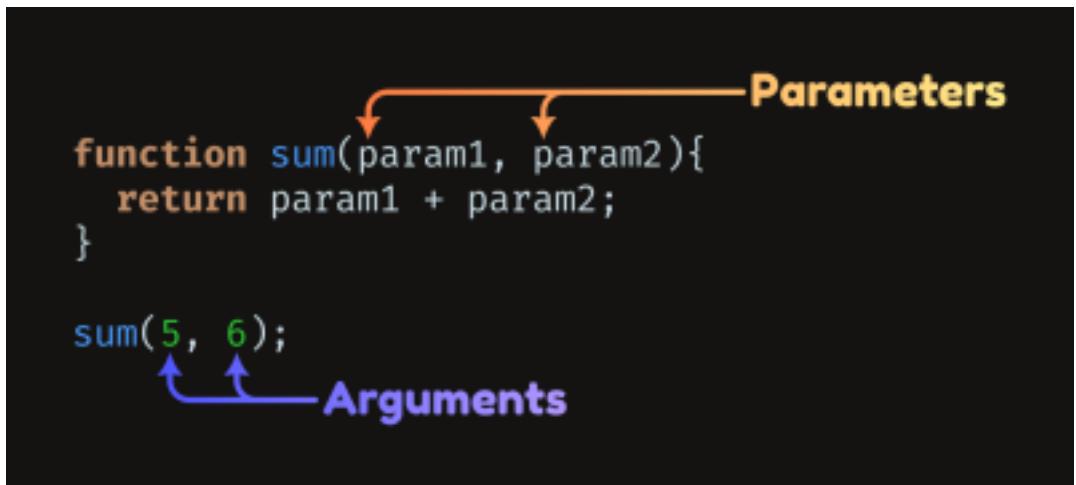
**Stack cleanup:** The caller is responsible for cleaning up the stack after the function returns.



## *STDCALL calling convention:*

---

Parameter passing: Parameters are pushed onto the stack in reverse order.



Stack cleanup: The called function is responsible for cleaning up the stack after it returns.



In other words, with the C calling convention, the caller has to tell the called function how many parameters it is passing.

With the STDCALL calling convention, the called function knows how many parameters it is receiving because it is responsible for cleaning up the stack.

Here is an example of a function call using the C calling convention:

```
337 int AddTwo(int a, int b) {
338     return a + b;
339 }
340
341 int main() {
342     int a = 5;
343     int b = 6;
344
345     ;Push the parameters onto the stack in reverse order.
346     push(b);
347     push(a);
348
349     ;Call the AddTwo function.
350     call AddTwo;
351
352     ;Add the size of the parameters to the stack pointer to clean up the stack.
353     add esp, 8;
354
355     ;Store the return value in a variable.
356     int result = eax;
357
358     ; ...
359 }
```

Here is an example of a function call using the STDCALL calling convention:

```
361 int AddTwo(int a, int b) {  
362     return a + b;  
363 }  
364  
365 int main() {  
366     int a = 5;  
367     int b = 6;  
368  
369     ;Push the parameters onto the stack in reverse order.  
370     push(b);  
371     push(a);  
372  
373     ;Call the AddTwo function.  
374     call AddTwo;  
375  
376     ;...  
377 }
```

As you can see, the only difference between the two calling conventions is who is responsible for cleaning up the stack. With the C calling convention, the caller is responsible. With the STDCALL calling convention, the called function is responsible.

The STDCALL calling convention is used by the Windows API, so it is important to be familiar with it if you are writing programs that call Windows API functions.

if you are writing programs that call Windows API functions.

=====

*Useless stuff you don't need:*

=====

There are many different calling conventions, each with its own advantages and disadvantages. Some common calling conventions include:

**C calling convention:** This calling convention is used by the C and C++ programming languages. It is simple to implement, but it can be inefficient for functions with many parameters.

# function Calling in C Programming Language

**STDCALL calling convention:** This calling convention is used by the Windows API. It is more efficient than the C calling convention for functions with many parameters, but it is more complex to

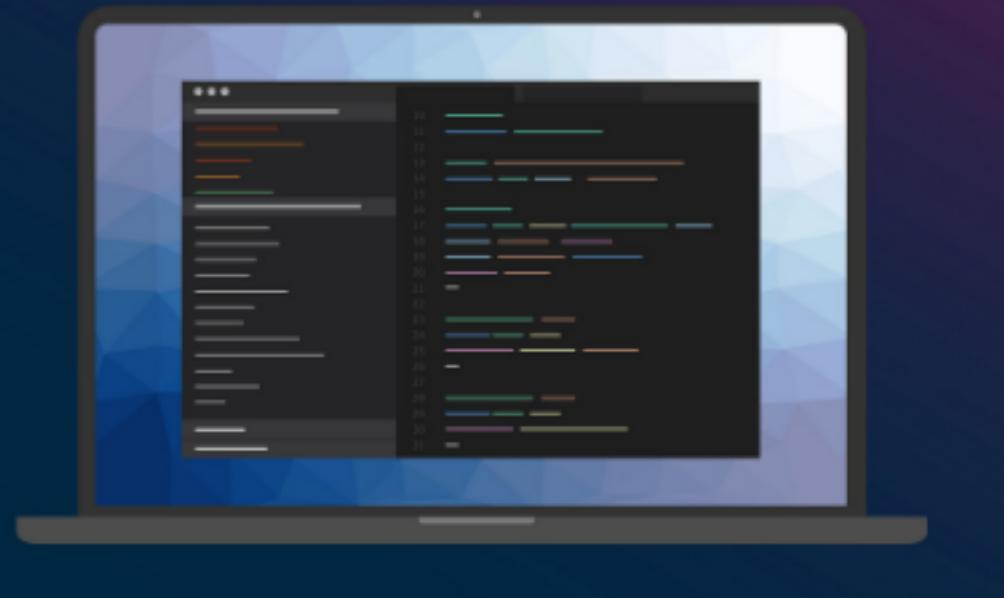
implement. **x86 calling convention:** This calling convention is used by 64-bit x86 processors. It is similar to the STDCALL calling convention, but it has been optimized for 64-bit performance.



**Pascal calling convention:** This calling convention is used by the Pascal programming language. It is similar to the C calling convention, but it passes the first parameter in a register instead of on the stack.

# PASCAL

PROGRAMMING LANGUAGE



**FORTRAN calling convention:** This calling convention is used by the FORTRAN programming language. It is different from the other calling conventions in that it passes all parameters in registers, rather than on the stack. In addition to these general-purpose calling conventions, there are also many specialized calling conventions that are used for specific purposes.



# Calling Fortran code with Julia



For example, there are calling conventions for operating system calls, library functions, and even specific types of functions, such as floating-point functions.

The number of calling conventions that exist depends on the specific processor architecture and programming language. However, there are a few common calling conventions that are used by most processors and programming languages.

## *Saving and Restoring Registers*

Subroutines often save the current contents of registers on the stack before modifying them.

This is a good practice, because the original values can be restored just before the subroutine returns.

This ensures that the subroutine does not modify registers that are used by the caller, and that the caller's state is preserved.

The ideal time to save registers is just after setting EBP to ESP, and just before reserving space for local variables.

This is because the stack grows below EBP, so pushing registers does not affect the displacement from EBP of parameters already on the stack.

Here is an example of a subroutine that saves and restores registers:

```
381 MySub PROC  
382     push ebp  
383     mov ebp, esp  
384     push ecx  
385     push edx  
386     ; ...  
387     pop edx  
388     pop ecx  
389     pop ebp  
390     ret  
391 MySub ENDP
```

The subroutine first pushes the base pointer (EBP) onto the stack. This saves the current value of the stack pointer, which is used as the base of the stack frame for the subroutine. The subroutine then moves the stack pointer to EBP, which makes EBP the base of the stack frame for the subroutine.

The subroutine then pushes the ECX and EDX registers onto the stack. These are two commonly used registers, so it is a good practice to save them before modifying them.

The subroutine then performs its work.

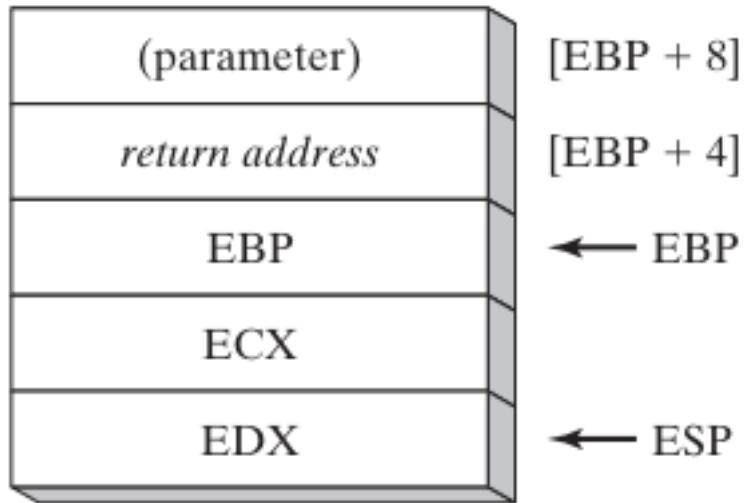
Before returning, the subroutine restores the ECX and EDX registers from the stack. It also pops the base pointer (EBP) from the stack, which restores the stack pointer to its original value.

## ***Stack Frames***

A stack frame is a region of the stack that is used to store the local variables and parameters of a subroutine. The stack frame is created when the subroutine is called, and it is destroyed when the subroutine returns.

The base pointer (EBP) register is used to point to the base of the stack frame. This allows the subroutine to access its local variables and parameters without having to keep track of the stack pointer explicitly.

Here is a diagram of a stack frame for the MySub procedure:



The **stack pointer (ESP)** is a register that points to the top of the stack. When a function is called, the caller pushes the function's parameters onto the stack and then calls the function. The called function then allocates space for its local variables on the stack.

In the case of **MySub**, the caller pushes two parameters onto the stack. When MySub is called, it first **saves the current value of the EBP register** onto the stack. This is because the EBP register is used to reference the stack frame for the current function.

MySub then **moves the ESP register into the EBP register**. This makes the EBP register point to the top of the stack frame for MySub. **MySub can then access its local variables** by using the EBP register as a base pointer.

The **first parameter to MySub is stored at [ESP + 8]**. This is because the ESP register points to the top of the stack, and the first parameter was pushed onto the stack before the EBP register was pushed onto the stack.

The **return address for MySub is stored at [ESP + 4]**. This is because the return address is the next

instruction that will be executed after MySub returns. The return address is pushed onto the stack by the caller before the caller calls MySub.

The **EBP register** is stored at [ESP]. This is because the EBP register is used to reference the stack frame for the current function.

The **ECX and EDX registers** are stored at [EBP - 4] and [EBP - 8], respectively. This is because the ECX and EDX registers are **callee-saved registers**. This means that the caller is responsible for saving and restoring the values of these registers before and after calling MySub.

MySub can access its local variables by using the EBP register as a base pointer. For example, to access the first local variable, MySub would use the following instruction:

**mov eax, [ebp + 12]**

This instruction would copy the contents of the memory location at offset 12 from the EBP register into the EAX register.

When MySub is finished executing, it **pops the ECX and EDX registers off of the stack**. It then pops the EBP register off of the stack. This restores the EBP register to its previous value, which was the EBP register for the calling function.

Finally, **MySub executes the RET instruction**. This instruction returns to the caller and pops the return address off of the stack. The caller then executes the next instruction after the call to MySub.

**Conclusion**

Saving and restoring registers is a good practice for subroutines, because it ensures that the subroutine does not modify registers that are used by the caller, and that the caller's state is preserved. Stack frames are used to store the local variables and parameters of a subroutine.

## *Local Variables*

The C++ function MySub() declares two local variables, X and Y. When this function is compiled into machine language, the following assembly code is generated:

```
01 void MySub()
02 {
03     int X = 10;
04     int Y = 20;
05 }
```

```
242 MySub PROC  
243     push ebp  
244     mov ebp, esp  
245     sub esp, 8 ; create locals  
246     mov DWORD PTR [ebp-4], 10 ; X  
247     mov DWORD PTR [ebp-8], 20 ; Y  
248     mov esp, ebp ; remove locals from stack  
249     pop ebp  
250     ret  
251 MySub ENDP
```

This assembly code shows how local variables are allocated on the stack. The push ebp and mov ebp, esp instructions save the current value of the EBP register onto the stack. The EBP register is used to reference the stack frame for the current function.

The sub esp, 8 instruction allocates 8 bytes on the stack for the two local variables. This is because each stack entry defaults to 32 bits, and each variable's storage size is rounded upward to a multiple of 4.

The mov DWORD PTR [ebp-4], 10 and mov DWORD PTR [ebp-8], 20 instructions initialize the local variables X and Y to the values 10 and 20, respectively.

When MySub() is finished executing, the mov esp, ebp and pop ebp instructions remove the local variables from the stack and restore the previous value of the EBP register.

The image you sent shows the stack frame for the MySub() function. The following table shows the contents of the stack frame:

Variable	Bytes	Stack Offset
X	4	EBP - 4
Y	4	EBP - 8
(parameter)	4	EBP + 8
return address	4	EBP + 4
EBP	4	ESP
ECX	4	ESP + 4
EDX	4	ESP + 8

The first column in the table shows the name of the variable. The second column shows the number of bytes that the variable occupies in memory. The third column shows the stack offset of the variable. The stack offset is the distance from the base pointer (EBP) to the variable.

The X and Y variables are local variables. This means that they are created and destroyed within the current function. Local variables are allocated on the stack. In the image you sent, the X variable is located at stack offset -4, and the Y variable is located at stack offset -8.

The (parameter) variable is a parameter to the current function. Parameters are passed to functions by pushing them onto the stack before the function is called. In the image above, the parameter is located at stack offset 8.

The return address is the address of the instruction that will be executed after the current function returns. The return address is pushed onto the stack by the caller before the caller calls the current function. In the image you sent, the return address is located at stack offset 4.

The EBP register is the base pointer register. The base pointer register is used to reference the stack frame for the current function. In the image above, the EBP register is located at the same address as the ESP register.

The ECX and EDX registers are **callee-saved registers**. This means that the caller is responsible for saving and restoring the values of these registers before and after calling the current function. In the image you sent, the ECX and EDX registers are located at stack offsets 4 and 8, respectively.

---

The stack offset of a variable is the distance from the base pointer (EBP) to the variable. In the case of the X and Y variables, their stack offsets are -4 and -8, respectively.

This means that the X variable is located 4 bytes below the EBP register, and the Y variable is located 8 bytes below the EBP register.

The stack grows downward in memory, so the stack offset of a variable is the distance from the base pointer (EBP) to the variable in bytes, going down the stack.

The **EBP register** is used to reference the stack frame for the current function. This means that the

X and Y variables can be accessed by using the EBP register as a base pointer.

```
262 Stack pointer (ESP)
263 .
264 .
265 .
266 [ESP + 8] : (parameter)
267 [ESP + 4] : Return address
268 [ESP]      : EBP
269 [EBP - 4] : X
270 [EBP - 8] : Y
```

The following diagram shows the stack frame for the MySub() function, with the stack offsets of the X and Y variables indicated: (you can use the image above or this one):

Variable	Bytes	Stack Offset
X	4	EBP - 4
Y	4	EBP - 8

The EBP register is used to reference the **stack frame for the current function**. This means that the X and Y variables can be accessed by using the EBP register as a base pointer. For example, to access the X variable, the following instruction would be used:

`mov eax, [ebp-4]`

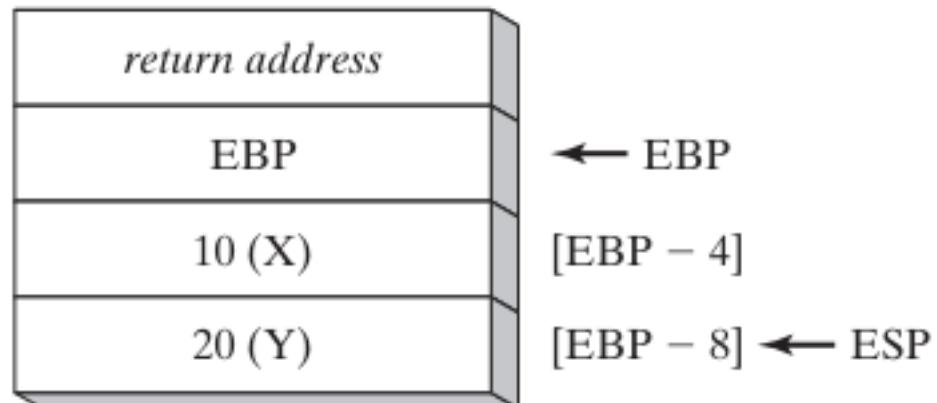
This instruction would copy the contents of the memory location at stack offset -4 into the EAX register.

The stack offset of a variable is also important for function calls. When a function is called, the caller pushes the function's parameters onto the stack.

The called function then allocates space on the stack for its local variables. The stack offset of a parameter is its distance from the EBP register in the caller's stack frame.

The stack offset of a local variable is its distance from the EBP register in the called function's stack frame.

Stack frame after creating local variables.



## *Local Variables Symbols*

It is often useful to define a symbol for each local variable's offset. This can make the code easier to read and maintain.

To define a symbol for a local variable's offset, you use the **EQU directive**. For example, the following code defines a symbol for the X local variable:

```
X_local EQU DWORD PTR [ebp-4]
```

This symbol can then be used to access the X local variable, as shown in the following code:

```
292 MySub PROC  
293     push ebp  
294     mov ebp, esp  
295     sub esp, 8 ; create locals  
296     mov X_local, 10 ; X  
297     mov esp, ebp ; remove locals from stack  
298     pop ebp  
299     ret  
300 MySub ENDP
```

The line of code `X_local EQU DWORD PTR [ebp-4]` defines a symbol called `X_local`. This symbol is equivalent to the value of the memory location at stack offset -4.

The stack offset of a variable is the distance from the base pointer (EBP) to the variable in bytes, going down the stack. In this case, the stack offset is -4, which means that the variable is located 4 bytes below the base pointer.

The `DWORD PTR` keyword specifies that the variable is a 32-bit `DWORD` (double word).

This symbol can then be used to access the local variable `X`, as shown above.

Using symbols to access local variables can make the code easier to read and maintain. For example, the following code is easier to read than the previous code:

```
304 MySub PROC  
305     push ebp  
306     mov ebp, esp  
307     sub esp, 8 ; create locals  
308     mov [ebp-4], 10 ; X  
309     mov esp, ebp ; remove locals from stack  
310     pop ebp  
311     ret  
312 MySub ENDP
```

## *Reference Parameters*

**Reference parameters** are passed to a procedure by address.

This means that the procedure receives a pointer to the actual variable, instead of a copy of the variable's value.

This allows the procedure to modify the value of the variable in the caller's scope.

To access a reference parameter, the procedure can use base-offset addressing with the EBP register.

The EBP register points to the base of the current stack frame.

Each reference parameter is pushed onto the stack in reverse order, so the offset of the first

reference parameter is 12 bytes from EBP.

For example, the following code loads the pointer to the array passed to the ArrayFill procedure into the ESI register:

```
mov esi, [ebp+12]
```

The ESI register can then be used to access the elements of the array.

The ArrayFill procedure fills an array with a pseudorandom sequence of 16-bit integers.

It receives two arguments: a pointer to the array and the array length.

The first argument is passed by reference and the second argument is passed by value.

Here is a simplified version of the ArrayFill procedure:

```
417 ArrayFill PROC
418     ;Save the stack frame pointer
419     push ebp
420     mov ebp, esp
421     ;Save the general-purpose registers
422     pushad
423     ;Get the pointer to the array
424     mov esi, [ebp+12]
425     ;Get the array length
426     mov ecx, [ebp+8]
427     ;Fill the array with pseudorandom values
428     L1:
429         mov eax, 10000h
430         call RandomRange
431         mov [esi], ax
432         add esi, TYPE WORD
433         loop L1
434     ;Restore the general-purpose registers
435     popad
436     ;Restore the stack frame pointer
437     pop ebp
438     ;Return
439     ret
440 ArrayFill ENDP
```

The first few lines of the procedure save the stack frame pointer and the general-purpose registers. Then, the procedure gets the pointer to the array and the array length from the stack.

The procedure then loops through the array and fills each element with a pseudorandom value using the RandomRange function. The RandomRange function is a library function that generates a random number between 0 and FFFFh.

After the loop, the procedure restores the general-purpose registers and the stack frame pointer, and then returns.

-----

The ArrayFill procedure is written in assembly language and serves the purpose of filling an array with pseudorandom values. Here's a detailed breakdown of how it works:

#### ***Procedure Declaration:***

The ArrayFill PROC statement indicates the beginning of the procedure.

#### ***Prologue - Saving the Stack Frame:***

push ebp: This instruction pushes the current stack frame pointer onto the stack. mov ebp, esp: Here, the current stack pointer (esp) is copied into the base pointer (ebp). This step is essential for setting up a new stack frame for the function.

#### ***Saving General-Purpose Registers:***

pushad: The pushad instruction is used to save the values of all general-purpose registers (EAX, ECX, EDX, EBX, ESI, EDI, and EBP) on the stack. This is done to preserve the state of these registers during the execution of the procedure.

### ***Getting Array Pointer and Length:***

`mov esi, [ebp+12]`: This line loads the address of the array into the esi register. `mov ecx, [ebp+8]`: The value of the array length is loaded into the ecx register. These values are passed as parameters to the function, with `[ebp+12]` representing the array pointer and `[ebp+8]` representing the array length. **Array Filling Loop:** The labeled loop, L1, is the core of the procedure.

### ***In each iteration:***

`mov eax, 10000h`: The eax register is loaded with the value `10000h` (40960 in decimal). `call RandomRange`: This likely calls a function named `RandomRange` to generate pseudorandom values. `mov [esi], ax`: The result of the `RandomRange` call is stored in the memory location pointed to by esi. `add esi, TYPE WORD`: The esi register is incremented by the size of a word, effectively pointing to the next element in the array.

### ***Loop L1:***

This checks if the loop counter (ecx) is not zero and decrements it. If it's not zero, the code jumps back to L1, continuing the array filling process.

### ***Epilogue - Restoring Registers and Exiting:***

`popad`: This instruction restores the values of the general-purpose registers to their original state. `pop ebp`: It pops the stack frame pointer (ebp) to restore the previous stack frame.

`ret`: Finally, the `ret` instruction is used to return from the procedure, effectively exiting it. In summary, the `ArrayFill` procedure follows a standard structure: it saves the current state, sets up a loop to fill the array with pseudorandom values, and then restores the saved state before exiting. It's a crucial part of the code for filling an array with random data.

### ***Conclusion***

Reference parameters are a powerful feature of assembly language that allow procedures to modify the values of variables in the caller's scope. By understanding how to use reference parameters, you can write more efficient and reusable assembly code.

## **LEA Instruction**

The OFFSET directive in assembly language allows you to get the address of a variable or label at compile time. However, **it does not work with stack parameters** because the addresses of stack parameters are not known until runtime.

The following statement would not assemble:

```
mov esi, OFFSET [ebp-30]
```

This is because the **compiler does not know the value of ebp at compile time**. ebp is the base pointer register, and it points to the top of the stack frame.

The offset of the local variable myString from the base pointer is -30, but the value of the base pointer is not known until runtime.

```
404 void makeArray( )
405 {
406     char myString[30];
407     for( int i = 0; i < 30; i++ )
408         myString[i] = '*';
409 }
```

The code then enters a for loop that iterates from  $i = 0$  to  $i = 29$ . In each iteration, it assigns the character '\*' to the  $i$ -th element of the `myString` array.

Effectively, this code initializes all 30 elements of the `myString` array to the character '\*'. After the function is called, the `myString` array will contain 30 asterisk characters, like this:

**The LEA instruction, on the other hand, can be used to calculate the address of a stack parameter at runtime.** The LEA instruction takes a memory operand as its operand and loads the effective address of the operand into the destination register.

The following assembly language code is equivalent to the C++ code in the example:

```
386 makeArray PROC  
387     push ebp  
388     mov ebp, esp  
389     sub esp, 32 ; myString is at EBP-30  
390     lea esi, [ebp-30] ; load address of myString  
391     mov ecx, 30 ; loop counter  
392     L1:  
393     mov BYTE PTR [esi], '*' ; fill one position  
394     inc esi ; move to next  
395     loop L1 ; continue until ECX = 0  
396     add esp, 32 ; remove the array (restore ESP)  
397     pop ebp  
398     ret  
399 makeArray ENDP
```

The LEA instruction calculates the effective address of the operand [ebp-30] and loads it into the register esi. The operand [ebp-30] references the local variable myString because myString is located 30 bytes below the base pointer register.

Once you have loaded the address of the stack parameter into a register, you can use the register to access the stack parameter. For example, the following assembly language code shows how to use the register esi to access the local variable myString:

```
mov BYTE PTR [esi], '*' ; fill one position
```

This code stores the character '\*' in the first byte of the local variable myString.

The LEA instruction is a powerful tool that can be used to calculate the addresses of memory locations at runtime. It is especially useful for working with stack parameters and dynamic data structures.

## ***ENTER and LEAVE Instructions***

The ENTER and LEAVE instructions are used to manage stack frames in assembly language.

The **ENTER instruction** creates a stack frame for a called procedure, while the **LEAVE instruction** destroys the stack frame for the current procedure.

The ENTER instruction takes two operands: the number of bytes of stack space to reserve for local variables and the lexical nesting level of the procedure.

The lexical nesting level is the number of nested function calls that have occurred to reach the current function. In most cases, the lexical nesting level is zero.

### **The ENTER instruction performs the following actions:**

Pushes the value of the EBP register onto the stack.

Sets the EBP register to the address of the current stack frame.

Reserves the specified number of bytes of stack space for local variables.

**The LEAVE instruction performs the following actions:**

Pops the value of the EBP register from the stack.

Restores the ESP register to its value before the ENTER instruction was executed.

The following example shows how to use the ENTER and LEAVE instructions to create and destroy a stack frame for a procedure:

```
416 MySub PROC
417     enter 8, 0 ; Reserve 8 bytes of stack space for local variables.
418     ; ...
419     leave ; Destroy the stack frame.
420     ret
421 MySub ENDP
```

It is important to note that the ENTER and LEAVE instructions should be used together. If you use the ENTER instruction to create a stack frame, you must also use the LEAVE instruction to destroy the stack frame. Otherwise, the stack space that you reserved for local variables will not be released.

The image that you provided shows the stack before and after the ENTER instruction has executed. The ENTER instruction has pushed the value of the EBP register onto the stack and set the EBP register to the address of the current stack frame. The ENTER instruction has also reserved 8 bytes of stack space for local variables.

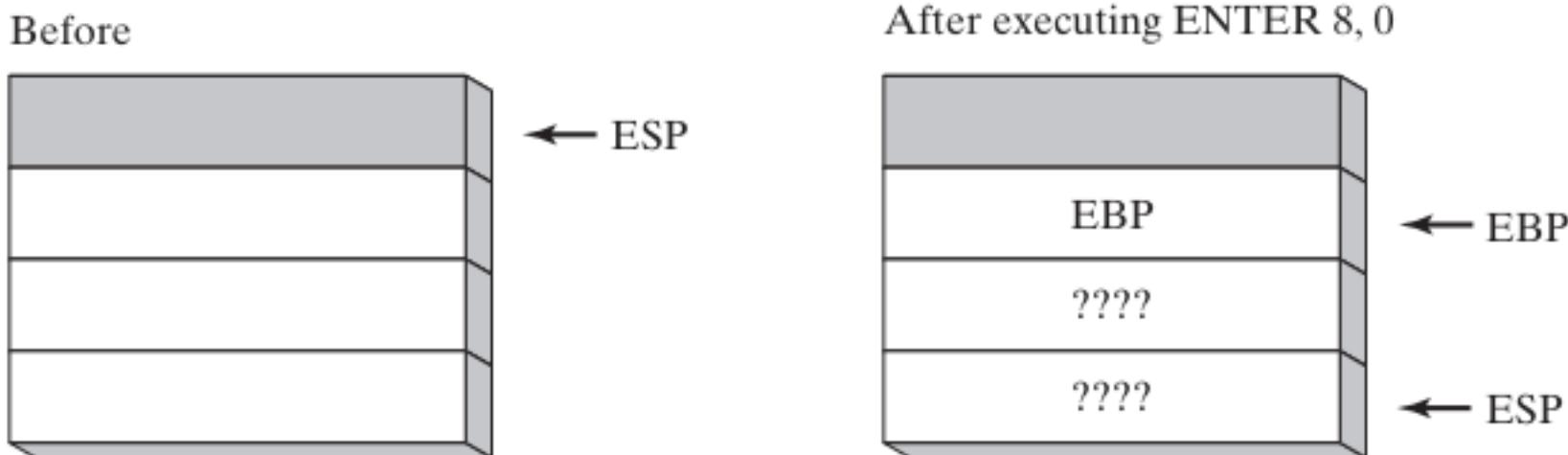
## Why is it important to use the ENTER and LEAVE instructions together?

The ENTER and LEAVE instructions should be used together because they work together to manage the stack.

The ENTER instruction creates a stack frame for a called procedure, while the LEAVE instruction destroys the stack frame for the current procedure.

If you use the ENTER instruction to create a stack frame but do not use the LEAVE instruction to destroy the stack frame, **the stack space that you reserved for local variables will not be released**.

This will cause the stack to grow larger and larger, and it could eventually cause the program to crash.



The image above shows the stack before and after the ENTER instruction has executed. The ENTER instruction has pushed the value of the EBP register onto the stack and set the EBP register to the address of the current stack frame. The ENTER instruction has also reserved 8 bytes of stack space.

for local variables.

## ***LOCAL Directive***

The LOCAL directive in assembly language is used to declare local variables.

It can be used to declare named local variables of any type, including standard types such as BYTE, DWORD, and PTR WORD, as well as user-defined types such as structures.

The LOCAL directive must be placed on the line immediately following the PROC directive. Its syntax is as follows:

### **LOCAL varlist**

Where varlist is a list of variable definitions, separated by commas. Each variable definition takes the following form:

**label:type** where label is the name of the local variable and type is the type of the local variable.

For example, the following assembly language code declares a local variable named var1 of type BYTE:

```
MySub PROC  
LOCAL var1:BYTE
```

The following assembly language code declares a doubleword local variable named temp of type DWORD and a variable named SwapFlag of type BYTE:

```
BubbleSort PROC  
LOCAL temp:DWORD, SwapFlag:BYTE
```

The following assembly language code declares a PTR WORD local variable named pArray, which is a pointer to a 16-bit integer:

```
Merge PROC  
LOCAL pArray:PTR WORD
```

The following assembly language code declares a local variable named TempArray which is an array of 10 doublewords:

```
LOCAL TempArray[10]:DWORD
```

The LOCAL directive reserves stack space for the local variables that it declares. The amount of stack space reserved depends on the type and size of each local variable.

For example, a BYTE variable requires 1 byte of stack space, while a DWORD variable requires 4 bytes of stack space.

Local variables are accessible within the procedure in which they are declared. They are not accessible to other procedures.

It is important to note that the LOCAL directive is not equivalent to the ENTER instruction. The ENTER instruction creates a stack frame for a called procedure, while the LOCAL directive simply declares local variables.

The ENTER instruction must be used in conjunction with the LEAVE instruction to destroy the stack frame.

The LOCAL directive is a convenient and easy-to-use way to declare local variables in assembly language.

It is a good idea to use the LOCAL directive for all local variables, even if they are only used in a single procedure. This will make your code more readable and maintainable.

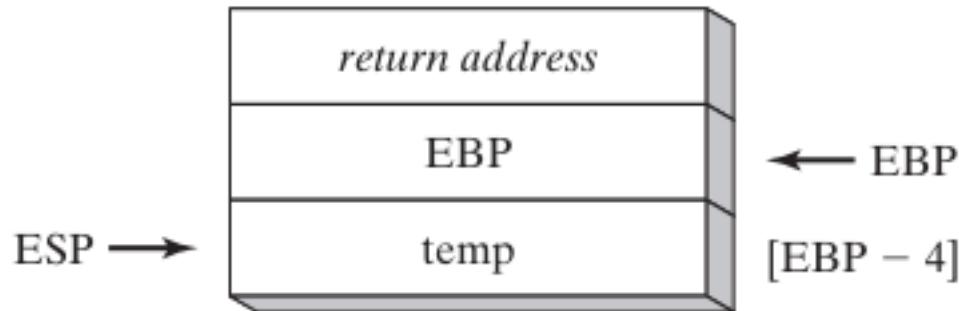
The following is a more in-depth explanation of the MASM code generation for the LOCAL directive:

```
485 Example1 PROC  
486     LOCAL temp:DWORD  
487     mov  
488     eax,temp  
489     ret  
490 Example1 ENDP
```

MASM generated code:

```
499 push  
500 ebp  
501 mov  
502 ebp,esp  
503 add  
504 esp,0FFFFFFFCh  
505 ; add -4 to ESP  
506 mov  
507 eax,[ebp-4]  
508 leave  
509 ret
```

Stack frame diagram:



### ***The MASM code generator works as follows:***

It pushes the value of the EBP register onto the stack.

This saves the current value of the base pointer register. It sets the EBP register to the address of the current stack frame.

This makes the base pointer register point to the top of the new stack frame.

It subtracts 4 bytes from the ESP register.

This reserves 4 bytes of stack space for the local variable temp.

It loads the value of the local variable temp into the EAX register.

It calls the leave instruction to destroy the stack frame and restore the ESP register to its value before the enter instruction was executed.

It returns from the procedure.

## ***Image explanation:***

The image shows the stack before and after the Example1 procedure has executed.

The Example1 procedure has reserved 4 bytes of stack space for the local variable temp. The local variable temp is now located at the address ebp-4.

The esp register points to the top of the stack.

The ebp register points to the base of the stack frame. The base of the stack frame is the address of the first local variable.

The example1 procedure has no parameters.

Therefore, the ebp register points to the same address before and after the procedure has executed.

## **Microsoft x64 calling convention**

The **Microsoft x64 calling convention** is a set of rules that govern how parameters are passed to and from functions in 64-bit Windows programs. It is used by C and C++ compilers, as well as by the Windows API library.

***Here is a summary of the key points of the Microsoft x64 calling convention:***

The first four parameters to a function are passed in registers: RCX, RDX, R8, and R9. Additional parameters are pushed onto the stack in left-to-right order.

Parameters less than 64 bits long are not zero extended, so the upper bits have indeterminate values. The return value from a function is returned in the RAX register, if it is an integer whose

size is less than or equal to 64 bits.

Otherwise, the return value is placed on the stack and RCX points to its location. The caller is responsible for allocating at least 32 bytes of shadow space on the runtime stack, so called functions can optionally save the register parameters in this area.

The stack pointer (RSP) must be aligned on a 16-byte boundary when calling a function.

The caller is responsible for removing all parameters and shadow space from the runtime stack after the called function has finished.

*Here are some additional details about the Microsoft x64 calling convention:*

The CALL instruction subtracts 8 from the RSP register, since addresses are 64 bits long.

The RAX, RCX, RDX, R8, R9, R10, and R11 registers are often altered by functions, so if the calling program wants them preserved, it must push them onto the stack before the function call and pop them off the stack afterwards.

The values of the RBX, RBP, RDI, RSI, R12, R14, R14, and R15 registers must be preserved by functions.

The Microsoft x64 calling convention is a complex topic, but it is important to understand it if you are writing 64-bit Windows programs.

## **Questions**

1. (True/False): A subroutine's stack frame always contains the caller's return address and the

**subroutine's local variables.**

Answer: True

Explanation: A subroutine's stack frame is a region of memory on the stack that is used to store information about the subroutine, such as its local variables and the caller's return address. The caller's return address is the address of the instruction in the calling function that will be executed after the subroutine returns.

**2. (True/False): Arrays are passed by reference to avoid copying them onto the stack.**

Answer: True

Explanation: Arrays are typically passed by reference to functions to avoid copying them onto the stack. This is because arrays can be very large, and copying them onto the stack would be inefficient.

**3. (True/False): A subroutine's prologue code always pushes EBP on the stack.**

Answer: True

Explanation: The prologue code for a subroutine is the code that is executed at the beginning of the subroutine. The prologue code typically saves the value of the EBP register on the stack. The EBP register is used to point to the base of the current stack frame.

**4. (True/False): Local variables are created by adding a positive value to the stack pointer.**

Answer: True

Explanation: Local variables are created by adding a positive value to the stack pointer. This value is the size of the local variable in bytes.

**5. (True/False): In 32-bit mode, the last argument to be pushed on the stack in a subroutine call is stored at location EBP + 8.**

Answer: False

Explanation: In 32-bit mode, the last argument to be pushed on the stack in a subroutine call is stored at location EBP + 4.

**6. (True/False): Passing by reference means that an argument's address is stored on the runtime stack.**

Answer: True

Explanation: Passing by reference means that an argument's address is stored on the runtime stack. This means that the function can directly access the argument in the calling function.

**7. What are the two common types of stack parameters?**

Answer: The two common types of stack parameters are:

**Value parameters:** Value parameters are copied onto the stack when the function is called. When the function returns, the changes made to the parameter on the stack are not reflected in the calling function.

**Reference parameters:** Reference parameters are passed by address. When a function is called with a reference parameter, the stack contains the address of the parameter in the calling function. Changes made to the parameter in the called function are also reflected in the calling function.

Example:

```
void swap_values(int a, int b) {  
    // a and b are value parameters  
  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
void swap_references(int* a, int* b) {  
    // a and b are reference parameters
```

```
int temp = *a;
*a = *b;
*b = temp;
}

int main() {
    int x = 10;
    int y = 20;

    // Call swap_values() with the value parameters x and y
    swap_values(x, y);

    // x and y will still be 10 and 20, respectively

    // Call swap_references() with the reference parameters &x and &y
    swap_references(&x, &y);

    // x and y will now be 20 and 10, respectively

    return 0;
}
```

### ***Explanation:***

In the above example, the `swap_values()` function takes two value parameters, `a` and `b`. When the function is called, the values of `a` and `b` are copied onto the stack. The function then swaps the values of the two parameters on the stack. When the function returns, the values of `a` and `b` in the calling function are not affected.

The `swap_references()` function takes two reference parameters, `a` and `b`. When the function is called, the stack contains the addresses of the `a` and `b` parameters in the calling function. The function then swaps the values of the two parameters in the calling function. When the function returns, the

values of a and b in the calling function are affected.

## Recursion

Recursion is a programming technique where a function calls itself directly or indirectly. It can be a powerful tool for solving complex problems, but it is important to understand how it works and how to avoid writing recursive functions that can cause stack overflows.

### Endless Recursion

The example of endless recursion you provided is a good illustration of what can go wrong when recursion is not used correctly.

The Endless procedure calls itself repeatedly without ever checking for a **base case**. As a result, the stack will continue to grow until it overflows, causing the program to crash.

To rewrite the Endless procedure correctly, we need to add a base case. This is a condition that will cause the procedure to terminate instead of calling itself again.

In the case of the Endless procedure, the base case could be something like "if the input is 0, then return".

Here is a rewritten version of the Endless procedure that includes a base case:

```
519 ; Endless Recursion (Endless.asm)
520 INCLUDE Irvine32.inc
521 .data
522     endlessStr BYTE "This recursion never stops",0
523 .code
524     main PROC
525         call
526         Endless
527         exit
528     main ENDP
529     Endless PROC
530         mov ecx, 1 ; input parameter
531         ; base case
532         cmp ecx, 0
533         je endless_exit
534
535         ; recursive call
536         call Endless
537
538         ; decrement input parameter
539         dec ecx
540         ; and call again
541         jmp Endless
542
543         endless_exit:
544             ret
545     Endless ENDP
546 END main
```

This rewritten version of the Endless procedure will now terminate correctly when the input is 0. It will also print the message "This recursion never stops" to the console before it terminates.

## When to Use Recursion

Recursion is not a good choice for all problems. It can be inefficient and difficult to debug. However, it can be a powerful tool for solving problems that have repeating patterns. For example, recursion is often used to implement algorithms for traversing linked lists and trees.

If you are considering using recursion in your program, it is important to make sure that the problem you are trying to solve is a good fit for recursion. You should also carefully design your recursive function to avoid stack overflows.

Pushed on Stack	Value in ECX	Value in EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

## Recursively Calculating a Sum

A recursive procedure is one that calls itself. This can be useful for solving problems that can be broken down into smaller subproblems of the same type.

To calculate the sum of the integers from 1 to n, we can use the following recursive procedure:

```
554 CalcSum(n):  
555     if n == 0:  
556         return 0  
557     else:  
558         return n + CalcSum(n - 1)
```

This procedure works by recursively calling itself to calculate the sum of the integers from 1 to n - 1, and then adding n to the result. The base case is when n == 0, in which case the sum is simply 0.

The following table shows a stack trace for the recursive call of CalcSum(5):

Stack Frame	ECX (counter)	EAX (sum)
main()	5	0
CalcSum(5)	4	0
CalcSum(4)	3	4
CalcSum(3)	2	7
CalcSum(2)	1	10
CalcSum(1)	0	11

*Explanation of the Table:*

The stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed. The stack frame contains the return address, which is the address of the next instruction to be executed after the recursive call returns.

The ECX register contains the counter value for the current recursive call. The EAX register contains the sum of the integers calculated so far.

At the first recursive call to CalcSum(5), the counter value is 4 and the sum is 0. The program

calculates the sum of the integers from 1 to 4 by recursively calling CalcSum(4).

At the second recursive call to CalcSum(4), the counter value is 3 and the sum is 0. The program calculates the sum of the integers from 1 to 3 by recursively calling CalcSum(3).

This process continues until the base case is reached, when  $n == 0$ . At this point, the program returns 0 from the recursive call. The program then returns from the recursive call to CalcSum(3), and so on.

By the time the program returns from the recursive call to CalcSum(5), the sum of the integers from 1 to 5 has been calculated and stored in the EAX register. The program can then return the sum from the main() function.

```
571 ;Sum of Integers (RecursiveSum.asm)
572 INCLUDE Irvine32.inc
573 .code
574 main PROC
575     mov ecx, 5      ; Set ECX to 5, the number of integers to sum.
576     mov eax, 0      ; Initialize EAX to 0; it will hold the sum.
577     call CalcSum   ; Call the CalcSum function to calculate the sum.
578 L1:
579     call WriteDec  ; Display the result in EAX.
580     call Crlf      ; Print a new line.
581     exit
582 main ENDP
583 ;-----
584 CalcSum PROC
585     ; Calculates the sum of a list of integers
586     ; Receives: ECX = count
587     ; Returns: EAX = sum
588 ;-----
589     cmp ecx, 0      ; Compare ECX (counter) with 0.
590     jz L2           ; If it's zero, jump to L2 and quit.
591     add eax, ecx    ; Add ECX to EAX, updating the sum.
592     dec ecx         ; Decrement the counter.
593     call CalcSum   ; Recursively call CalcSum to process the next integer.
594
595 L2:
596     ret
597 CalcSum ENDP
598 end main
```

This code first sets up the main procedure, where it initializes ecx to 5 (the number of integers to sum) and eax to 0 (to store the sum). It then calls the CalcSum procedure to calculate the sum. Afterward, it prints the result using WriteDec and adds a new line with Crlf.

The CalcSum procedure is a recursive function that calculates the sum of integers. It checks if ecx (the counter) is zero; if not, it adds the current value of ecx to the sum in eax, decrements ecx, and then makes a recursive call to CalcSum. This process continues until ecx reaches 0, at which point the function returns (ret).

## Factorial of an Integer

The Factorial procedure uses recursion to calculate the factorial of a number. It receives one **stack parameter**, N, which is the number to calculate. The calling program's return address is automatically pushed on the stack by the CALL instruction.

The first thing Factorial does is to push EBP on the stack, to save the base pointer to the calling program's stack. It then sets EBP to the beginning of the current stack frame. This allows the procedure to access its parameters and local variables using base-offset addressing.

Next, Factorial checks the base case, which is when N equals zero. In this case, Factorial returns 1, which is the factorial of 0.

If N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter. This process continues until the base case is reached.

When Factorial returns from a recursive call, it multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

## **Example Stack Trace:**

The following table shows a stack trace for a call to Factorial(3):

Stack Frame	EBP	ESP	N
main()	0x00000000	0x00000004	3
Factorial(3)	0x00000004	0x00000000	3
Factorial(2)	0x00000000	0x00000004	2
Factorial(1)	0x00000004	0x00000000	1
Factorial(0)	0x00000000	0x00000004	0

The stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed. The stack frame contains the return address, which is the address of the next instruction to be executed after the recursive call returns.

The EBP register contains the base pointer to the current stack frame. The ESP register contains the stack pointer, which points to the top of the stack.

The N register contains the value of the parameter passed to Factorial.

At the first recursive call to Factorial(3), the EBP register is set to the beginning of the current stack frame. The N register is loaded with the value 3, which is the parameter passed to Factorial.

Factorial checks the base case, which is when N equals zero. Since N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter.

At the second recursive call to Factorial(2), the EBP register is set to the beginning of the new stack frame. The N register is loaded with the value 2, which is the parameter passed to Factorial.

Factorial checks the base case, which is when N equals zero. Since N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter.

This process continues until the base case is reached, when N equals zero. At this point, Factorial returns 1, which is the factorial of 0.

The program then returns from the recursive call to Factorial(2). The N register is loaded with the value 2, which is the result of the recursive call.

Factorial multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

The program then returns from the recursive call to Factorial(3). The N register is loaded with the value 6, which is the result of the recursive call.

Factorial multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

The program then returns to the main() function. The EAX register contains the value 6, which is the factorial of 3.

factorial of 3.

---

Let's break down the provided assembly code for calculating the factorial of an integer, explained above, step by step, and I'll explain the key parts in detail.

```
605 INCLUDE Irvine32.inc           ;Calculating a Factorial (Fact.asm)
606 .code
607 main PROC
608     push 5                      ;Push the initial value (e.g., 5) on the stack.
609     call Factorial              ;Call the Factorial procedure to calculate the factorial.
610     call WriteDec               ;Display the result (EAX) on the console.
611     call Crlf                  ;Print a new line.
612     exit
613 main ENDP
614 ;-----
615 Factorial PROC
616     ; Calculates a factorial.
617     ; Receives: [ebp+8] = n, the number to calculate
618     ; Returns: eax = the factorial of n
619 ;-----
620     push ebp                  ; Save the current base pointer.
621     mov ebp, esp               ; Set up a new base pointer for the current stack frame.
622     mov eax, [ebp+8]            ; Get the value of n from the stack.
623     cmp eax, 0                 ; Check if n is zero.
624     ja L1                     ; If n is greater than zero, continue; otherwise, go to L2.
625     mov eax, 1                 ; If n is zero, return 1 as the value of 0!
626     jmp L2                     ; Jump to the point where we clean up the stack and return.
627 L1:
628     dec eax                  ; Decrement n.
629     push eax                  ; Push the decremented value onto the stack.
630     call Factorial             ; Recursively call the Factorial procedure with n-1.
631 L2:
632     pop ebp                  ; Clean up the stack by restoring the previous base pointer.
633     ret                       ; Return with the result (EAX).
634 Factorial ENDP
635 END main
```

## Here's an in-depth explanation:

The main procedure begins by pushing the initial value (5 in this case) onto the stack and then calls the Factorial procedure to calculate the factorial.

The Factorial procedure is a recursive function for calculating the factorial of an integer. It first saves the current base pointer on the stack and sets up a new base pointer for the current stack frame.

It retrieves the value of n from the stack (passed as a parameter) into the eax register.

It compares n to 0 using the cmp instruction. If n is greater than 0 (ja - jump above), it proceeds to L1; otherwise, it jumps to L2.

In L1, it decrements n and pushes the new value onto the stack. Then, it makes a recursive call to the Factorial procedure with n-1.

In L2, it pops the base pointer from the stack to clean up the stack frame and returns with the result in EAX.

This recursive approach continues to reduce n until it reaches 0, accumulating the product of each multiplication in EAX.

The result is then returned and displayed in the main procedure.

The program calculates factorials using recursion, and the result for the provided input of 5 would be 120.

## *Tip:*

It is important to keep track of which registers are modified when making recursive calls to a procedure, so that you can save and restore their values if necessary. This is especially important if the register values are needed across recursive procedure calls.

---

1. (True/False): Given the same task to accomplish, a recursive subroutine usually uses more memory than a nonrecursive one. False: Recursive subroutines typically use more memory than nonrecursive subroutines, because they require additional stack space to store the return addresses of the recursive calls.
2. In the Factorial function, what condition terminates the recursion? The recursion terminates when the input parameter, n, is equal to 0.
3. Which instructions in the assembly language Factorial procedure execute after each recursive call has finished? The following instructions in the assembly language Factorial procedure execute after each recursive call has finished:

```
669 mov ebx, [ebp+8]  
670 mul ebx
```

These instructions multiply the result of the recursive call by n. This is necessary because the factorial of n is equal to n multiplied by the factorial of n - 1.

What would happen to the Factorial program's output if you tried to calculate 13!? The Factorial program would fail to calculate 13! because the factorial of 13 is too large to be represented in a 32-bit integer.

**Challenge:** How many bytes of stack space would be used by the Factorial procedure when calculating  $5!$ ? The Factorial procedure would use 20 bytes of stack space when calculating  $5!$ . This is because the stack frame for each recursive call requires 4 bytes for the return address and 16 bytes for the local variables.

Here is a breakdown of the stack space requirements:

676 Return address: 4 bytes

677 Local variables: 16 bytes

678 Total: 20 bytes

The Factorial procedure makes 5 recursive calls when calculating  $5!$ , so the total stack space requirement is 20 bytes per recursive call \* 5 recursive calls = 100 bytes.

## ***INVOKE, ADDR, PROC, and PROTO***

The INVOKE, PROC, and PROTO directives provide powerful tools for defining and calling procedures in 32-bit mode.

They are more convenient to use than the traditional CALL and PROC directives, but they mask the underlying structure of the runtime stack.

In such cases, the PROTO directive helps the assembler to validate procedure calls by checking argument lists against procedure declarations. This can help to prevent errors and make programs more robust.

Advanced procedure directives are more convenient to use than traditional CALL and PROC directives, but they mask the underlying structure of the runtime stack.

It is important to develop a detailed understanding of the low-level mechanics involved in subroutine calls before using advanced procedure directives.

Advanced procedure directives can be used to improve program readability and maintainability, especially when programs execute procedure calls across module boundaries.

The PROTO directive helps the assembler to validate procedure calls by checking argument lists against procedure declarations. Recommendation:

If you are new to assembly language, it is recommended that you start by learning the traditional CALL and PROC directives.

Once you have a good understanding of how subroutine calls work, you can then consider using advanced procedure directives to improve your code.

=====

### ***INVOKE Directive***

=====

The INVOKE directive is a powerful tool for calling procedures in 32-bit mode. It allows you to pass multiple arguments to a procedure using a single line of code.

The general syntax of the INVOKE directive is as follows:

```
INVOKE procedureName [, argumentList]
```

procedureName is the name of the procedure to be called.

argumentList is an optional comma-delimited list of arguments passed to the procedure.

### Arguments to INVOKE

Arguments to INVOKE can be any valid expression, including:

- Immediate values (e.g., 10, 3000h, OFFSET myList).
- Integer expressions (e.g., (1020), COUNT).
- Variables (e.g., myList, array, myWord, myDword).
- Address expressions (e.g., [myList + 2], [ebx+ esi]).
- Registers (e.g., eax, bl, edi).

Arguments to INVOKE are pushed onto the stack in the reverse order that they are specified in the INVOKE statement.

The following example shows how to use the INVOKE directive to call a procedure named DumpArray():

```
INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

This statement will push the following values onto the stack:  
The address of the array  
The length of the array  
The size of the array elements  
The DumpArray() procedure will then be called with these arguments.

This statement is equivalent to the following code using the CALL instruction:

```
692 push TYPE array
693 push LENGTHOF array
694 push OFFSET array
695 call DumpArray
```

The INVOKE directive can handle almost any number of arguments, and individual arguments can appear on separate source code lines. This can be useful for documenting complex INVOKE statements or for breaking up long argument lists.

The following example shows an INVOKE statement with arguments on separate source code lines:

```
700  INVOKE DumpArray,  
701  ; displays an array  
702  ; points to the array  
703  OFFSET array,  
704  ; the array length  
705  LENGTHOF array,  
706  ; array component size  
707  TYPE array
```

Which form you choose is a matter of personal preference. Some programmers prefer to document their code extensively, while others prefer to keep their code as concise as possible.

### **Important Considerations**

When passing arguments to INVOKE that are smaller than 32 bits, the assembler may overwrite the EAX and EDX registers when it widens the arguments before pushing them onto the stack.

To avoid this behavior, you can either pass 32-bit arguments to INVOKE or save and restore the EAX and EDX registers before and after the procedure call.

The INVOKE directive is only available in 32-bit mode.

The INVOKE directive is a powerful tool for calling procedures in 32-bit mode.

It allows you to pass multiple arguments to a procedure using a single line of code.

However, it is important to be aware of the potential for overwriting the EAX and EDX registers when

passing small arguments to INVOKE.

=====

## ***ADDR Operator***

=====

The ADDR operator is a powerful tool for passing pointer arguments to procedures using INVOKE. It is only available in 32-bit mode.

The ADDR operator takes a single operand, which must be an assembly time constant. This means that the operand must be known at compile time, and cannot be a variable or expression that is evaluated at runtime.

The ADDR operator returns the address of the operand. This address can then be passed to a procedure using INVOKE.

The following example shows how to use the ADDR operator to pass the address of an array to a procedure named FillArray():

**INVOKE FillArray, ADDR myArray**

This statement is equivalent to the following code:

```
715 mov esi, myArray  
716 INVOKE FillArray, esi
```

However, the first form is more concise and readable.

The ADDR operator can only be used with the INVOKE directive.

It is not valid to use the ADDR operator with other instructions, such as MOV or CALL.

The ADDR operator can only be used to pass the address of an assembly time constant.

It is not valid to pass the address of a variable or expression that is evaluated at runtime.

The following code shows how to use the ADDR operator to call a procedure named Swap() and pass it the addresses of the first two elements in an array of doublewords:

```
722 .data  
723     Array DWORD 20 DUP(?)  
724 .code  
725     ...  
726     INVOKE Swap,  
727         ADDR Array,  
728         ADDR [Array+4]
```

The assembler will generate the following code:

```
734 push  
735 OFFSET Array+4  
736 push  
737 OFFSET Array  
738 call  
739 Swap
```

The ADDR operator is a powerful tool for passing pointer arguments to procedures using INVOKE. It allows you to write more concise and readable code.

The ADDR operator can also be used to pass the address of a function to another function. This can be useful for implementing callback functions.

For example, the following code shows how to define a function named PrintArray() that prints the elements of an array to the console:

```
745 .code  
746 PrintArray PROC Near  
747 ...  
748 ENDP
```

The following code shows how to pass the address of the PrintArray() function to a function named DoSomething():

```
753 .code  
754     DoSomething PROC Near  
755    (INVOKE PrintArray, ADDR PrintArray)  
756     ENDP
```

When the DoSomething() function is called, it will call the PrintArray() function to print the elements of an array to the console.

=====

### ***PROC Operator***

=====

The PROC directive is used to define a procedure in 32-bit mode. It has the following syntax:

```
label PROC [attributes] [USES reglist], parameter_list
```

- **label** is a user-defined label that follows the rules for identifiers.
- **attributes** is a list of optional attributes that can be used to control the behavior of the procedure. These attributes are:
  - **distance**: Specifies whether the procedure is near or far.
  - **langtype**: Specifies the calling convention (parameter passing convention) to use for the procedure.
  - **visibility**: Specifies the visibility of the procedure to other modules.

- **prologuearg**: Specifies arguments affecting generation of prologue and epilogue code.
- **parameter\_list** is a list of optional parameters that can be passed to the procedure.

## Parameters

Parameters can be of any type, including bytes, words, doublewords, floating-point numbers, and pointers. To declare a parameter, you use the following syntax:

**paramName:type**

- **paramName** is the name of the parameter.
- **type** is the type of the parameter.

For example, the following procedure declares two parameters, **val1** and **val2**, both of which are doublewords:

```
767 AddTwo PROC,  
768 val1:DWORD,  
769 val2:DWORD
```

## USES

The USES clause is an optional clause that can be used to specify which registers the procedure will need to use. This can be useful for optimizing the procedure's code.

For example, the following procedure declares that it will need to use the EAX and EBX registers:

```
Read_File PROC USES eax ebx,
```

The following example shows a simple procedure named AddTwo():

```
779 AddTwo PROC  
780     val1: DWORD    ; Define a DWORD parameter named val1.  
781     val2: DWORD    ; Define another DWORD parameter named val2.  
782     mov eax, val1 ; Move the value of val1 into the EAX register.  
783     add eax, val2 ; Add the value of val2 to EAX.  
784     ret            ; Return from the procedure, effectively returning the result in EAX.  
785 AddTwo ENDP
```

This procedure takes two doublewords as parameters and returns their sum.

The following shows the assembly code generated by MASM when assembling the AddTwo() procedure:

```
794 AddTwo PROC
795     push ebp          ; Save the current base pointer (BP).
796     mov ebp, esp      ; Set up a new base pointer, making ESP the stack frame pointer.
797
798     mov eax, dword ptr [ebp+8]  ; Load the first parameter from the stack into EAX.
799     add eax, dword ptr [ebp+0Ch] ; Add the second parameter from the stack to EAX.
800
801     leave             ; Release the current stack frame.
802     ret               ; Return from the procedure, effectively returning the result in EAX.
803
804     8                ; Indication of the number of bytes used by parameters. (Not part of the actual code.)
805
806 AddTwo ENDP
```

The first two lines of the generated code push the EBP register onto the stack and move the stack pointer to EBP. This is done to create a stack frame for the procedure.

The next two lines move the parameters from the stack to the EAX and EDX registers.

The next line adds the two parameters together in the EAX register.

The next two lines restore the EBP register and return from the procedure.

The constant at the end of the procedure is the size of the procedure's stack frame. This value is used by the RET instruction to pop the correct number of bytes off the stack when returning from the procedure.

The PROC directive is a powerful tool for defining procedures in 32-bit mode. It allows you to create procedures with named parameters and to control the behavior of the procedure's stack frame.

## Here's a detailed explanation of the code:

**push ebp:** This instruction saves the current base pointer (BP) by pushing it onto the stack. This is a common practice to establish a proper stack frame for the procedure.

**mov ebp, esp:** The ebp register is set to the current value of esp, establishing a new stack frame for this procedure. This step aligns the base pointer with the current top of the stack (ESP) and makes it easier to access function parameters and local variables.

**mov eax, dword ptr [ebp+8]:** This line loads the first parameter (at offset +8 from the base pointer) from the stack into the EAX register. The [ebp+8] notation indicates that the first parameter is located 8 bytes above the base pointer.

**add eax, dword ptr [ebp+0Ch]:** Here, the code loads the second parameter (at offset +12 from the base pointer) from the stack and adds it to the value in EAX.

**leave:** This instruction is often used to clean up the stack frame. It's the opposite of the enter instruction. It effectively performs the following operations:

**Restores the previous value of the base pointer (EBP) from the stack.** Adjusts the stack pointer (ESP) to remove the local variables and parameters of the current function. Essentially, it unwinds the stack frame to the previous state.  
**ret:** This instruction returns from the procedure, and the value in EAX becomes the return value of the function.

The **8 at the end** is likely a comment indicating that the parameters take up 8 bytes in total (4 bytes each), which is common for two 32-bit integers.

**In summary,** the AddTwo procedure adds two 32-bit integers passed as parameters, and the result is

returned in the EAX register. The use of the base pointer (EBP) simplifies parameter access within the stack frame.

=====

### ***Specifying the Parameter Passing Protocol:***

=====

The parameter passing protocol specifies how parameters are passed to and from procedures. There are different parameter passing protocols, such as C, Pascal, and STDCALL.

To specify the parameter passing protocol for a procedure in assembly language, you can use the attributes field of the PROC directive.

For example, the following procedure declares that it uses the C calling convention:

```
824 Example1 PROC C,  
825    parm1:DWORD, parm2:DWORD
```

If you execute Example1() using the INVOKE directive, the assembler will generate code that is consistent with the C calling convention.

Similarly, the following procedure declares that it uses the STDCALL calling convention:

```
835 Example1 PROC STDCALL,  
836    parm1:DWORD, parm2:DWORD
```

If you execute Example1() using the INVOKE directive, the assembler will generate code that is consistent with the STDCALL calling convention.

The following example shows how to use the PROC directive to declare a procedure with a specific parameter passing protocol:

```
835 Example1 PROC STDCALL,  
836    parm1:DWORD, parm2:DWORD  
837  
838  
839  
840  
841 .MODEL FLAT,STDCALL  
842  
843 ; Declare a procedure with the C calling convention.  
844 Example1 PROC C,  
845    parm1:DWORD, parm2:DWORD  
846  
847 ; ...  
848  
849 Example1 ENDP  
850  
851 ; Declare a procedure with the STDCALL calling convention.  
852 Example2 PROC STDCALL,  
853    parm1:DWORD, parm2:DWORD  
854  
855 ; ...  
856  
857 Example2 ENDP
```

The ability to specify the parameter passing protocol for a procedure is a powerful feature that

allows you to write assembly language code that can be called from other programming languages.

=====

## ***PROTO Directive in 32-bit Mode***

=====

In 32-bit mode, the **PROTO directive** is used to create a prototype for an existing procedure.

A prototype declares a procedure's name and parameter list.

It allows you to call a procedure before defining it and to verify that the number and types of arguments match the procedure definition.

The syntax of the PROTO directive is as follows:

```
label PROTO [attributes] [parameter_list]
```

- **label** is the name of the procedure.
- **attributes** is an optional field that can be used to specify the parameter passing protocol for the procedure.
- **parameter\_list** is an optional list of parameters that the procedure takes. Example

The following example shows how to create a prototype for a procedure named `ArraySum()`:

```
866 ArraySum PROTO,  
867 ptrArray:PTR DWORD,  
868 ; points to the array  
869 szArray:DWORD  
870 ; array size
```

This prototype declares that the `ArraySum()` procedure takes two parameters: a pointer to an array of doublewords and the size of the array.

Once you have created a prototype for a procedure, you can call it using the `INVOKE` directive.

The `INVOKE` directive will verify that the number and types of arguments match the prototype before calling the procedure.

For example, the following code calls the `ArraySum()` procedure:

```
INVOKE ArraySum, ptrArray, szArray
```

This code will call the `ArraySum()` procedure with the pointer to the array `ptrArray` and the size of the array `szArray` as arguments.

### **Important Considerations:**

Every procedure called by the `INVOKE` directive must have a prototype.

The prototype for a procedure must appear before the procedure is called.

The number and types of arguments in the prototype must match the number and types of arguments in the procedure definition.

The PROTO directive is a powerful tool for writing reusable and reliable assembly language code.

It allows you to call procedures before defining them and to verify that the number and types of arguments match the procedure definition.

## ***Assembly Time Argument Checking***

The PROTO directive helps the assembler check the number and types of arguments passed to a procedure when it is called. This is called assembly time argument checking.

However, assembly time argument checking is not as precise as you would find in languages like C and C++. MASM only checks for the correct number of parameters and to a limited extent, matches argument types to parameter types.

Suppose the following prototype is declared for a procedure named Sub1():

```
879 Sub1 PROTO,  
880 p1:BYTE,  
881 p2:WORD,  
882 p3:PTR BYTE
```

This prototype declares that the Sub1() procedure takes three parameters: a byte, a word, and a pointer to a byte.

The following is a valid call to Sub1():

```
INVOKE Sub1, byte_1, word_1, ADDR byte_1
```

The assembler will generate the following code for this INVOKE statement:

```
890 push 404000h ; Push the pointer to byte_1 onto the stack.  
891 sub esp, 2 ; Reserve 2 bytes on the stack for padding.  
892 push word ptr ds:[00404001h] ; Push the value of word_1 onto the stack.  
893 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.  
894 push eax ; Push the value from EAX onto the stack.  
895 call 00401071 ; Call the function at address 00401071.
```

The assembler pads the stack with two bytes because the second argument (word\_1) is a word, which is two bytes long.

### Errors Detected by MASM

MASM will generate an error if an argument exceeds the size of a declared parameter. For example, the following INVOKE statement will generate an error:

```
908 INVOKE Sub1, word_1, word_2, ADDR byte_1  
909 ;arg 1 error
```

MASM will also generate errors if an INVOKE statement has too few or too many arguments. For example, the following INVOKE statements will generate errors:

```
913 INVOKE Sub1, byte_1, word_2  
914 ; error: too few arguments  
915 INVOKE Sub1, byte_1,  
916 ; error: too many arguments  
917 word_2, ADDR byte_1, word_2
```

### Errors Not Detected by MASM

MASM will not detect an error if an argument's type is smaller than a declared parameter. For example, the following INVOKE statement will not generate an error:

```
(INVOKE Sub1, byte_1, byte_1, ADDR byte_1)
```

Instead, MASM will expand the smaller argument (byte\_1) to the size of the declared parameter (WORD).

In the following code generated by MASM, the second argument (byte\_1) is expanded into EAX before pushing it on the stack:

```
925 push 404000h ; Push the address of byte_1 onto the stack.  
926 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.  
927 movzx eax, al ; Expand the value in AL into EAX.  
928 push eax ; Push the value from EAX onto the stack.  
929 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.  
930 push eax ; Push the value from EAX onto the stack.  
931 call 00401071 ; Call the function at address 00401071 (Assuming it's a function).
```

### Here's a more detailed explanation:

**push 404000h:** This instruction pushes the pointer to byte\_1 onto the stack. It's pushing an address to the stack, which may be used as a parameter for the function you're calling (at address 00401071).

**sub esp, 2:** This instruction subtracts 2 from the stack pointer (esp). It's used to reserve 2 bytes on the stack for padding. This padding might be needed to align the stack correctly, especially when dealing with functions or system calls that expect specific stack alignment.

**push word ptr ds:[00404001h]:** Here, the code pushes the value of word\_1 onto the stack. It's assumed that word\_1 is a 16-bit (2-byte) value. The word ptr specifies that you are dealing with a word-sized value, and it's loaded from memory address 00404001h.

**mov al, byte ptr ds:[00404000h]:** This instruction loads the value of byte\_1 into the AL register. It's assumed that byte\_1 is an 8-bit (1-byte) value, and it's loaded from memory address 00404000h.

**push eax:** The value from the EAX register is pushed onto the stack. This is likely done to make it available as a parameter for the function being called at address 00401071.

**call 00401071:** This instruction calls a function located at address 00401071. The behavior of this function depends on its implementation and the purpose it serves within your program.

Overall, this code appears to be setting up some parameters on the stack and then calling a function at address 00401071, passing these parameters. The specifics of how these parameters are used and the purpose of the function being called would require more context to fully understand.

=====

## *ArraySum*

=====

```
952 ; ArraySum Procedure
953 ; Parameters:
954 ;    esi: Points to the array
955 ;    ecx: Size of the array
956 ; Returns:
957 ;    eax: The sum of the array
958 ArraySum PROC USES esi ecx,
959     ptrArray: PTR DWORD, ; Pointer to the array
960     szArray: DWORD        ; Array size
961
962     mov esi, ptrArray ; Load the address of the array into esi.
963     mov ecx, szArray ; Load the size of the array into ecx.
964     mov eax, 0         ; Initialize the sum to zero.
965
966     cmp ecx, 0         ; Check if the array size is zero.
967     je L2              ; If yes, quit.
968
969 L1:
970     add eax, [esi]      ; Add the value at esi to the sum in eax.
971     add esi, 4          ; Move to the next integer in the array (4 bytes forward).
972     loop L1             ; Repeat for the remaining array size.
973
974 L2:
975     ret                ; Return with the sum in EAX.
976
977 ArraySum ENDP
```

The ArraySum() procedure takes two parameters: a pointer to an array of doublewords and the size of the array. The procedure uses the ESI and ECX registers to store the address of the array and the size of the array, respectively.

The procedure begins by setting the EAX register to zero. This will be the sum of the array elements. Then, the procedure checks the size of the array. If the size is zero, the procedure simply returns. Otherwise, the procedure enters a loop.

In the loop, the procedure adds the value at the current address in the array to the EAX register. Then, the procedure increments the ESI register to point to the next element in the array. The loop repeats until all of the elements in the array have been added.

After the loop has finished, the sum of the array elements is stored in the EAX register. The procedure then returns.

Here is an example of how to call the ArraySum() procedure:

```
0983 .data
0984     array DWORD 10000h, 20000h, 30000h, 40000h, 50000h
0985     theSum DWORD ?
0986
0987 .code
0988 main PROC
0989     INVOKE ArraySum, ADDR array, LENGTHOF array
0990     ; Call the ArraySum procedure, passing the address of the array and the number of elements.
0991
0992     mov theSum, eax
0993     ; Store the sum returned by ArraySum in theSum.
0994
0995     ; Your program logic can continue here, using the calculated sum.
0996
0997 main ENDP
```

The INVOKE statement calls the ArraySum() procedure with the address of the array variable and the number of elements in the array variable as arguments.

The LENGTHOF operator is used to calculate the number of elements in the array variable.

After the ArraySum() procedure has returned, the sum of the array elements is stored in the theSum variable.

The ArraySum() example is a good example of how to use the PROC directive to declare stack parameters and how to use the INVOKE directive to call procedures with stack parameters.

```
1000 .data
1001     array DWORD 10000h, 20000h, 30000h, 40000h, 50000h
1002     theSum DWORD ?
1003
1004 .code
1005     ; ArraySum Procedure
1006     ArraySum PROC USES esi ecx,
1007         ptrArray: PTR DWORD, ; Pointer to the array
1008         szArray: DWORD        ; Array size
1009         mov esi, ptrArray    ; Load the address of the array into esi.
1010         mov ecx, szArray      ; Load the size of the array into ecx.
1011         mov eax, 0            ; Initialize the sum to zero.
1012         cmp ecx, 0            ; Check if the array size is zero.
1013         je L2                 ; If yes, quit.
1014
1015     L1:
1016         add eax, [esi]       ; Add the value at esi to the sum in eax.
1017         add esi, 4           ; Move to the next integer in the array (4 bytes forward).
1018         loop L1               ; Repeat for the remaining array size.
1019
1020     L2:
1021         ret                  ; Return with the sum in EAX.
1022     ArraySum ENDP
1023     main PROC
1024         INVOKE ArraySum, ADDR array, LENGTHOF array
1025         ; Call the ArraySum procedure, passing the address of the array and the number of elements.
1026         mov theSum, eax
1027         ; Store the sum returned by ArraySum in theSum.
1028         ; Your program logic can continue here, using the calculated sum.
1029     main ENDP
```

*In the .data section:*

- An array named `array` is defined with five DWORD (32-bit) elements and initial values.
- A DWORD variable named `theSum` is declared with a question mark to indicate that it's uninitialized.

### ***In the .code section:***

- The `ArraySum` procedure is defined to calculate the sum of an array of DWORDs. It expects two parameters:
- `ptrArray`: A pointer to the array.
- `szArray`: The size (number of elements) of the array. Inside `ArraySum`:
- `esi` is used to hold the address of the array.
- `ecx` stores the size of the array.
- `eax` is initialized to zero and used to accumulate the sum.
- The code checks if the array size is zero. If it is, it immediately jumps to L2, effectively quitting the procedure.
- In L1, it adds the value at the address pointed by `esi` to the sum in `eax`, increments `esi` by 4 to move to the next DWORD in the array, and repeats this process for the entire array size using the `loop` instruction.
- Finally, in L2, it returns with the sum stored in `eax`.

### ***The main procedure:***

- Calls the `ArraySum` procedure using the `INVOKE` directive and passes the address of the array and the number of elements (`LENGTHOF array`) as parameters.
- It stores the result (the sum) returned by `ArraySum` in the `theSum` variable.
- After this code, your program logic can continue, making use of the calculated sum stored in `theSum`.
- This code efficiently calculates the sum of the elements in the array and stores it in `theSum`.

## Parameter Classifications:

In the context of procedure parameters, these parameters can be classified based on the direction of data transfer between the calling program and the called procedure:

Here is a simpler explanation of input and output parameters in assembly language:

**Input parameters** are passed to a procedure from the calling program. The procedure can use the data, but it cannot change it. This means that when the procedure returns, the data in the calling program will be the same as it was before the procedure was called. Input parameters are typically used when the procedure needs data to operate on, but does not need to return any data.

**Output parameters** are used to return data from a procedure to the calling program. The procedure can change the data in the output parameter, and the calling program will see the change after the procedure returns. Output parameters are typically used when the procedure needs to return data to the calling program, such as the result of a calculation.

Here is an example of an input parameter:

```
1033 .data
1034     buffer BYTE 80 DUP(?)
1035     inputHandle DWORD ?
1036 .code
1037     INVOKE ReadConsole, inputHandle, ADDR buffer
1038 ; ReadConsole is expected to store user input in the 'buffer' variable.
```

and

```
1042 procedure add_two_numbers(x: DWORD, y: DWORD): DWORD
1043     ; ...
1044     add eax, x
1045     add eax, y
1046     ret
1047 endp
1048
1049 ; Calling the procedure
1050 mov eax, 10
1051 mov ebx, 20
1052 call add_two_numbers
1053 mov ecx, eax ; ecx will now contain the value 30
```

In this example, the x and y parameters are input parameters. The procedure `add_two_numbers()` uses the data in these parameters to calculate the sum of the two numbers. However, the procedure does not change the values of x and y.

Here is an example of an output parameter:

```
1057 procedure get_system_time(time: PTR DWORD)
1058     ; ...
1059     mov [time], eax
1060     ret
1061 endp
1062
1063 ; Calling the procedure
1064 mov eax, OFFSET time_variable
1065 call get_system_time
1066
1067 ; The time variable will now contain the system time
```

In this example, the time parameter is an output parameter. The procedure `get_system_time()` uses the pointer in the time parameter to store the system time in the memory location that the pointer points to.

Input and output parameters can be used together in a procedure. For example, a procedure could take an input parameter that specifies the size of an array, and it could use an output parameter to return the sum of the elements in the array.

**One example of an input/output parameter is a buffer.** A buffer is a block of memory that is used to store data temporarily. A procedure might take an input/output parameter of type buffer to read data from a file and then return the data to the calling program.

The procedure could also use the buffer to modify the data and then return the modified data to the calling program.

Here is an example of how to use an input/output parameter in assembly language:

```
1071 procedure read_file(buffer: PTR BYTE, size: DWORD): DWORD
1072     ; ...
1073     ; Read data from the file into the buffer
1074     ; ...
1075     ret
1076 endp
1077
1078 ; Calling the procedure
1079 mov eax, OFFSET buffer
1080 mov ebx, size
1081 call read_file
1082
1083 ; The buffer variable will now contain the data that was read from the file
```

In this example, the buffer parameter is an input/output parameter. The `read_file()` procedure reads data from the file into the buffer.

The `read_file()` procedure also returns the number of bytes that were read from the file. The calling program can use this information to determine how much data is in the buffer.

### **Example: Exchanging Two Integers**

```
1090 include Irvine32.inc
1091
1092 Swap PROTO, pValX:PTR DWORD, pValY:PTR DWORD
1093 ; Exchange the values of two 32-bit integers
1094 ; Returns: nothing
1095 Swap PROC USES eax esi edi,
1096 pValX:PTR DWORD,
1097 ; pointer to first integer
1098 pValY:PTR DWORD
1099 ; pointer to second integer
1100 ; get pointers
1101 mov esi, pValX
1102 mov edi, pValY
1103 ; get first integer
1104 mov eax, [esi]
1105 ; exchange with second
1106 xchg eax, [edi]
1107 ; replace first integer
1108 mov [esi], eax
1109 ; PROC generates RET 8 here
1110 ret
1111 Swap ENDP
```

The Swap procedure takes two input/output parameters: pValX and pValY. These parameters contain the addresses of the two integers that need to be swapped.

The procedure begins by getting the pointers to the two integers.

Then, the procedure gets the value of the first integer and stores it in the EAX register.

Next, the procedure uses the **XCHG instruction** to exchange the values of the EAX register and the second integer.

Finally, the procedure stores the value of the EAX register in the first integer.

The Swap procedure does not return any value, so it simply ends with a RET instruction.

However, the PROC directive generates a RET 8 instruction at the end of the procedure, assuming that the STDCALL calling convention is being used.

The Swap procedure can be called from the main procedure as follows:

```
1117 ; Display the array before the exchange:  
1118 mov esi, OFFSET Array  
1119 mov ecx, 2  
1120 ; count = 2  
1121 mov ebx, TYPE Array  
1122 call  
1123 DumpMem  
1124 ; dump the array values  
1125 INVOKE Swap, ADDR Array, ADDR [Array+4]  
1126 ; Display the array after the exchange:  
1127 call  
1128 DumpMem
```

The INVOKE statement calls the Swap procedure with the addresses of the first two elements of the Array variable as arguments. After the Swap procedure returns, the first two elements of the Array variable will be swapped.

### *Missing information:*

The Swap procedure does not check for errors. For example, if the addresses of the two integers are not valid, the procedure will crash.

The Swap procedure is not optimized for speed. For example, the procedure could use a temporary variable to store the value of the first integer while the second integer is being swapped.

Overall, the Swap procedure is a simple example of how to use input/output parameters in assembly language.

language.

---

---

## *Debugging Tips*

---

---

### Argument Size Mismatch

When passing arguments to a procedure, it is important to make sure that the arguments are the correct size.

For example, if a procedure expects a doubleword pointer, you should pass a doubleword pointer.

If you pass a smaller pointer, such as a word pointer, the procedure will not be able to access the data correctly.

Here is an example of an argument size mismatch:

```
1134 ; Swap procedure from Section 8.4.6
1135 Swap PROC, pValX:PTR DWORD, pValY:PTR DWORD
1136 ...
1137
1138 ; Incorrect call to Swap
1139 INVOKE Swap, ADDR [DoubleArray + 0], ADDR [DoubleArray + 1]
```

The Swap() procedure expects two doubleword pointers. However, the incorrect call to Swap() passes two word pointers.

This will cause the procedure to not be able to access the data correctly.

## Passing the Wrong Type of Pointer

When passing arguments to a procedure, it is also important to make sure that the arguments are the correct type.

For example, if a procedure expects a doubleword pointer, you should pass a doubleword pointer. If you pass a different type of pointer, such as a byte pointer, the procedure will not be able to access the data correctly.

Here is an example of passing the wrong type of pointer:

```
1145 ; Swap procedure from Section 8.4.6
1146 Swap PROC, pValX:PTR DWORD, pValY:PTR DWORD
1147 ...
1148
1149 ; Incorrect call to Swap
1150 INVOKE Swap, ADDR [ByteArray + 0], ADDR [ByteArray + 1]
```

The Swap() procedure expects two doubleword pointers. However, the incorrect call to Swap() passes two byte pointers.

This will cause the procedure to not be able to access the data correctly.

## Passing Immediate Values

You should not pass immediate values to reference parameters.

A **reference parameter** is a parameter that expects a pointer to data.

If you pass an immediate value to a reference parameter, the procedure will not be able to access the data correctly.

Here is an example of passing an immediate value to a reference parameter:

```
1156 ; Sub2 procedure
1157 Sub2 PROC, dataPtr:PTR WORD
1158 mov
1159 esi,dataPtr
1160 ; get the address
1161 mov
1162 WORD PTR [esi],0
1163 ; dereference, assign zero
1164 ret
1165 Sub2 ENDP
1166
1167 ; Incorrect call to Sub2
1168 INVOKE
1169 Sub2, 1000h
```

The Sub2() procedure expects a pointer to a word as its only parameter. However, the incorrect call to Sub2() passes an immediate value. This will cause the procedure to not be able to access the data correctly.

It is important to be careful when passing arguments to procedures in assembly language. If you make a mistake, it can cause the program to crash or produce incorrect results. Be sure to check the documentation for the procedure that you are calling to make sure that you are passing the correct type and number of arguments.

## ***WriteStackFrame Procedure***

Here is a more in-depth explanation of the WriteStackFrame and WriteStackFrameName procedures:

The Irvine32 library contains a useful procedure named WriteStackFrame that displays the contents of the current procedure's stack frame. It shows the procedure's stack parameters, return address, local variables, and saved registers.

```
1198 WriteStackFrame PROTO,  
1199     numParam:DWORD,  
1200     ; number of passed parameters  
1201     numLocalVal: DWORD,  
1202     ; number of DWordLocal variables  
1203     numSavedReg: DWORD  
1204     ; number of saved registers
```

The `WriteStackFrame` procedure displays the contents of the current stack frame, which contains the stack parameters, local variables, saved registers, and return address for the current procedure.

It takes 3 parameters:

- `numParam` - The number of parameters passed to the current procedure. This determines how many DWORDs to show for the parameters at the top of the stack.
- `numLocalVal` - The number of DWORD local variables allocated on the stack for the current procedure.
- `numSavedReg` - The number of registers saved on the stack for the current procedure. Typically this is 2 for EAX and EBX.

It displays the stack contents by starting at EBP and moving downward to ESP. For each DWORD it displays the offset from EBP and the hex value stored there.

The parameters passed to the procedure are displayed first at the highest offsets from EBP. Then the return address, saved EBP, local variables, and saved registers are displayed in descending offset order.

ESP points to the last used stack location, so the display stops when it reaches ESP.

`WriteStackFrameName` does the same thing, but takes an additional parameter:

- `procName` - A pointer to a null-terminated string containing the name of the current procedure. This is displayed at the top of the output.

So `WriteStackFrameName` allows you to identify which procedure's stack frame is being displayed. This is useful when multiple procedures call `WriteStackFrame/Name`.

In summary, these procedures give visibility into the stack contents at any point within a procedure. This helps debug issues with stack parameters, local variables, register saving, etc.

Here is an explanation of the MASM code example that was shown in the original text:

```
1173 ; In main procedure
1174 main PROC
1175     mov eax, 0EAEEAEAEAh    ; Save test value in EAX
1176     mov ebx, 0EBEBEBEBh    ; Save test value in EBX
1177     INVOKE myProc, 1111h, 2222h ; Call myProc, passing 2 parameters
1178     exit main              ; Exit program
1179
1180 main ENDP
1181 ; In myProc procedure
1182 myProc PROC
1183     ; Procedure uses EAX and EBX, so they will be saved
1184     USES eax ebx
1185     x: DWORD, y:DWORD      ; Declare parameter variables
1186     LOCAL a:DWORD, b:DWORD ; Declare local variables
1187     PARAMS = 2             ; 2 parameters
1188     LOCALS = 2              ; 2 local DWORD variables
1189     SAVED_REGS = 2          ; 2 saved registers (EAX and EBX)
1190     mov a,0AAAAAh          ; Load value into local variable a
1191     mov b,0BBBBBh          ; Load value into local variable b
1192     ; Display stack frame contents
1193     INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
1194 myProc ENDP
```

The following sample output was produced by the call:

```
1208 Stack Frame
1209 00002222 ebp+12 (parameter)
1210 00001111 ebp+8 (parameter)
1211 00401083 ebp+4 (return address)
1212 0012FFF0 ebp+0 (saved ebp) <--- ebp
1213 0000AAAA ebp-4 (local variable)
1214 0000BBBB ebp-8 (local variable)
1215 EAEAEAEA ebp-12 (saved register)
1216 EBEBEBEB ebp-16 (saved register) <--- esp
```

A second procedure, named WriteStackFrameName, has an additional parameter that holds the name of the procedure owning the stack frame:

```
1221 WriteStackFrameName PROTO,
1222     numParam:DWORD,
1223     ; number of passed parameters
1224     numLocalVal:DWORD,
1225     ; number of DWORD local variables
1226     numSavedReg:DWORD,
1227     ; number of saved registers
1228     procName:PTR BYTE
1229     ; null-terminated string
```

The main procedure:

- Loads some sample values into EAX and EBX to be saved on the stack later.
- Calls the myProc procedure, passing two DWORD arguments (1111h and 2222h).
- Exits the program.

The myProc procedure:

- Uses EAX, EBX registers so they will be saved on the stack.
- Declares x and y parameters and a and b local variables.
- Loads sample values into the local variables.
- Calls WriteStackFrame, passing:
  - 2 for the number of parameters
  - 2 for the number of local DWORD variables
  - 2 for the number of saved registers (EAX and EBX)

This displays the contents of myProc's stack frame, including:

- The 1111h and 2222h parameters
- The return address back to main
- The saved EBP from main
- The local variables a and b
- The saved EAX and EBX registers from main

So this demonstrates how WriteStackFrame can display a procedure's stack contents to help understand and debug the stack usage.

You can find the source code for the Irvine32 library in the \Examples\Lib32 directory of our book's install directory (usually C:\Irvine). Look for the file named Irvine32.asm.

## **Multimodule Programs**

A large program can be divided into multiple modules (assembled units) to make it easier to manage and assemble. Each module is assembled independently, so a change to one module's source code only requires reassembling the single module. The linker combines all assembled modules (OBJ files) into a single executable file.

**There are two general approaches to creating multimodule programs:**

### **Traditional approach:**

Using the EXTERN directive to declare external procedures and the PUBLIC directive to export procedures to other modules. Microsoft's advanced INVOKE and PROTO directives: simplify procedure calls and hide some low-level details.

### **Hiding and Exporting Procedure Names**

By default, MASM makes all procedures public, permitting them to be called from any other module in the same program.

You can override this behavior using the PRIVATE qualifier or the OPTION PROC:PRIVATE directive.

**PRIVATE qualifier:** makes a single procedure private.

**OPTION PROC:PRIVATE directive:** makes all procedures in a module private by default.

You can use the PUBLIC directive to export any procedures you want.

If you use OPTION PROC:PRIVATE in your program's startup module, be sure to designate your startup procedure (usually main) as PUBLIC, or the operating system's loader will not be able to find it.

The following example shows how to create a multimodule program using the traditional approach:

```
1235 ; Module 1: mod1.asm
1236
1237 myProc PROC PUBLIC
1238 ; ...
1239
1240 myProc ENDP
1241
1242 ; Module 2: mod2.asm
1243
1244 EXTERN myProc
1245
1246 main PROC
1247 ; ...
1248
1249 INVOKE myProc
1250
1251 main ENDP
```

To assemble the program, you would use the following commands:

This would create an executable file called myprog.exe.

The EXTERN directive tells the assembler that the myProc() procedure is defined in another module.

The PUBLIC directive tells the assembler that the myProc() procedure can be called from other modules.

## Using the INVOKE and PROTO Directives

The following example shows how to create a multimodule program using Microsoft's advanced INVOKE and PROTO directives:

```
1257 ; Module 1: mod1.asm
1258
1259 PROTO myProc
1260
1261 myProc PROC PUBLIC
1262 ; ...
1263
1264 myProc ENDP
1265
1266 ; Module 2: mod2.asm
1267
1268 INVOKE myProc
1269
1270 main PROC
1271 ; ...
1272
1273 main ENDP
```

To assemble the program, you would use the following commands:

```
1282 ml /c /obj mod1.asm
1283 ml /c /obj mod2.asm
1284 link mod1.obj mod2.obj /out:myprog.exe
```

This would create an executable file called myprog.exe.

The **/c option** tells MASM to compile the source code but not link it. The **/obj option** tells MASM to generate object files. The **/out: option** tells MASM to generate an executable file with the specified name.

The PROTO directive tells the assembler about the prototype of the myProc() procedure, including its name and the number and types of its arguments.

The INVOKE directive tells the assembler to call the myProc() procedure.

The INVOKE and PROTO directives simplify procedure calls and hide some low-level details, such as the need to push and pop arguments on the stack.

## ***Calling External Procedures***

To call an external procedure in MASM, you use the EXTERN directive.

The EXTERN directive tells the assembler that the procedure is defined in another module and gives the procedure's name and stack frame size.

The following example shows how to call an external procedure named sub1():

```
1297 INCLUDE Irvine32.inc
1298 EXTERN sub1@0:PROC
1299
1300 .code
1301     main PROC
1302         call sub1@0
1303         exit
1304     main ENDP
1305 END main
```

The **@0 suffix** at the end of the procedure name indicates that the procedure **does not have any parameters**.

If the procedure has parameters, you must include the **stack frame size** in the EXTERN directive.

The stack frame size is the total amount of stack space that the procedure uses to store its parameters and local variables.

The following example shows how to call an external procedure named AddTwo(), which has two doubleword parameters:

```
1310 INCLUDE Irvine32.inc
1311 EXTERN AddTwo@8:PROC
1312
1313 .code
1314     main PROC
1315         call AddTwo@8
1316         exit
1317     main ENDP
1318 END main
```

The @8 suffix at the end of the procedure name indicates that the procedure uses 8 bytes of stack space for its parameters.

You can also use the PROTO directive in place of the EXTERN directive. The PROTO directive tells the assembler about the prototype of the procedure, including its name and the number and types of its arguments.

The following example shows how to use the PROTO directive to declare the prototype of the AddTwo() procedure:

```
1323 INCLUDE Irvine32.inc
1324 PROTO AddTwo,
1325 val1:DWORD,
1326 val2:DWORD
1327
1328 .code
1329     main PROC
1330         call AddTwo
1331         exit
1332     main ENDP
1333 END main
```

The PROTO directive tells the assembler that the AddTwo() procedure has two doubleword parameters.

When the assembler sees a call to the AddTwo() procedure, it can check to make sure that the correct number of arguments are passed to the procedure.

The EXTERN and PROTO directives are both used to call external procedures.

The EXTERN directive is simpler to use, but the PROTO directive provides more information to the assembler, which can help to prevent errors.

### ***Which directive should you use?***

If you are only calling a few external procedures and you are not concerned about the performance of your program, then you can use the EXTERN directive.

If you are calling a large number of external procedures or if you are concerned about the performance of your program, then you should use the PROTO directive.

The PROTO directive provides more information to the assembler, which can help to optimize the code.

=====

*Using Variables and Symbols across Module Boundaries:*

*Exporting Variables and Symbols*

=====

**What is EXTERNDEF?**

EXTERNDEF is a directive that combines the functionality of the PUBLIC and EXTERN directives. It can be used to export variables and symbols from one module and import them into another module.

**How to use EXTERNDEF?**

To use EXTERNDEF, you first need to create an include file that contains the EXTERNDEF declarations for the variables and symbols that you want to share. For example, the following include file defines two variables, count and SYM1:

```
1336 ; vars.inc  
1337 EXTERNDEF count:DWORD, SYM1:ABS
```

Then, you can include the include file in any module that needs to access the variables or symbols. For example, the following module exports the count and SYM1 variables:

```
1343 ; sub1.asm  
1344 .386  
1345 .model flat,STDCALL  
1346 INCLUDE vars.inc  
1347 SYM1 = 10  
1348 .data  
1349     count DWORD 0  
1350     END
```

Finally, you can import the variables or symbols into any module that needs to use them. For example, the following module imports the count and SYM1 variables and uses them to calculate a value:

```
1356 ; main.asm
1357 .386
1358 .model flat,stdcall
1359 .stack 4096
1360 ExitProcess proto, dwExitCode:dword
1361 INCLUDE vars.inc
1362 .code
1363     main PROC
1364     mov
1365     count,2000h
1366     mov
1367     eax,SYM1
1368     INVOKE ExitProcess,0
1369     main ENDP
1370 END main
```

## Benefits of using EXTERNDEF

There are several benefits to using EXTERNDEF to share variables and symbols across module boundaries:

It makes it easy to share variables and symbols between modules. It helps to reduce the amount of duplicate code.

It makes the code more modular and reusable. Conclusion

EXTERNDEF is a powerful directive that can be used to share variables and symbols across module boundaries.

It is a good practice to use EXTERNDEF to share variables and symbols between modules, as it makes the code more modular and reusable.

Let's make our program modular and see these concepts in action:

=====

### ***ArraySum program***

=====

The ArraySum program, first presented in Chapters before, is a good example of a multimodule program. The program can be divided into the following modules:

**main.asm**: The startup module, which calls the other modules to perform the program's tasks.

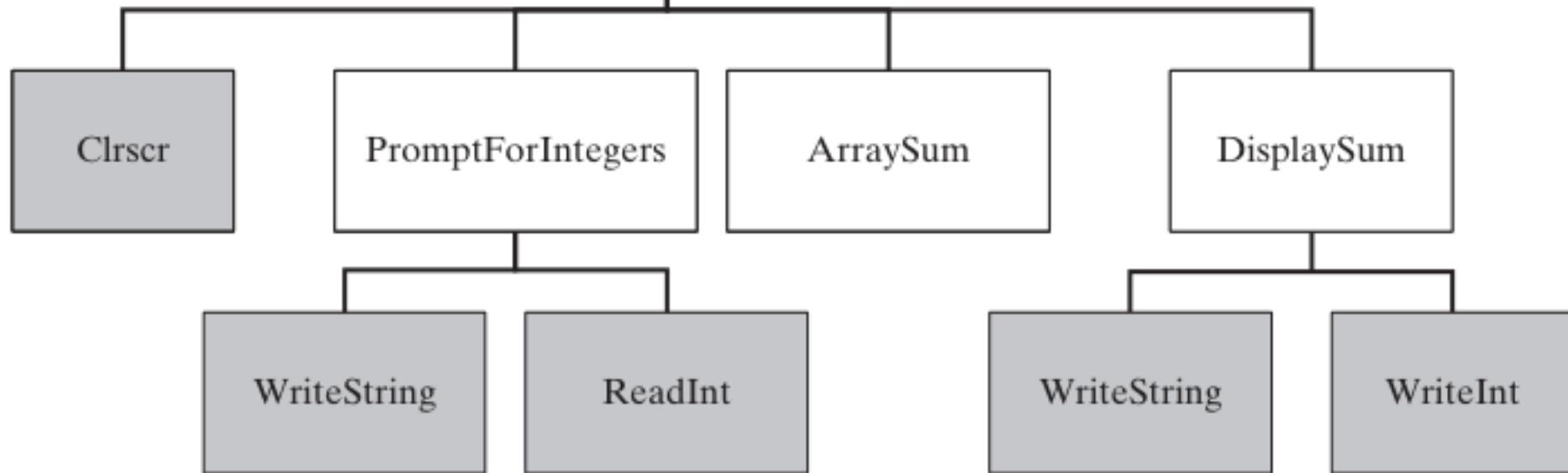
**promptforintegers.asm**: Prompts the user to enter an array of integers and reads the integers from the console.

**arraysum.asm**: Calculates the sum of the integers in the array.

**writeinteger.asm**: Writes an integer to the console.

The following diagram shows the structure chart of the ArraySum program:

ArraySum  
Program (main)



=====  
*Prompt for Integers*  
=====

```
; Prompt For Integers (_prompt.asm)
INCLUDE Irvine32.inc
.code
;-----
PromptForIntegers PROC
; Prompts the user for an array of integers and fills
; the array with the user's input.
; Receives:
; ptrPrompt: PTR BYTE - prompt string
; ptrArray: PTR DWORD - pointer to array
; arraySize: DWORD - size of the array
; Returns: nothing
;-----
arraySize EQU [ebp+16]
ptrArray EQU [ebp+12]
ptrPrompt EQU [ebp+8]
enter 0,0
pushad
; save all registers
mov ecx, arraySize
cmp ecx, 0
; array size != 0?
jle L2
; yes: quit
mov edx, ptrPrompt
; address of the prompt
mov esi, ptrArray
L1:
call WriteString
; display string
call ReadInt
; read integer into EAX
call Crlf
; go to next output line
mov [esi], eax
; store in array
add esi, 4
; next integer
```

```
loop L1
L2:
popad
; restore all registers
leave
ret
12
; restore the stack
PromptForIntegers ENDP
end
```

The prompt.asm file contains the source code for the PromptForIntegers() procedure. This procedure prompts the user for an array of integers and fills the array with the user's input.

- The **PromptForIntegers()** procedure takes three parameters:
- **ptrPrompt**: A pointer to the prompt string.
- **ptrArray**: A pointer to the array.
- **arraySize**: The size of the array.

### **The PromptForIntegers() procedure works as follows:**

It saves all of the registers.

It compares the array size to 0. If the array size is 0, the procedure exits. It displays the prompt string using the WriteString() procedure. It reads an integer from the console using the ReadInt() procedure. It stores the integer in the array.

It increments the array pointer. It repeats steps 3-6 until all of the integers have been read. It restores all of the registers. It leaves the procedure. Here is a more detailed explanation of each step:

### **Step 1: Save all of the registers**

The PromptForIntegers() procedure saves all of the registers because it needs to use them and it does not want to overwrite any of the values that are in the registers when the procedure is called.

### **Step 2: Compare the array size to 0**

The PromptForIntegers() procedure checks the array size to make sure that it is not 0. If the array size is 0, the procedure exits. This is because it does not make sense to prompt the user for an array of integers if the array is empty.

### **Step 3: Display the prompt string**

The PromptForIntegers() procedure displays the prompt string using the WriteString() procedure. The WriteString() procedure is a library procedure that writes a string to the console.

### **Step 4: Read an integer from the console**

The PromptForIntegers() procedure reads an integer from the console using the ReadInt() procedure. The ReadInt() procedure is a library procedure that reads an integer from the console and stores it in the EAX register.

### **Step 5: Store the integer in the array**

The PromptForIntegers() procedure stores the integer in the array by moving the EAX register to the array element that is pointed to by the array pointer.

### **Step 6: Increment the array pointer**

The PromptForIntegers() procedure increments the array pointer so that it points to the next element in the array.

### **Step 7: Repeat steps 3-6 until all of the integers have been read**

The PromptForIntegers() procedure repeats steps 3-6 until all of the integers have been read. This

is done by using the loop instruction. The loop instruction repeats a block of instructions until a specified condition is met. In this case, the condition is that the array pointer is not equal to the end of the array.

### Step 8: Restore all of the registers

The `PromptForIntegers()` procedure restores all of the registers that it saved in step 1.

### Step 9: Leave the procedure

The `PromptForIntegers()` procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The `PromptForIntegers()` procedure is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses library procedures to perform common tasks, such as writing a string to the console and reading an integer from the console.

=====

### ArraySum program

=====

```
; ArraySum Procedure (_arraysum.asm)
INCLUDE Irvine32.inc
.code
;-----  
ArraySum PROC  
;
```

```
; Calculates the sum of an array of 32-bit integers.  
; Receives:  
; ptrArray - pointer to array  
; arraySize - size of array (DWORD)  
; Returns: EAX = sum  
;  
-----  
ptrArray EQU [ebp+8]  
arraySize EQU [ebp+12]  
enter 0,0  
push ecx  
; don't push EAX  
push esi  
mov eax,0  
; set the sum to zero  
mov esi,ptrArray  
mov ecx,arraySize  
cmp ecx,0  
; array size != 0?  
jle L2  
; yes: quit  
L1:  
add eax,[esi]  
; add each integer to sum  
add esi,4  
; point to next integer  
loop L1  
; repeat for array size  
L2:  
pop esi  
pop ecx  
; return sum in EAX  
leave  
ret  
8  
; restore the stack  
ArraySum ENDP  
END
```

The code you provided is the implementation of the `ArraySum()` procedure in assembly language. The `ArraySum()` procedure calculates the sum of an array of 32-bit integers.

***The `ArraySum()` procedure takes two parameters:***

- `ptrArray`: A pointer to the array.
- `arraySize`: The size of the array.

***The `ArraySum()` procedure works as follows:***

It saves the ECX register, because it needs to use it and it does not want to overwrite the value that is in the register when the procedure is called. It sets the EAX register to 0.

This is because the EAX register will be used to store the sum of the integers in the array. It moves the pointer to the first element of the array into the ESI register. It compares the array size to 0.

If the array size is 0, the procedure exits. This is because it does not make sense to calculate the sum of an empty array.

It adds the integer at the current position in the array to the EAX register. It increments the ESI register so that it points to the next element in the array.

It repeats steps 5 and 6 until all of the integers in the array have been added to the EAX register. It restores the ECX register.

It leaves the procedure.

Here is a more detailed explanation of each step:

**Step 1: Save the ECX register**

The ArraySum() procedure saves the ECX register because it needs to use it and it does not want to overwrite the value that is in the register when the procedure is called.

**Step 2: Set the EAX register to 0**

The ArraySum() procedure sets the EAX register to 0 because it will be used to store the sum of the integers in the array.

**Step 3: Move the pointer to the first element of the array into the ESI register**

The ArraySum() procedure moves the pointer to the first element of the array into the ESI register. This is because the ESI register will be used to iterate through the array.

**Step 4: Compare the array size to 0**

The ArraySum() procedure checks the array size to make sure that it is not 0. If the array size is 0, the procedure exits. This is because it does not make sense to calculate the sum of an empty array.

**Step 5: Add the integer at the current position in the array to the EAX register**

The ArraySum() procedure adds the integer at the current position in the array to the EAX register. This is done using the add instruction. The add instruction adds two operands and stores the result in the first operand.

**Step 6: Increment the ESI register so that it points to the next element in the array**

The ArraySum() procedure increments the ESI register so that it points to the next element in the array. This is done using the inc instruction. The inc instruction increments the value of the operand by 1.

## **Step 7: Repeat steps 5 and 6 until all of the integers in the array have been added to the EAX register**

The ArraySum() procedure repeats steps 5 and 6 until all of the integers in the array have been added to the EAX register.

This is done using the loop instruction. The loop instruction repeats a block of instructions until a specified condition is met.

In this case, the condition is that the ESI register is not equal to the value of the ptrArray parameter.

## **Step 8: Restore the ECX register**

The ArraySum() procedure restores the ECX register.

## **Step 9: Leave the procedure**

The ArraySum() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The ArraySum() procedure is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses a loop to iterate through the array, which is a common technique in assembly language.

=====

## *Display Sum Proc*

=====

```
; DisplaySum Procedure (_display.asm)
INCLUDE Irvine32.inc
.code
;-----
DisplaySum PROC
; Displays the sum on the console.
; Receives:
; ptrPrompt - offset of the prompt string
; theSum - the array sum (DWORD)
; Returns: nothing
;-----
theSum EQU [ebp+12]
ptrPrompt EQU [ebp+8]
enter 0,0
push eax
push edx
mov edx,ptrPrompt
; pointer to prompt
call WriteString
mov eax,theSum
call WriteInt
; display EAX
call Crlf
pop edx
pop eax
leave
ret
8
; restore the stack
DisplaySum ENDP
END
```

The code you provided is the implementation of the `DisplaySum()` procedure in assembly language. The `DisplaySum()` procedure displays the sum of an array of 32-bit integers on the console.

The DisplaySum() procedure takes two parameters:

- **ptrPrompt**: A pointer to the prompt string.
- **theSum**: The sum of the integers in the array. The DisplaySum() procedure works as follows:

It saves the EAX and EDX registers, because it needs to use them and it does not want to overwrite the values that are in the registers when the procedure is called. It moves the pointer to the prompt string into the EDX register.

It calls the WriteString() procedure to display the prompt string on the console. It moves the sum of the integers in the array into the EAX register.

It calls the WriteInt() procedure to display the sum of the integers in the array on the console.

It calls the Crlf() procedure to move the cursor to the next line on the console. It restores the EAX and EDX registers.

It leaves the procedure.

### **Here is a more detailed explanation of each step:**

#### **Step 1: Save the EAX and EDX registers**

The DisplaySum() procedure saves the EAX and EDX registers because it needs to use them and it does not want to overwrite the values that are in the registers when the procedure is called.

#### **Step 2: Move the pointer to the prompt string into the EDX register**

The DisplaySum() procedure moves the pointer to the prompt string into the EDX register. This is because the EDX register will be used to pass the pointer to the prompt string to the WriteString()

procedure.

**Step 3: Call the WriteString() procedure to display the prompt string on the console**

The DisplaySum() procedure calls the WriteString() procedure to display the prompt string on the console. The WriteString() procedure is a library procedure that writes a string to the console.

**Step 4: Move the sum of the integers in the array into the EAX register**

The DisplaySum() procedure moves the sum of the integers in the array into the EAX register. This is because the EAX register will be used to pass the sum of the integers in the array to the WriteInt() procedure.

**Step 5: Call the WriteInt() procedure to display the sum of the integers in the array on the console**

The DisplaySum() procedure calls the WriteInt() procedure to display the sum of the integers in the array on the console. The WriteInt() procedure is a library procedure that writes an integer to the console.

**Step 6: Call the Crlf() procedure to move the cursor to the next line on the console**

The DisplaySum() procedure calls the Crlf() procedure to move the cursor to the next line on the console. The Crlf() procedure is a library procedure that moves the cursor to the next line on the console.

**Step 7: Restore the EAX and EDX registers**

The DisplaySum() procedure restores the EAX and EDX registers.

**Step 8: Leave the procedure**

The DisplaySum() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The DisplaySum() procedure is a good example of how to write a procedure in assembly language. The

procedure is well-structured and easy to understand. The procedure also uses library procedures to perform common tasks, such as writing a string to the console and writing an integer to the console.

=====

### *Sum\_main.asm*

=====

```
; Integer Summation Program (Sum_main.asm)
; Multimodule example:
; This program inputs multiple integers from the user,
; stores them in an array, calculates the sum of the
; array, and displays the sum.
INCLUDE Irvine32.inc
INCLUDE macros.asm ; Include for INVOKE and PROTO

EXTERN PromptForIntegers:PROC
EXTERN ArraySum:PROC
EXTERN DisplaySum:PROC

; Modify Count to change the size of the array:
Count = 3

.data
prompt1 BYTE "Enter a signed integer: ",0
prompt2 BYTE "The sum of the integers is: ",0
array DWORD Count DUP(?)
sum DWORD ?
```

```
.code
main PROC
    call Clrscr
    ; PromptForIntegers(ADDR prompt1, ADDR array, Count)
    INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count

    ; sum = ArraySum(ADDR array, Count)
    INVOKE ArraySum, ADDR array, Count
    mov sum, eax

    ; DisplaySum(ADDR prompt2, sum)
    INVOKE DisplaySum, ADDR prompt2, sum
    call Crlf
    exit
main ENDP

END main
```

This code retains the same functionality as the original version but utilizes Microsoft's INVOKE and PROTO directives for calling procedures, making the code more structured and easier to read.

The code you provided is a multimodule example of an integer summation program. The program inputs multiple integers from the user, stores them in an array, calculates the sum of the array, and displays the sum.

### ***The program is divided into three modules:***

**Sum\_main.asm:** This is the main module, which contains the main() procedure. The main() procedure is responsible for calling the other modules to perform the program's tasks.

**Promptforintegers.asm:** This module contains the PromptForIntegers() procedure, which prompts the user for multiple integers and stores them in an array.

user for multiple integers and stores them in an array.

**arraysum.asm**: This module contains the ArraySum() procedure, which calculates the sum of the integers in an array.

**display.asm**: This module contains the DisplaySum() procedure, which displays the sum of the integers in an array on the console.

**The Sum\_main.asm module is the main module of the program. The main() procedure in this module performs the following steps:**

It calls the Clrscr() procedure to clear the console screen. It calls the PromptForIntegers() procedure to prompt the user for multiple integers and store them in an array. It calls the ArraySum() procedure to calculate the sum of the integers in the array.

It calls the DisplaySum() procedure to display the sum of the integers in the array on the console. It calls the Crlf() procedure to move the cursor to the next line on the console. It calls the exit() procedure to exit the program.

The promptforintegers.asm module contains the PromptForIntegers() procedure. This procedure prompts the user for multiple integers and stores them in an array. The procedure takes the following parameters:

- **ptrPrompt**: A pointer to the prompt string.
- **ptrArray**: A pointer to the array.
- **Count**: The number of integers to prompt the user for.

*The PromptForIntegers() procedure works as follows:*

It iterates over the array and prompts the user for each integer. It stores the integer that the user enters in the array.

It repeats steps 1 and 2 until all of the integers have been entered. The `arraysum.asm` module contains the `ArraySum()` procedure. This procedure calculates the sum of the integers in an array. The procedure takes the following parameters:

- `ptrArray`: A pointer to the array.
- `Count`: The number of integers in the array.

*The `ArraySum()` procedure works as follows:*

It initializes the sum to 0.

It iterates over the array and adds each integer to the sum. It returns the sum. The `display.asm` module contains the `DisplaySum()` procedure.

This procedure displays the sum of the integers in an array on the console. The procedure takes the following parameters:

- `ptrPrompt`: A pointer to the prompt string.
- `theSum`: The sum of the integers in the array.

*The `DisplaySum()` procedure works as follows:*

It displays the prompt string on the console. It displays the sum of the integers in the array on the console. It moves the cursor to the next line on the console.

The integer summation program is a good example of how to use multiple modules to write a program.

By dividing the program into modules, we can make the program more modular, reusable, and maintainable.

=====

### *Creating Modules using INVOKE and PROTO*

=====

Creating the Modules Using INVOKE and PROTO section are the use of the INVOKE, PROTO, and PROC directives. These directives are used to create multimodule programs in 32-bit mode.

The INVOKE directive is used to call a procedure in another module. The PROTO directive is used to declare a prototype for a procedure. The PROC directive is used to define a procedure.

The following table shows the differences between the traditional use of CALL and EXTERN and the use of INVOKE, PROTO, and PROC:

## Traditional Method:

Traditional Method	Advanced Method
CALL is used to call a procedure.	INVOKE is used to call a procedure in another module.
EXTERN is used to declare a symbol that is defined in another module.	PROTO is used to declare a prototype for a procedure.
PROC is used to define a procedure.	PROC is used to define a procedure, but it can also be used to declare parameters for a procedure.

The **main advantage of using the INVOKE, PROTO, and PROC directives** is that they allow the assembler to match up the argument lists passed by INVOKE to the corresponding parameter lists declared by PROC. This helps ensure that the program is correct and that it does not crash.

Example Using INVOKE, PROTO, and PROC:

```
1430 ; Include the necessary include file
1431 INCLUDE sum.inc
1432
1433 ; Define data and code sections
1434 .data
1435 Count = 3
1436 prompt1 BYTE "Enter a signed integer: ",0
1437 prompt2 BYTE "The sum of the integers is: ",0
1438 array DWORD Count DUP(?)
1439 sum DWORD ?
1440
1441 .code
1442 main PROC
1443     call Clrscr
1444
1445     ; Call PromptForIntegers using INVOKE with argument lists
1446     INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
1447
1448     ; Call ArraySum using INVOKE with argument lists
1449     INVOKE ArraySum, ADDR array, Count
1450     mov sum, eax
1451
1452     ; Call DisplaySum using INVOKE with argument lists
1453     INVOKE DisplaySum, ADDR prompt2, sum
1454
1455     call Crlf
1456     exit
1457 main ENDP
```

These are all the functions using advanced methods:

```
1479 ; sum.inc
1480 INCLUDE Irvine32.inc
1481
1482 PromptForIntegers PROTO,
1483     ptrPrompt:PTR BYTE,
1484     ptrArray:PTR DWORD,
1485     arraySize:DWORD
1486
1487 ArraySum PROTO,
1488     ptrArray:PTR DWORD,
1489     arraySize:DWORD
1490
1491 DisplaySum PROTO,
1492     ptrPrompt:PTR BYTE,
1493     theSum:DWORD
```

```
1500 ; prompt.asm
1501 INCLUDE sum.inc
1502
1503 .code
1504 PromptForIntegers PROC,
1505     ptrPrompt:PTR BYTE,
1506     ptrArray:PTR DWORD,
1507     arraySize:DWORD
1508
1509     pushad
1510     mov ecx, arraySize
1511     cmp ecx, 0
1512     jle L2
1513     mov edx, ptrPrompt
1514     mov esi, ptrArray
1515 L1:
1516     call WriteString
1517     call ReadInt
1518     call Crlf
1519     mov [esi], eax
1520     add esi, 4
1521     loop L1
1522 L2:
1523     popad
1524     ret
1525 PromptForIntegers ENDP
1526 END
```



```
1530 ; arraysum.asm
1531 INCLUDE sum.inc
1532
1533 .code
1534 ArraySum PROC,
1535     ptrArray:PTR DWORD,
1536     arraySize:DWORD
1537
1538     push ecx
1539     mov eax, 0
1540     mov esi, ptrArray
1541     mov ecx, arraySize
1542     cmp ecx, 0
1543     jle L2
1544 L1:
1545     add eax, [esi]
1546     add esi, 4
1547     loop L1
1548 L2:
1549     pop ecx
1550     ret
1551 ArraySum ENDP
1552 END
```

```
1530 ; arraysum.asm
1531 INCLUDE sum.inc
1532
1533 .code
1534 ArraySum PROC,
1535     ptrArray:PTR DWORD,
1536     arraySize:DWORD
1537
1538     push ecx
1539     mov eax, 0
1540     mov esi, ptrArray
1541     mov ecx, arraySize
1542     cmp ecx, 0
1543     jle L2
1544 L1:
1545     add eax, [esi]
1546     add esi, 4
1547     loop L1
1548 L2:
1549     pop ecx
1550     ret
1551 ArraySum ENDP
1552 END
```

```
1557 ; display.asm
1558 INCLUDE sum.inc
1559
1560 .code
1561 DisplaySum PROC,
1562     ptrPrompt:PTR BYTE,
1563     theSum:DWORD
1564
1565     push eax
1566     push edx
1567     mov edx, ptrPrompt
1568     call WriteString
1569     mov eax, theSum
1570     call WriteInt
1571     call Crlf
1572     pop edx
1573     pop eax
1574     ret
1575 DisplaySum ENDP
1576 END
```

```
1580 ; sum_main.asm
1581 INCLUDE sum.inc
1582
1583 Count = 3
1584
1585 .data
1586     prompt1 BYTE "Enter a signed integer: ", 0
1587     prompt2 BYTE "The sum of the integers is: ", 0
1588     array DWORD Count DUP(?)
1589     sum DWORD ?
1590 .code
1591     main PROC
1592         call Clrscr
1593         INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
1594         INVOKE ArraySum, ADDR array, Count
1595         mov sum, eax
1596         INVOKE DisplaySum, ADDR prompt2, sum
1597         call Crlf
1598         exit
1599     main ENDP
1600 END
```

Here is a summary of the two ways to create multimodule programs:

### **Traditional method:**

Use the EXTERN directive to declare symbols that are defined in another module. Use the CALL directive to call procedures in other modules.



### **Advanced method:**

Use the PROTO directive to declare prototypes for procedures in other modules. Use the INVOKE directive to call procedures in other modules. Use the PROC directive to define procedures, and declare parameters for procedures. The advanced method is simpler to use and more efficient, but it is only available in 32-bit mode.



### **Conclusion:**

The advanced method is the preferred method for creating multimodule programs in 32-bit mode. It is simpler to use and more efficient than the traditional method. However, the traditional method is still supported, and it may be necessary for some programs.

## ***Advanced Optional Topic 1-USERS operator***

The USES operator is a powerful tool that can be used to save and restore registers at the beginning and end of a procedure.

However, it should not be used when declaring procedures that access their stack parameters using constant offsets such as [ebp + 8].

The following MySub1 procedure employs the USES operator to save and restore ECX and EDX:

```
1604 MySub1 PROC USES ecx edx  
1605 ret  
1606 MySub1 ENDP
```

The following code is generated by MASM when it assembles MySub1:

```
1610 push ecx  
1611 push edx  
1612 pop edx  
1613 pop ecx  
1614 ret
```

Suppose we combine USES with a stack parameter, as does the following MySub2 procedure. Its parameter is expected to be located on the stack at EBP+8:

```
1620 MySub2 PROC USES ecx edx
1621 push ebp
1622 mov ebp,esp
1623 mov eax,[ebp+8]
1624 ; this is wrong!
1625 pop ebp
1626 ret 4
1627 MySub2 ENDP
```

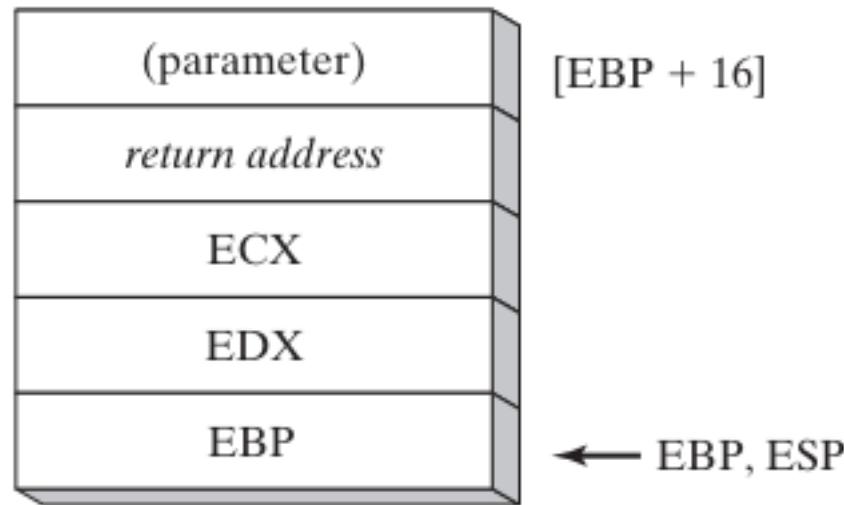
Here is the corresponding code generated by MASM for MySub2:

```
1630 push ecx
1631 push edx
1632 push ebp
1633 mov ebp,esp
1634 mov eax,dword ptr [ebp+8]
1635 pop ebp
1636 pop edx
1637 pop ecx
1638 ret 4
```

An error results because the assembler inserted the PUSH instructions for ECX and EDX at the beginning of the procedure, altering the offset of the stack parameter.

Figure 8-6 below shows how the stack parameter must now be referenced as [EBP+16]. USES modifies the

stack before saving EBP, which corrupts the standard prologue code commonly used for subroutines.



This is why it is important to avoid using the USES operator when declaring procedures that access their stack parameters using constant offsets.

If you need to save and restore registers in such a procedure, you can use the PUSH and POP instructions explicitly.

Here is a more in-depth explanation of why the USES operator causes problems in this case:  
When a procedure is called, the caller pushes the procedure's arguments onto the stack.

The procedure then saves its caller-saved registers (EBP, ESI, EDI, and EBX) onto the stack. The procedure's base pointer (EBP) is set to point to the top of the stack frame.

The USES operator tells the assembler to save and restore the specified registers at the beginning and end of the procedure.

When the USES operator is used in a procedure that accesses its stack parameters using constant offsets, the assembler inserts the PUSH and POP instructions for the specified registers at the beginning and end of the procedure.

This corrupts the standard prologue code commonly used for subroutines, which relies on the stack pointer (ESP) to be pointing to the top of the stack frame.

In the example of MySub2, the USES operator is used to save and restore ECX and EDX. When MySub2 is called, its argument is pushed onto the stack.

The USES operator then causes the assembler to push ECX and EDX onto the stack. This corrupts the stack frame, because the stack pointer is now pointing to the wrong location.

To avoid this problem, you should not use the USES operator in procedures that access their stack parameters using constant offsets.

If you need to save and restore registers in such a procedure, you can use the PUSH and POP instructions explicitly.

## ***Passing 8-Bit and 16-Bit Arguments on the Stack***

### **Passing 8-bit Arguments**

When passing stack arguments to procedures in 32-bit mode, it is best to push 32-bit operands.

This is because the stack pointer (ESP) must be aligned on a doubleword boundary.

If 16-bit operands are pushed onto the stack, ESP will not be aligned and a page fault may occur. Additionally, runtime performance may be degraded.

If you need to pass a 16-bit operand to a procedure in 32-bit mode, you can use the MOVZX instruction to expand the operand to 32 bits before pushing it onto the stack.

For example, the following Uppercase procedure receives a character argument and returns its uppercase equivalent in AL:

```
1645 Uppercase PROC  
1646 push ebp  
1647 mov ebp,esp  
1648 mov al,[esp+8]  
1649 ; AL = character  
1650 cmp al,'a'  
1651 ; less than 'a'?  
1652 jb L1  
1653 ; yes: do nothing  
1654 cmp al,'z'  
1655 ; greater than 'z'?  
1656 ja L1  
1657 ; yes: do nothing  
1658 sub al,32  
1659 ; no: convert it  
1660 L1:  
1661 pop ebp  
1662 ret 4  
1663 ; clean up the stack  
1664 Uppercase ENDP
```

If we pass a character literal to Uppercase, the PUSH instruction will automatically expand the character to 32 bits:

```
1667 push 'x'  
1668 call Uppercase
```

However, if we pass a character variable to Uppercase, the PUSH instruction will not allow us to push an 8-bit operand onto the stack.

To work around this, we can use the MOVZX instruction to expand the character into EAX before pushing it onto the stack:

```
1672 .data  
1673     charVal BYTE 'x'  
1674 .code  
1675     movzx eax,charVal  
1676     ; move with extension  
1677     push eax  
1678     call Uppercase
```

This will ensure that ESP is aligned on a doubleword boundary and that the call to Uppercase is successful.

## **Passing 16-bit Arguments.**

The AddTwo procedure expects two 32-bit integer arguments (the two integers to be added). However, the word1 and word2 variables are 16-bit integers.

Therefore, if we push word1 and word2 onto the stack and call AddTwo, the procedure will not be able to correctly add the two integers.

To fix this, we can zero-extend each argument before pushing it onto the stack. Zero-extension means that the high-order 16 bits of the argument are set to zero.

This will effectively convert the 16-bit argument to a 32-bit argument.

The following code correctly calls AddTwo by zero-extending each argument before pushing it onto the stack:

```
1686 .data
1687     word1 WORD 1234h
1688     word2 WORD 4111h
1689 .code
1690     movzx eax,word1
1691     push eax
1692     movzx eax,word2
1693     push eax
1694     call AddTwo
1695 ; sum is in EAX
```

The **MOVZX instruction** is used to zero-extend the 16-bit word1 and word2 variables into the 32-bit EAX register.

Once the arguments have been zero-extended, they are pushed onto the stack in reverse order (word2 first, then word1).

first, then word1).

When AddTwo is called, it will pop the two arguments off the stack and add them together. The sum of the two integers will be returned in the EAX register.

It is important to note that the caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

In the case of stack parameters, the order and size of the parameters are important.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

## **Passing 64-bit Arguments**

To pass 64-bit integer arguments to procedures in 32-bit mode, we must push the high-order doubleword of the argument first, followed by the low-order doubleword.

This is because the stack grows downwards, so the lower order doubleword of the argument will be at the lower address on the stack.

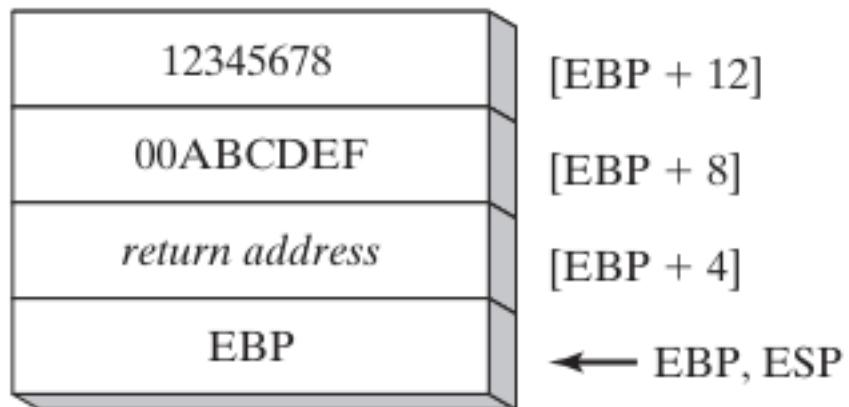
The following WriteHex64 procedure displays a 64-bit integer in hexadecimal:

```
1700 WriteHex64 PROC  
1701     push ebp  
1702     mov ebp,esp  
1703     mov eax,[ebp+12]  
1704     ; high doubleword  
1705     call WriteHex  
1706     mov eax,[ebp+8]  
1707     ; low doubleword  
1708     call WriteHex  
1709     pop ebp  
1710     ret 8  
1711 WriteHex64 ENDP
```

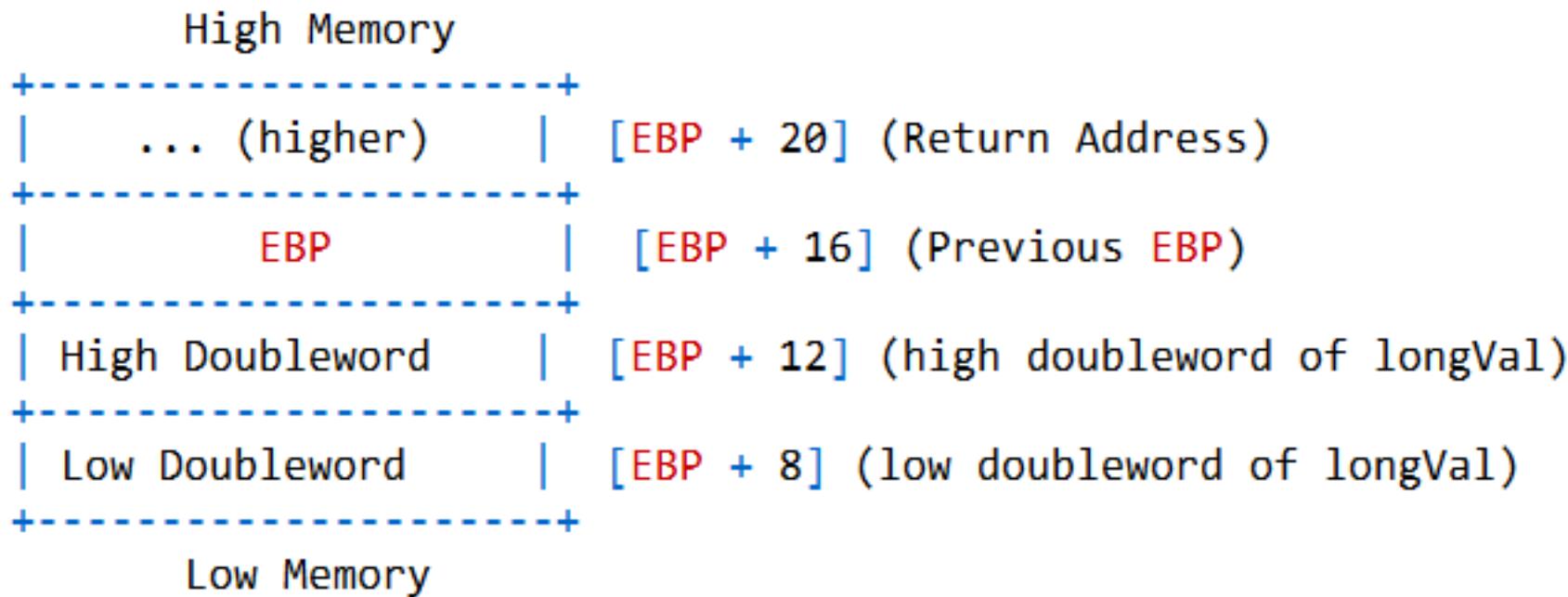
The following sample call to WriteHex64 pushes the upper half of longVal, followed by the lower half:

```
1715 .data  
1716     longVal QWORD 1234567800ABCDEFh  
1717 .code  
1718     push DWORD PTR longVal + 4  
1719     ; high doubleword  
1720     push DWORD PTR longVal  
1721     ; low doubleword  
1722     call WriteHex64
```

Figure below shows the stack frame inside WriteHex64 just after EBP was pushed on the stack and ESP was copied to EBP:



Or



The WriteHex64 procedure can then easily retrieve the high and low doublewords of the argument from the stack and display them in hexadecimal.

It is important to note that the caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

In the case of stack parameters, the order and size of the parameters are important.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

Here is a more in-depth explanation of why we must push the high-order doubleword of a 64-bit integer first when passing it to a procedure in 32-bit mode:

In 32-bit mode, the stack grows downwards. This means that when we push a value onto the stack, the stack pointer (ESP) is decremented.

When we pop a value off the stack, ESP is incremented.

When we pass a 64-bit integer to a procedure in 32-bit mode, we must push the high-order doubleword of the integer first, followed by the low-order doubleword.

This is because we want the integer to be stored on the stack in little-endian order.

In **little-endian order**, the low-order byte of the integer is stored at the lowest address on the stack.

If we were to push the low-order doubleword of the integer first, followed by the high-order doubleword, the integer would be stored on the stack in big-endian order.

In **big-endian order**, the high-order byte of the integer is stored at the lowest address on the stack.

The following diagram shows how a 64-bit integer is stored on the stack in little-endian order:

[EBP + 12] high doubleword of the integer  
[EBP + 8] low doubleword of the integer

The WriteHex64 procedure can then easily retrieve the high and low doublewords of the integer from the stack and display them in hexadecimal.

Why is it important to ensure that the arguments passed to a procedure are consistent with the parameters expected by the procedure?

The caller of a procedure must ensure that the arguments it passes are consistent with the parameters expected by the procedure.

This is because the procedure is expecting certain values to be passed to it in a certain order.

If the caller passes the wrong number of arguments, or if the arguments are in the wrong order or have the wrong size, the procedure may not work correctly or may even crash.

For example, if the WriteHex64 procedure expects one 64-bit integer argument, and the caller passes two 64-bit integer arguments, the procedure will not be able to correctly display the two integers.

Or, if the caller passes a 32-bit integer argument instead of a 64-bit integer argument, the procedure will also not be able to correctly display the integer.

It is important to note that the compiler will not check to make sure that the caller is passing the correct number of arguments to a procedure, or that the arguments are in the correct order or have the wrong size. This is the responsibility of the programmer.

## **Non-Doubleword Local Variables**

In 32-bit mode, the stack grows downwards. This means that when we push a value onto the stack, the stack pointer (ESP) is decremented.

When we pop a value off the stack, ESP is incremented.

When we declare a local variable in a procedure, MASM will allocate space for it on the stack. The size of the space allocated will depend on the size of the variable.

For example, if we declare a byte variable, MASM will allocate one byte of space on the stack. If we declare a doubleword variable, MASM will allocate four bytes of space on the stack.

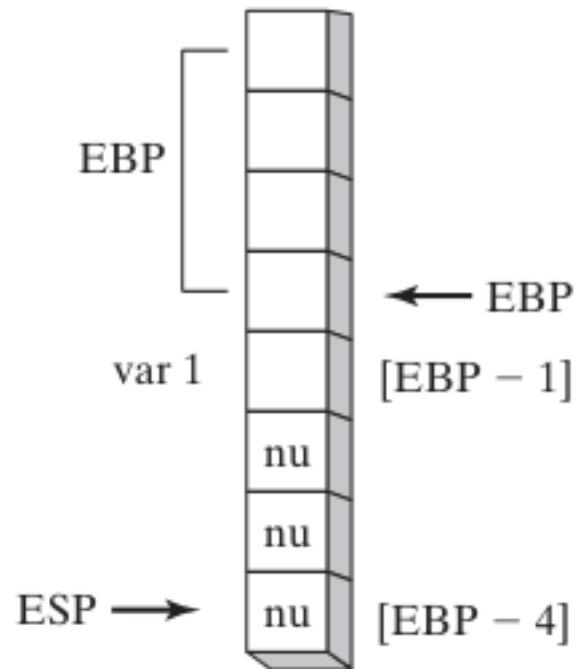
If we declare a local variable of a size that is not a multiple of four bytes (such as a byte or a word), MASM will round the size of the variable up to the next multiple of four bytes.

This is because the stack is aligned on a doubleword boundary. This means that all addresses on the stack must be divisible by four.

For example, if we declare a byte variable named var1 in the Example1 procedure, MASM will allocate four bytes of space for it on the stack, even though the variable is only one byte in size. The remaining three bytes will be unused.

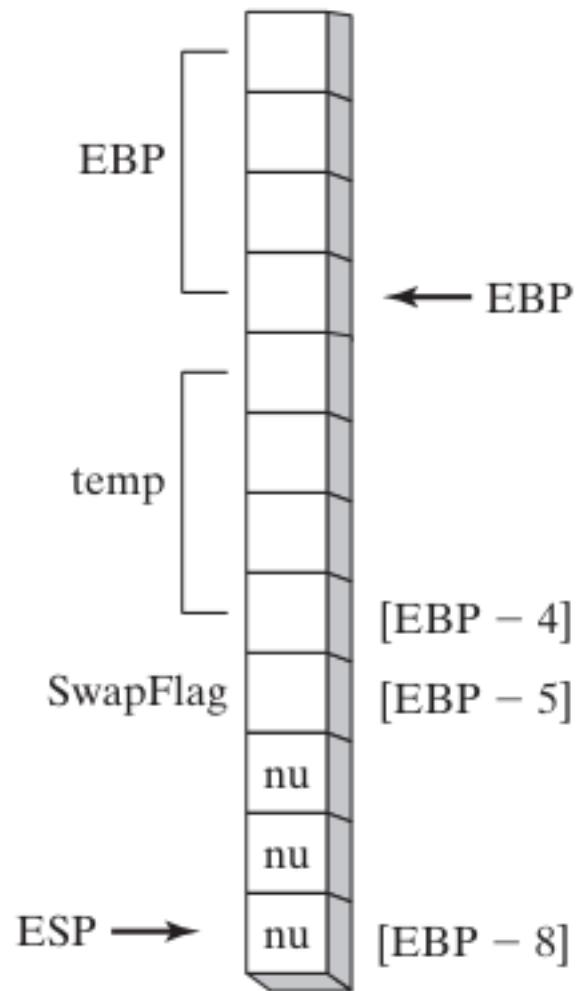
The following diagram shows how the stack looks after the Example1 procedure has been compiled and assembled:

Creating space for local variables (*Example1* Procedure).



The nu blocks represent unused bytes. The following diagram shows how the stack looks after the Example2 procedure has been compiled and assembled:

Creating space in Example 2 for local variables.



The temp variable is a doubleword variable, so it is aligned on a doubleword boundary.

The SwapFlag variable is a byte variable, but it is still allocated four bytes of space on the stack because the stack is aligned on a doubleword boundary.

## Stack size for nested procedure calls

The stack size required for nested procedure calls is the sum of the stack sizes required for each individual procedure call.

This is because the stack is used to store the local variables and return addresses for all active procedure calls.

For example, in the following code:

```
1750 Sub1 PROC
1751 local array1[50]:dword
1752 ; 200 bytes
1753 callSub2
1754 .
1755 .
1756 ret
1757 Sub1 ENDP
1758 Sub2 PROC
1759 local array2[80]:word
1760 ; 160 bytes
1761 callSub3
1762 .
1763 .
1764 ret
1765 Sub2 ENDP
1766 Sub3 PROC
1767 local array3[300]:dword
1768 ; 1200 bytes
1769 .
1770 .
1771 ret
1772 Sub3 ENDP
```

The stack size required for Sub1 is 200 bytes, the stack size required for Sub2 is 160 bytes, and the stack size required for Sub3 is 1200 bytes. Therefore, the total stack size required for this code is 1560 bytes.

This stack size is the minimum amount of stack space that must be available in order for this code to execute correctly. If there is not enough stack space available, the program will crash.

Recursive procedure calls

If a procedure is **called recursively**, the stack space it uses will be approximately the size of its local variables and parameters multiplied by the estimated depth of the recursion.

For example, if a procedure has 100 bytes of local variables and parameters, and it is called recursively to a depth of 10, then the procedure will use approximately 1000 bytes of stack space.

## Stack overflow

If the stack space required for a program exceeds the amount of stack space available, the program will crash. This is called a **stack overflow**.

To avoid stack overflows, it is important to be aware of the **stack space requirements of your program**. You can use the **STACK directive** to reserve additional stack space if necessary.

The stack size required for nested procedure calls is the sum of the stack sizes required for each individual procedure call.

If a procedure is called recursively, the stack space it uses will be approximately the size of its local variables and parameters multiplied by the estimated depth of the recursion.

To avoid stack overflows, it is important to be aware of the stack space requirements of your program and to reserve additional stack space, if necessary.

**Here is a summary of the key points from the chapter:**

- There are two types of procedure parameters: register parameters (faster, used by Irvine libraries) and stack parameters (more flexible).
- A stack frame is the region of stack used by a procedure for its parameters, local variables, saved registers, and return address.
- Parameters can be passed by value (copied) or by reference (address passed). Arrays should be passed by reference.
- Stack parameters are accessed using EBP offset addressing like [EBP-8]. LEA is good for getting stack parameter addresses.
- ENTER/LEAVE instructions manage the stack frame set up/teardown.
- Recursive procedures call themselves directly or indirectly. Recursion works well with repeating data structures.
- Local variables have restricted scope, lifetime tied to the procedure, don't cause naming clashes, and enable recursion.
- INVOKE directive calls procedures with multiple arguments. ADDR passes pointers.

- PROC declares procedures, PROTO prototypes existing procedures.
- Large programs should be split into multiple source code modules for manageability.
- Java bytecode is the machine language in compiled Java programs. The JVM executes it. Bytecodes use a stack-oriented model.

## ***STRINGS AND ARRAYS***

Skip...

### ***String Operations***

---

#### ***String Primitive Instructions***

---

String primitive instructions are a set of instructions that are designed to manipulate strings of data. They are particularly useful for moving, comparing, loading, and storing blocks of data.

The x86 instruction set has five groups of string primitive instructions:

- Move string data: MOVSB, MOVSW, MOVSD
- Compare strings: CMPSB, CMPSW, CMPSD

- Scan string: SCASB, SCASW, SCASD
- Store string data: STOSB, STOSW, STOSD
- Load accumulator from string: LODSB, LODSW, LODSD **Image Description**

The image below shows a table of string primitive instructions, including their descriptions and mnemonics.

**Table 9-1 String Primitive Instructions.**

Instruction	Description
MOVSB, MOVSW, MOVSD	Move string data: Copy data from memory addressed by ESI to memory addressed by EDI.
CMPSB, CMPSW, CMPSD	Compare strings: Compare the contents of two memory locations addressed by ESI and EDI.
SCASB, SCASW, SCASD	Scan string: Compare the accumulator (AL, AX, or EAX) to the contents of memory addressed by EDI.
STOSB, STOSW, STOSD	Store string data: Store the accumulator contents into memory addressed by EDI.
LODSB, LODSW, LODSD	Load accumulator from string: Load memory addressed by ESI into the accumulator.

## Using a Repeat Prefix

By itself, a string primitive instruction only processes a single memory value or pair of values.

However, you can use a repeat prefix to cause the instruction to repeat until a certain condition is

met.

The most common repeat prefix is REP, which causes the instruction to repeat while the ECX register is greater than zero.

For example, the following code would copy 10 bytes from the string1 buffer to the string2 buffer:

---

```
001 cld ; clear Direction flag
002 mov esi, OFFSET string1 ; ESI points to source
003 mov edi, OFFSET string2 ; EDI points to target
004 mov ecx, 10 ; set counter to 10
005 rep movsb ; move 10 bytes
```

## Direction Flag

String primitive instructions increment or decrement the ESI and EDI registers based on the state of the Direction flag. The Direction flag can be explicitly modified using the CLD and STD instructions:

**CLD** clears the Direction flag, causing the ESI and EDI registers to increment.

**STD** sets the Direction flag, causing the ESI and EDI registers to decrement.

Table 9-2 Direction Flag Usage in String Primitive Instructions.

Value of the Direction Flag	Effect on ESI and EDI	Address Sequence
Clear	Incremented	Low-high
Set	Decrement	High-low

When the Direction flag is clear, the ESI and EDI registers are incremented after each operation. This means that the string operation will move from a lower address to a higher address.

When the Direction flag is set, the ESI and EDI registers are decremented after each operation. This means that the string operation will move from a higher address to a lower address.

It is important to set the Direction flag correctly before using a string primitive instruction, as otherwise the ESI and EDI registers may not increment or decrement as intended.

For example, if you want to copy a string from one buffer to another, you would need to clear the Direction flag first. Otherwise, the ESI and EDI registers would be decremented after each operation, and the string would be copied in reverse order.

It is important to set the Direction flag correctly before using a string primitive instruction, as otherwise the ESI and EDI registers may not increment or decrement as intended.

The following example shows how to use the MOVS B instruction to copy a string from one buffer to another:

```
009 ; Copy string1 to string2
010 cld ; clear Direction flag
011 mov esi, OFFSET string1 ; ESI points to source
012 mov edi, OFFSET string2 ; EDI points to target
013 mov ecx, 10 ; set counter to 10
014 rep movsb ; move 10 bytes
```

## **MOVSB, MOVSW, and MOVSD**

The MOVSB, MOVSW, and MOVSD instructions are used to copy data from one memory location to another. They differ in the size of the data that they copy:

- **MOVSB copies bytes.**
- **MOVSW copies words.**
- **MOVSD copies doublewords.**

All three instructions use the ESI and EDI registers to address the source and destination memory locations, respectively.

The Direction flag determines whether the ESI and EDI registers are incremented or decremented after each operation.

The following code shows how to use the MOVSD instruction to copy a doubleword array from one buffer to another:

```
026 ; Copy source to target
027 cld ; clear Direction flag
028 mov ecx, LENGTHOF source ; set REP counter
029 mov esi, OFFSET source ; ESI points to source
030 mov edi, OFFSET target ; EDI points to target
031 rep movsd ; copy doublewords
```

The **CLD instruction** clears the Direction flag, causing the ESI and EDI registers to be incremented after each operation.

The **REP prefix** causes the MOVSD instruction to repeat while the ECX register is greater than zero.

After the code is executed, the ESI and EDI registers will point one position (4 bytes) beyond the end of each array.

This is because the MOVSD instruction copies 4 bytes of data each time it executes.

The notes you provided are unclear and hard to understand because they are missing some key information.

For example, the notes do not explain what the LENGTHOF macro is or how it is used. The notes also do not explain why the ESI and EDI registers are incremented or decremented after each operation.

## CMPSB, CMPSW, and CMPSD

The CMPSB, CMPSW, and CMPSD instructions are used to compare two memory operands. They differ in the size of the operands that they compare:

- **CMPSB** compares bytes
- **CMPSW** compares words
- **CMPSD** compares doublewords

All three instructions use the ESI and EDI registers to address the source and destination operands, respectively.

The Direction flag determines whether the ESI and EDI registers are incremented or decremented after each operation.

**Example: Comparing Doublewords:**

```
035 ; Compare source and target
036 mov esi, OFFSET source
037 mov edi, OFFSET target
038 cmpsd
039 ; compare doublewords
```

If the two doubleword values are equal, the Zero flag is set. If the source doubleword is greater than the target doubleword, the Carry flag is set. Otherwise, the Carry flag is cleared.

To compare multiple doublewords, you can use a repeat prefix with the CMPSD instruction:

```
042 mov esi, OFFSET source
043 mov edi, OFFSET target
044 cld ; clear Direction flag
045 mov ecx, LENGTHOF source ; repetition counter
046 repe cmpsd ; repeat while equal
```

**REPE (Repeat While Equal).** It's used to indicate that a repetitive operation should continue while a certain condition holds, in this case, while the comparison with CMPSD yields equality.

The REPE prefix repeats the comparison until ECX equals zero or a pair of doublewords is found to be different.

The notes you provided are missing some key information, such as an explanation of the LENGTHOF macro and the Direction flag.

The notes also do not provide a complete example of how to compare multiple doublewords using a repeat prefix.

The CMPSB, CMPSW, and CMPSD instructions can be used to implement a variety of string operations, such as searching for a specific character in a string or comparing two strings.

For example, the following code shows how to use the CMPSB instruction to search for the letter 'A' in a string:

```
050 mov esi, OFFSET string  
051 mov edi, OFFSET 'A'  
052 repe cmpsb ; repeat while equal  
053 jne not_found ; jump if not equal
```

If the letter 'A' is found in the string, the jne instruction will jump to the not\_found label. Otherwise, the jne instruction will not jump.

The CMPSB, CMPSW, and CMPSD instructions are powerful tools for comparing memory operands.

## SCASB, SCASW, and SCASD

The SCASB, SCASW, and SCASD instructions compare the value in the AL, AX, or EAX register, respectively, to the byte, word, or doubleword, respectively, addressed by the EDI register.

These instructions are useful for searching for a single value in a string or array.

Combined with the REPE (or REPZ) prefix, the string or array is scanned while ECX > 0 and the value in the AL, AX, or EAX register matches each subsequent value in memory.

The REPNE prefix scans until either the value in the AL, AX, or EAX register matches a value in memory or ECX = 0.

**Example: Scan for a Matching Character.** The following code shows how to use the SCASB instruction to search for the letter 'F' in the string alpha:

```
059 .data
060     alpha BYTE "ABCDEFGHI",0
061 .code
062     mov edi, OFFSET alpha ; EDI points to the string
063     mov al, 'F' ; search for the letter F
064     mov ecx, LENGTHOF alpha ; set the search count
065     cld ; direction = forward
066     repne scasb ; repeat while not equal
067
068 ; Test if the loop stopped because ECX = 0 and the character in AL was not found
069 jnz quit
070
071 ; Found: back up EDI
072 dec edi
073
074 ; Otherwise, the letter F was found
```

The REPNE prefix causes the SCASB instruction to repeat while the Zero flag is cleared and ECX is greater than zero.

The Zero flag is cleared when the value in the AL register does not match the byte at the memory address pointed to by the EDI register.

The jnz instruction jumps to the quit label if the Zero flag is set, meaning that the letter 'F' was not found.

Otherwise, the letter 'F' was found and the dec edi instruction is executed to back up the EDI

register one position.

The jnz instruction is needed to test for the possibility that the loop stopped because ECX = 0 and the character in AL was not found.

If the loop stopped because ECX = 0, then the Zero flag will be set, and the jnz instruction will jump to the quit label.

Otherwise, the letter 'F' was found and the dec edi instruction is executed to back up the EDI register one position.

The SCASB, SCASW, and SCASD instructions are powerful tools for searching for a single value in a string or array.

## **STOSB, STOSW, and STOSD**

The STOSB, STOSW, and STOSD instructions store the contents of the AL, AX, or EAX register, respectively, in memory at the offset pointed to by the EDI register.

- **STOSB: Store byte.**
- **STOSW: Store word.**
- **STOSD: Store doubleword.**

The EDI register is incremented or decremented based on the state of the Direction flag.

When used with the REP prefix, these instructions are useful for filling all elements of a string or array with a single value.

Example: Initialize an Array. The following code shows how to use the STOSB instruction to initialize each byte in the string1 array to 0xFFh:

```
080 ; Initialize each byte in string1 to 0xFFh
081
082 .data
083     Count = 100
084     string1 BYTE Count DUP(?)
085 .code
086     mov al, 0xFFh ; value to be stored
087     mov edi, OFFSET string1 ; EDI points to target
088     mov ecx, Count ; character count
089     cld ; direction = forward
090     rep stosb ; fill with contents of AL
```

The REP prefix causes the STOSB instruction to repeat while ECX is greater than zero.

The CLD instruction clears the Direction flag, causing the EDI register to be incremented after each operation.

After the code is executed, each byte in the string1 array will be set to 0xFFh.

The original notes are unclear and hard to understand because they do not explain why the CLD instruction is needed.

The CLD instruction is needed to clear the Direction flag.

If the Direction flag is set, then the EDI register will be decremented after each operation, and the STOSB instruction will fill the string1 array in reverse order.

The STOSB, STOSW, and STOSD instructions are powerful tools for filling all elements of a string or array with a single value.

```
094 mov al, 0xFFh  
095 mov edi, OFFSET string1  
096 mov ecx, LENGTHOF string1  
097 cld  
098 rep stosb
```

This code would first move the value 0xFFh into the AL register.

It would then move the value of the EDI register into the ECX register.

Next, it would clear the Direction flag.

Finally, it would repeat the STOSB instruction while ECX is greater than zero.

After this code is executed, each byte in the string1 array would be set to 0xFFh.

## LODSB, LODSW, and LODSD

The LODSB, LODSW, and LODSD instructions load a byte, word, or doubleword, respectively, from memory

at the offset pointed to by the ESI register into the AL, AX, or EAX register, respectively.

The ESI register is incremented or decremented based on the state of the Direction flag.

The REP prefix is rarely used with LODS because each new value loaded into the accumulator overwrites its previous contents. Instead, LODS is used to load a single value.

**Example: Load a Single Value.** The following code shows how to use the LODSB instruction to load a single byte from memory into the AL register:

```
101 ; Load a single byte from memory into the AL register  
102 lod sb
```

This instruction will load the byte at the memory address pointed to by the ESI register into the AL register. The ESI register will then be incremented by one.

**Array Multiplication Example.** The following code shows how to use the LODSD and STOSD instructions to multiply each element of an array by a constant value:

```
105 ; Array Multiplication Example
106 .data
107     array DWORD 1,2,3,4,5,6,7,8,9,10
108     ; test data
109     multiplier DWORD 10
110     ; test data
111 .code
112     main PROC
113     cld
114     ; direction = forward
115     mov
116     esi,OFFSET array
117     ; source index
118     mov
119     edi,esi
120     ; destination index
121     mov
122     ecx,LENGTHOF array
123     ; loop counter
124     L1:
125     lodsd
126     ; load [ESI] into EAX
127     mul
128     multiplier ; multiply by a value
129     stosd      ; store EAX into [EDI]
130     loop
131     L1
132     exit
133     main ENDP
134 END main
```

This code first clears the Direction flag. It then moves the offset of the array variable to the ESI and EDI registers.

The ESI register will be used to address the source array, and the EDI register will be used to address the destination array.

The code then moves the length of the array variable to the ECX register. This will be used as the loop counter.

The code then enters a loop. On each iteration of the loop, the LODSD instruction is used to load a doubleword from the source array into the EAX register.

The EAX register is then multiplied by the value of the multiplier variable. The result is then stored in the destination array using the STOSD instruction.

The loop is repeated until the ECX register is equal to zero. After the loop is finished, the code exits.

The original notes are unclear and hard to understand because they do not explain why the Direction flag is cleared and why the ECX register is used as a loop counter.

The Direction flag is cleared so that the ESI and EDI registers will be incremented after each iteration of the loop.

This ensures that the LODSD and STOSD instructions will access the next element of the source and destination arrays, respectively.

The ECX register is used as a loop counter to ensure that the loop will repeat the correct number of times.

The loop will repeat until the ECX register is equal to zero. This means that the loop will repeat the same number of times as the length of the array variable.

The LODSB, LODSW, and LODSD instructions are powerful tools for loading data from memory into the accumulator.

## ***String Procedures in Irvine32***

The Irvine32 library provides a number of procedures for manipulating null-terminated strings. These procedures are similar to the functions in the standard C library.

### **Str\_copy**

The Str\_copy procedure copies a source string to a target string. It takes two arguments: a pointer to the source string and a pointer to the target string. The procedure returns a pointer to the target string.

```
140 ; Copy a source string to a target string.  
141 Str_copy PROTO,  
142 source:PTR BYTE,  
143 target:PTR BYTE
```

The following code shows how to use the `Str_copy` procedure to copy the string "Hello, world!" to the string "buffer":

```
145 mov eax, OFFSET "Hello, world!"  
146 mov ebx, OFFSET buffer  
147 call Str_copy  
148 ; buffer now contains the string "Hello, world!"
```

## Str\_length

The `Str_length` procedure returns the length of a string (excluding the null byte) in the EAX register. It takes one argument: a pointer to the string.

```
152 ; Return the length of a string (excluding the null byte) in EAX.  
153 Str_length PROTO,  
154 pString:PTR BYTE
```

The following code shows how to use the `Str_length` procedure to get the length of the string "Hello, world!" and store it in the EAX register:

```
157 mov eax, OFFSET "Hello, world!"  
158 call Str_length  
159 ; EAX now contains the value 12, which is the length of the string "Hello, world!"
```

## Str\_compare

The Str\_compare procedure compares two strings and sets the Zero and Carry flags in the same way as the CMP instruction. It takes two arguments: pointers to the two strings.

```
162 ; Compare string1 to string2. Set the Zero and
163 ; Carry flags in the same way as the CMP instruction.
164 Str_compare PROTO,
165 string1:PTR BYTE,
166 string2:PTR BYTE
```

The following code shows how to use the Str\_compare procedure to compare the strings "Hello, world!" and "Hello, world!" :

```
170 mov eax, OFFSET "Hello, world!"
171 mov ebx, OFFSET "Hello, world!"
172 call Str_compare
173 ; The Zero flag will be set, indicating that the two strings are equal.
```

## Str\_trim

The Str\_trim procedure trims a given trailing character from a string. It takes two arguments: a pointer to the string and the character to trim.

```
180 ; Trim a given trailing character from a string.  
181 ; The second argument is the character to trim.  
182 Str_trim PROTO,  
183 pString:PTR BYTE,  
184 char:BYTE
```

The following code shows how to use the `Str_trim` procedure to trim the trailing newline character from the string "Hello, world!\n":

```
185 mov eax, OFFSET "Hello, world!\n"  
186 mov ebx, AL ; '\n' character  
187 call Str_trim  
188 ; The string "Hello, world!" is now stored in the memory location pointed to by EAX.
```

## Str\_ucase

The `Str_ucase` procedure converts a string to upper case. It takes one argument: a pointer to the string.

```
190 ; Convert a string to upper case.  
191 Str_ucase PROTO,  
192 pString:PTR BYTE
```

The following code shows how to use the `Str_ucase` procedure to convert the string "hello, world!" to upper case:

```
195 mov eax, OFFSET "hello, world!"  
196 call Str_ucase  
197 ; The string "HELLO, WORLD!" is now stored in the memory location pointed to by EAX.
```

## ***Str\_compare Procedure***

The Str\_compare procedure compares two strings and sets the Carry and Zero flags in the same way as the CMP instruction. It takes two arguments: pointers to the two strings.

The Str\_compare procedure works by comparing byte by byte of the two strings.

If a byte is found that is not equal in both strings, the Str\_compare procedure sets the Carry flag and exits the loop.

If the end of both strings is reached, the Str\_compare procedure sets the Zero flag and exits the loop.

*The following is a more detailed explanation of the Str\_compare procedure:*

```
220 Str_compare PROC USES eax edx esi edi,  
221     string1:PTR BYTE,  
222     string2:PTR BYTE  
223     ; Initialize esi and edi to point to the beginning of the strings.  
224     mov esi, string1  
225     mov edi, string2  
226     ; Start of the loop to compare characters in the strings.  
227 L1:  
228     ; Load the characters from string1 and string2 into AL and DL.  
229     mov al, [esi]  
230     mov dl, [edi]  
231     ; Check if we have reached the end of string1 (null terminator).  
232     cmp al, 0  
233     jne L2 ; If not, continue comparing.  
234     ; If we have reached the end of string1, check if we have also reached the end of string2.  
235     cmp dl, 0  
236     jne L2 ; If not, continue comparing.  
237     ; If we reach this point, both strings are equal, and we exit with ZF = 1.  
238     jmp L3  
239 L2:  
240     ; Increment esi and edi to move to the next characters in the strings.  
241     inc esi  
242     inc edi  
243     ; Compare the characters in AL and DL. If they are equal, continue the loop.  
244     cmp al, dl  
245     je L1 ; Characters are equal; continue comparing.  
246  
247     ; If characters are not equal, exit the loop with flags set.  
248 L3:  
249     ret  
250 Str_compare ENDP
```

**Line 1:** The Str\_compare procedure pushes the EAX, EDX, ESI, and EDI registers onto the stack. This is necessary because the procedure uses these registers.

**Line 2:** The Str\_compare procedure moves the pointers to the two strings into the ESI and EDI registers, respectively.

**Line 3:** The Str\_compare procedure enters a loop. On each iteration of the loop, the following steps are performed:

- The Str\_compare procedure compares the bytes at the memory locations pointed to by the ESI and EDI registers.
- If the bytes are equal, the Str\_compare procedure increments the ESI and EDI registers and continues to the next iteration of the loop.
- If the bytes are not equal, the Str\_compare procedure sets the Carry flag and exits the loop.

**Line 10:** The Str\_compare procedure checks if the end of the string pointed to by the ESI register has been reached. If the end of the string has been reached, the Str\_compare procedure checks if the end of the string pointed to by the EDI register has also been reached.

**Line 12:** If the end of both strings has been reached, the Str\_compare procedure sets the Zero flag and exits the loop.

**Line 14:** The Str\_compare procedure increments the ESI and EDI registers so that they point to the next byte in the respective strings.

**Line 16:** The Str\_compare procedure compares the bytes at the memory locations pointed to by the ESI and EDI registers. If the bytes are equal, the Str\_compare procedure continues to the next iteration

of the loop. If the bytes are not equal, the Str\_compare procedure sets the Carry flag and exits the loop.

**Line 20:** The Str\_compare procedure exits the loop and returns to the caller.

*The following is a table of the flags affected by the Str\_compare procedure:*

Relation	Carry Flag	Zero Flag	Branch If True
string1 < string2	1	0	JB
string1 = string2	0	1	JE
string1 > string2	0	0	JA

This code defines the Str\_compare procedure, which compares two strings character by character. It uses esi and edi to traverse the strings and compares characters while checking for the end of the strings.

The procedure exits with the Zero and Carry flags set according to the result of the comparison. If both strings are equal, ZF is set to 1; otherwise, ZF is set to 0.

The Carry flag is set to indicate the relation between the strings:

(JB for "string1 < string2," JA for "string1 > string2," JE for "string1 = string2")

## ***Str\_length Procedure***

The Str\_length procedure returns the length of a string in the EAX register. It takes one argument: a pointer to the string.

The Str\_length procedure works by scanning the string byte by byte until it reaches the null terminator.

The procedure increments the EAX register for each byte in the string. After reaching the null terminator, the procedure returns the value in the EAX register.

*The following is a more detailed explanation of the Str\_length procedure:*

This code works by scanning the string byte by byte until it reaches the null terminator. The code increments the EAX register for each byte in the string. After reaching the null terminator, the code returns the value in the EAX register.

To use the Str\_length procedure, you would pass the address of the string to the procedure as an argument.

```
255 ;-----  
256 ; Str_length Procedure  
257 ; Calculates the length of a null-terminated string.  
258 ; Returns the length of the string in EAX.  
259 ;-----  
260 Str_length PROC USES edi,  
261     pString:PTR BYTE    ; pointer to the string  
262  
263     mov edi, pString    ; Initialize edi with the pointer to the string.  
264     mov eax, 0            ; Initialize eax to 0, which will store the character count.  
265  
266 L1:  
267     cmp BYTE PTR [edi], 0    ; Check if the current character is the null terminator (end of string).  
268     je L2                  ; If it's the end of the string, exit the loop.  
269  
270     inc edi                ; Move to the next character in the string.  
271     inc eax                ; Increment the character count by 1.  
272  
273     jmp L1                  ; Repeat the loop to process the next character.  
274  
275 L2:  
276     ret                    ; Return with the length of the string in EAX.  
277  
278 Str_length ENDP  
---
```

**Line 1:** The Str\_length procedure pushes the EDI register onto the stack. This is necessary because the procedure uses this register.

**Line 2:** The Str\_length procedure moves the pointer to the string into the EDI register.

**Line 3:** The Str\_length procedure moves the value 0 into the EAX register. This will be used to store the length of the string.

**Line 4:** The Str\_length procedure enters a loop. On each iteration of the loop, the following steps are performed:

- The Str\_length procedure compares the byte at the memory location pointed to by the EDI register to the null terminator (0x00).
- If the byte is equal to the null terminator, the Str\_length procedure exits the loop.
- If the byte is not equal to the null terminator, the Str\_length procedure increments the EDI register and the EAX register.

Line 10: The Str\_length procedure exits the loop and returns to the caller.

*The following is an example of how to use the Str\_length procedure:*

```
282 ; Get the length of the string "Hello, world!"  
283 mov eax, OFFSET "Hello, world!"  
284 call Str_length  
285  
286 ; The EAX register will now contain the value 12, which is the length of the string "Hello, world!"
```

## **Str\_copy Procedure**

The Str\_copy procedure copies a null-terminated string from a source location to a target location.

It takes two arguments: pointers to the source and target strings, respectively.

The Str\_copy procedure works by first calling the Str\_length procedure to get the length of the source string.

The Str\_copy procedure then copies the source string to the target string byte by byte using the REP MOVSB instruction.

*The following is the MASM code for the Str\_copy procedure:*

```
325 ;-----  
326 ; Str_copy Procedure  
327 ; Copies a string from the source to the target.  
328 ; Requires: the target string must contain enough space  
329 ; to hold a copy of the source string.  
330 ;-----  
331 Str_copy PROC USES eax ecx esi edi,  
332     source:PTR BYTE,    ; source string  
333     target:PTR BYTE    ; target string  
334  
335     INVOKE Str_length, source    ; Calculate the length of the source string and store it in EAX.  
336     mov ecx, eax            ; Copy the length into ECX for REP count.  
337     inc ecx                ; Add 1 for the null byte at the end of the string.  
338  
339     mov esi, source        ; Initialize esi with the source pointer.  
340     mov edi, target        ; Initialize edi with the target pointer.  
341  
342     cld                    ; Set the direction flag to forward.  
343  
344     rep movsb             ; Use REP to copy the string byte by byte from source to target.  
345  
346     ret                    ; Return when the string is copied.  
347  
348 Str_copy ENDP
```

This code defines the `Str_copy` procedure, which copies a null-terminated string from the source to the target.

It uses the `Str_length` procedure to calculate the length of the source string, sets up pointers to both source and target strings, and then uses the `rep movsb` instruction to copy the string byte by

byte.

The procedure returns once the entire string is copied to the target.

To use the `Str_copy` procedure, you would pass the addresses of the source and target strings to the procedure as arguments.

For example, to copy the string "Hello, world!" from one location to another, you would use the following code:

```
350 mov eax, OFFSET "Hello, world!"
351 mov ebx, OFFSET target_string
352 call Str_copy
353
354 ; The target_string variable will now contain a copy of the string "Hello, world!"
```

## ***Str\_trim Procedure***

The `Str_trim` procedure removes all occurrences of a specified trailing character from the end of a null-terminated string. For example, it can trim trailing spaces from a string.

It takes two parameters:

`pString` - Pointer to the null-terminated ASCII string to trim.

`char` - The ASCII character to trim from the end of the string. It does not return anything, but modifies the string in-place by truncating it.

## Logic

It handles several cases:

1. Empty string - nothing to do
2. String with trailing character(s) - remove them
3. String with only the trailing char - truncate to empty string
4. String without any trailing char - leave unchanged
5. String with trailing char(s) followed by other chars - remove only trailing char(s)

To trim the string, it inserts a null byte (`\0`) after the last character to keep. Any chars after the null become insignificant.

It uses these steps:

1. Get length of string
2. Check if length is 0 and exit if so (empty string case)
3. Initialize loop counter to string length
4. Point to last character
5. Loop backwards until beginning

- Check if current char matches trailing char to trim
- If yes, decrement counter to keep backing up
- If no, insert null byte after current char and exit loop
- Insert null byte truncates the string

```
360 Str_trim PROC USES eax ecx edi,  
361     pString:PTR BYTE, ; string pointer  
362     char: BYTE         ; trailing char to trim  
363  
364  
365     mov edi,pString    ; point EDI to string  
366     INVOKE Str_length,edi ; get length in EAX  
367  
368     cmp eax,0           ; is length 0?  
369     je L3                ; yes, exit  
370  
371     mov ecx,eax          ; ECX = length  
372     dec eax              ; EAX = length - 1  
373     add edi,eax          ; point to last char  
374  
375 L1:  
376     mov al,[edi]          ; load character  
377     cmp al,char           ; compare to trailing char  
378     jne L2                ; no match, insert null  
379     dec edi              ; match, keep backing up  
380     loop L1  
381  
382 L2:  
383     mov BYTE PTR [edi+1],0 ; insert null byte  
384  
385 L3:  
386     ret  
387 Str_trim ENDP
```

**Line 1:** The `Str_trim` procedure pushes the EAX, ECX, and EDI registers onto the stack. This is necessary because the procedure uses these registers.

**Line 2:** The `Str_trim` procedure moves the pointer to the string to be trimmed into the EDI register.

**Line 3:** The `Str_trim` procedure calls the `Str_length` procedure to get the length of the string. The length of the string is stored in the EAX register.

**Line 4:** The `Str_trim` procedure compares the length of the string to 0. If the length of the string is equal to 0, then the string is empty and the procedure exits.

**Line 6:** The `Str_trim` procedure moves the length of the string to the ECX register. This will be used as the loop counter.

**Line 7:** The `Str_trim` procedure decrements the EAX register. This will be used to point to the last character in the string.

**Line 8:** The `Str_trim` procedure adds the EAX register to the EDI register. This will point the EDI register to the last character in the string.

**Line 9:** The `Str_trim` procedure enters a loop. On each iteration of the loop, the following steps are performed:

- The `Str_trim` procedure moves the byte at the memory location pointed to by the EDI register into the AL register.
- The `Str_trim` procedure compares the AL register to the character to be trimmed.

- If the two characters are equal, then the Str\_trim procedure decrements the EDI register and continues to the next iteration of the loop.
- If the two characters are not equal, then the Str\_trim procedure breaks out of the loop.

**Line 14:** The Str\_trim procedure inserts a null byte at the memory location pointed to by the EDI register plus one. This will terminate the string at the last character that is not the character to be trimmed.

**Line 15:** The Str\_trim procedure exits the loop.

**Line 16:** The Str\_trim procedure pops the EAX, ECX, and EDI registers from the stack.

**Line 17:** The Str\_trim procedure returns.

Here is an example of how to use the Str\_trim procedure:

```
390 ; Trim all trailing spaces from the string "Hello, world!    "
391 mov eax, OFFSET "Hello, world!    "
392 call Str_trim
393 ; The EAX register will now contain a pointer to the string "Hello, world!"
```

## ***Str\_uCase Procedure***

```
397 ;-----  
398 ; Str_ucase Procedure  
399 ; Converts a null-terminated string to uppercase.  
400 ; Returns: nothing  
401 ;-----  
402 Str_ucase PROC USES eax esi,  
403     pString:PTR BYTE  ; Pointer to the string  
404  
405     mov esi, pString ; Initialize esi with the address of the string.  
L1:  
407     mov al, [esi]      ; Load the character from the string.  
408     cmp al, 0          ; Check if it's the end of the string.  
409     je L3              ; If yes, exit the loop.  
410     cmp al, 'a'        ; Compare the character with 'a'.  
411     jb L2              ; If it's below 'a', go to L2.  
412     cmp al, 'z'        ; Compare the character with 'z'.  
413     ja L2              ; If it's above 'z', go to L2.  
414  
415     and BYTE PTR [esi], 11011111b  
416     ; Convert the character to uppercase by clearing the 6th bit.  
L2:  
418     inc esi           ; Move to the next character.  
419     jmp L1              ; Repeat the loop.  
L3:  
421     ret                ; Return when the entire string is converted to uppercase.  
422 Str_ucase ENDP
```

The first part of the code sets up the procedure. It expects a pointer to a string (null-terminated) in pString, and it initializes esi with the address of this string.

Here's what's happening within the loop:

**mov al, [esi]** loads the character at the memory address pointed to by esi into the al register.

**cmp al, 0** checks if the character is the null terminator (end of the string). If it is, the loop exits (je L3).

**cmp al, 'a'** compares the character with the ASCII value of 'a'. If the character is below 'a', it jumps to L2.

**cmp al, 'z'** compares the character with the ASCII value of 'z'. If the character is above 'z', it also jumps to L2.

**and BYTE PTR [esi], 11011111b** converts the character to uppercase by clearing the 6th bit (bit 5) in the ASCII code. This changes lowercase letters to uppercase.

**L2:** marks the location to which the code jumps when the character is not a lowercase letter, ensuring it's skipped.

**inc esi** increments the esi register to point to the next character in the string.

**jmp L1** jumps back to the beginning of the loop, continuing the process until the end of the string is reached.

**Finally, L3:** is the label that is reached when the end of the string is detected. At this point, the loop exits, and the procedure returns.

The entire string is converted to uppercase by iterating through each character, and no value is returned; the original string is modified in memory.

## ***String Library Demo Program***

The "String Library Demo" program demonstrates the usage of string-handling procedures from the Irvine32 library. It performs the following tasks:

Trimming trailing characters from string\_1 using the Str\_trim procedure.

Converting string\_1 to uppercase using the Str\_ucase procedure.

Comparing string\_1 to string\_2 using the Str\_compare procedure.

Displaying the length of string\_2 using the Str\_length procedure. Here's the code with detailed explanations:

```
INCLUDE Irvine32.inc
.data
string_1 BYTE "abcde///", 0
string_2 BYTE "ABCDE", 0
```

```
msg0 BYTE "string_1 in upper case: ", 0
msg1 BYTE "string_1 and string_2 are equal", 0
msg2 BYTE "string_1 is less than string_2", 0
msg3 BYTE "string_2 is less than string_1", 0
msg4 BYTE "Length of string_2 is ", 0
msg5 BYTE "string_1 after trimming: ", 0

.code
main PROC
    call trim_string ; Remove trailing characters from string_1.
    call upper_case ; Convert string_1 to uppercase.
    call compare_strings ; Compare string_1 to string_2.
    call print_length ; Display the length of string_2.
    exit
main ENDP

trim_string PROC
    ; Remove trailing characters from string_1.
    INVOKE Str_trim, ADDR string_1, '/'
    mov edx, OFFSET msg5
    call WriteString
    mov edx, OFFSET string_1
    call WriteString
    call Crlf
    ret
trim_string ENDP

upper_case PROC
    ; Convert string_1 to upper case.
    mov edx, OFFSET msg0
    call WriteString
    INVOKE Str_ucase, ADDR string_1
    mov edx, OFFSET string_1
    call WriteString
    call Crlf
    ret
upper_case ENDP
```

```

compare_strings PROC
    ; Compare string_1 to string_2.
    INVOKE Str_compare, ADDR string_1, ADDR string_2
    .IF ZERO?
        mov edx, OFFSET msg1
    .ELSEIF CARRY?
        mov edx, OFFSET msg2
    .ELSE
        mov edx, OFFSET msg3
    .ENDIF
    call WriteString
    call Crlf
    ret
compare_strings ENDP

print_length PROC
    ; Display the length of string_2.
    mov edx, OFFSET msg4
    call WriteString
    INVOKE Str_length, ADDR string_2
    call WriteDec
    call Crlf
    ret
print_length ENDP

END main

```

The program's output is as follows:

After trimming string\_1, it displays "string\_1 after trimming: abcde."

After converting string\_1 to uppercase, it displays "string\_1 in upper case: ABCDE."

It then compares string\_1 and string\_2 and displays one of the messages depending on the result.

Finally, it displays the length of string\_2.

This program showcases the use of various string-handling procedures from the Irvine32 library and provides informative messages for each step.

## ***Strings using Irvine64***

```
427 INCLUDE Irvine64.inc
428 .data
429     source BYTE "AABCDEFGAABCDFG", 0
430     ; size = 15
431     target BYTE 20 DUP(0)
432
433 .code
434     Str_compare PROTO
435     Str_length PROTO
436     Str_copy PROTO
437     ExitProcess PROTO
438
439 main PROC
440     mov rcx, OFFSET source
441     call Str_length
442     ; Returns length in RAX
443     mov rsi, OFFSET source
444     mov rdi, OFFSET target
445     call Str_copy
446     ; We just copied the string, so they should be equal.
447     call Str_compare
448     ; ZF = 1, strings are equal
449     ; Change the first character of the target string, and
450     ; compare them again.
451     mov BYTE PTR [rdi], 'B'
452     call Str_compare
453     ; CF = 1, source < target
454     mov ecx, 0
455     call ExitProcess
456 main ENDP
```

## ***Actual Implementation of Procedures:***

To have a complete working program, you need to provide the actual implementation of the Str\_compare, Str\_length, and Str\_copy procedures.

These procedures are essential for the functionality of your program. They should be implemented with appropriate assembly code to perform the desired operations.

I'll provide you with the implementation of these procedures in Irvine64 assembly:

```
; Str_compare Procedure
; Compares two strings
; Receives:
; RSI points to the source string
; RDI points to the target string
; Returns:
; Sets ZF if the strings are equal
; Sets CF if source < target

Str_compare PROC
    ; Implementation of Str_compare
    ; ...
    ret
Str_compare ENDP

; Str_length Procedure
; Gets the length of a string
; Receives: RCX points to the string
; Returns: length of string in RAX
```

```
Str_length PROC
    ; Implementation of Str_length
    ; ...
    ret
Str_length ENDP

; Str_copy Procedure
; Copies a source string to a location indicated by a target pointer
; Receives:
; RSI points to the source string
; RDI points to the target string
; Returns: nothing

Str_copy PROC
    ; Implementation of Str_copy
    ; ...
    ret
Str_copy ENDP
```

### ***Output and Display:***

In the provided test program, there is no code for displaying the results of these operations. You should add code to display whether the strings are equal, the length of the string, and the comparison results. For example, you can use WriteString and WriteDec functions to display these results:

```
500 mov rsi, OFFSET msg1
501 call WriteString ; Display result message
502 call Crlf
503 ; Check ZF and CF flags to determine equality or comparison result
504 ; Display results accordingly
```

### Irvine64 Library Setup:

The Irvine64 library needs to be included and set up properly in your assembly environment. You should have instructions at the beginning of your program to include the Irvine64 library and set it up. This usually involves specifying the paths and configurations for the Irvine64 library. Here is an example:

```
512 INCLUDE Irvine64.inc ; Include Irvine64 library
513
514 .data
515 ; Your data declarations go here
516
517 .code
518 main PROC
519 ; Your program's main code goes here
520
521 main ENDP
522
523 END main
```

*Actual Program:*

```
532 INCLUDE Irvine64.inc
533
534 ; -----
535 ; Str_compare
536 ; Compares two strings
537 ; Receives:
538 ; RSI points to the source string
539 ; RDI points to the target string
540 ; Returns:
541 ; Sets ZF if the strings are equal
542 ; Sets CF if source < target
543 ; -----
544 Str_compare PROC
545     USES rax rdx rsi rdi
546
547 L1:
548     mov al, [rsi]
549     mov dl, [rdi]
550     cmp al, 0          ; End of string1?
551     jne L2              ; No
552     cmp dl, 0          ; Yes: End of string2?
553     jne L2              ; No
554     jmp L3              ; Yes, exit with ZF = 1
555
```

```
556 L2:  
557     inc rsi          ; Point to next  
558     inc rdi  
559     cmp al, dl      ; Characters equal?  
560     je L1           ; Yes, continue loop  
561                 ; No: Exit with flags set  
562  
563 L3:  
564     ret  
565  
566 Str_compare ENDP  
567  
568 ; -----  
569 ; Str_copy  
570 ; Copies a source string to a location indicated by a target pointer  
571 ; Receives:  
572 ; RSI points to the source string  
573 ; RDI points to the location where the copied string will be stored  
574 ; Returns: nothing  
575 ; -----  
576 Str_copy PROC  
577     USES rax rcx rsi rdi  
578  
579     mov rcx, rsi    ; Get length of the source string  
580     call Str_length ; Returns length in RAX  
581     mov rcx, rax    ; Loop counter  
582     inc rcx         ; Add 1 for the null byte  
583     cld             ; Direction = up  
584     rep movsb       ; Copy the string  
585     ret
```

```
587 Str_copy ENDP  
588  
589 ; -----  
590 ; Str_length  
591 ; Gets the length of a string  
592 ; Receives: RCX points to the string  
593 ; Returns: length of the string in RAX  
594 ; -----  
595 Str_length PROC  
596     USES rdi  
597  
598     mov rdi, rcx      ; Get the pointer  
599     mov eax, 0        ; Character count  
600 L1:  
601     cmp BYTE PTR [rdi], 0  ; End of string?  
602     je L2                ; Yes: quit  
603     inc rdi              ; No: Point to the next  
604     inc rax              ; Add 1 to count  
605     jmp L1  
606 L2:  
607     ret                  ; Return count in RAX  
608  
609 Str_length ENDP
```

```
611 .data
612     source BYTE "AABCDEFGAABCDEFG",0
613     target BYTE 20 dup(0)
614
615 .code
616     main PROC
617         mov rcx, offset source
618         call Str_length      ; Returns length in RAX
619         mov rsi, offset source
620         mov rdi, offset target
621         call Str_copy
622         ; We just copied the string, so they should be equal.
623         call Str_compare
624         ; ZF = 1, strings are equal
625         ; Change the first character of the target string, and compare them again.
626         mov target, 'B'
627         call Str_compare
628         ; CF = 1, source < target
629         mov ecx, 0
630         call ExitProcess
631
632     main ENDP
633
634 END main
```

*Explanation:*

**Irvine64 Library:** The INCLUDE Irvine64.inc statement includes the Irvine64 library, providing access to Irvine's assembly functions and features.

**USES Keyword:** In the Str\_compare and Str\_copy procedures, the USES keyword is used to specify registers that will be pushed onto the stack and popped off the stack upon return from the procedure. This helps maintain the calling conventions.

**Str\_compare Procedure:** Compares two strings pointed to by RSI and RDI. It sets the Zero Flag (ZF) if the strings are equal and the Carry Flag (CF) if the source is less than the target.

**Str\_copy Procedure:** Copies a source string (pointed to by RSI) to a location indicated by the target pointer (RDI). It calculates the length of the source string using Str\_length, then uses rep movsb to perform the copy.

**Str\_length Procedure:** Calculates the length of a null-terminated string. It receives a pointer in RCX, and the result is returned in RAX.

**.data Section:** Data declarations for source and target strings.

**.code Section:** The main procedure demonstrates the use of these string procedures, copying the string, comparing strings, and changing a character for comparison.

This code illustrates how to use these string procedures in Irvine64 assembly, focusing on the Irvine64 register usage, stack management, and proper procedure calling conventions.

It's essential to configure your environment correctly to work with Irvine64 and ensure you have the Irvine64 library properly set up.

## 2D Arrays

**Row-major order** and **column-major order** are two different ways of storing a two-dimensional array in memory. The main difference is the order in which the elements of the array are stored.

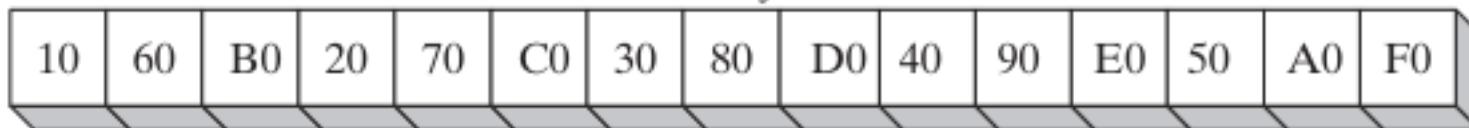
Logical arrangement:

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0

Row-major order

10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Column-major order



## Row-major order

**Row-major order** is the most common method, and it is the method used by most high-level programming languages. In row-major order, the elements of each row are stored contiguously in memory.

Row-major order:

```
[0][0] [0][1] [0][2]  
[1][0] [1][1] [1][2]  
[2][0] [2][1] [2][2]
```

This means that the first element of the first row is stored at the beginning of the memory block, followed by the second element of the first row, and so on. The last element of the first row is followed by the first element of the second row, and so on. This continues until the last element of the last row is stored.

## Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

## Column-major order

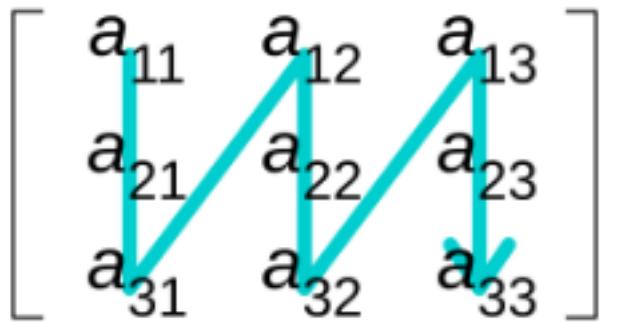
**Column-major order** is less common, but it is used in some applications, such as linear algebra. In column-major order, the elements of each column are stored contiguously in memory.

Column-major order:

[0][0]	[1][0]	[2][0]
[0][1]	[1][1]	[2][1]
[0][2]	[1][2]	[2][2]

This means that the first element of the first column is stored at the beginning of the memory block, followed by the first element of the second column, and so on. The last element of the first column is followed by the second element of the second column, and so on. This continues until the last element of the last column is stored.

## Column-major order



### Which order to use?

The choice of which order to use depends on the application. Row-major order is generally more efficient for accessing elements of the array by row, while column-major order is more efficient for accessing elements of the array by column.

How to implement a two-dimensional array in MASM To implement a two-dimensional array in MASM, you can use either row-major order or column-major order. However, it is important to be **consistent with the ordering that you choose**.

### Implementations

*To implement a two-dimensional array in row-major order, you can use the following steps:*

- Allocate a block of memory for the array.
- The size of the memory block depends on the size of the array and the data type of the elements of

the array.

- Initialize the elements of the array. To access an element of the array, you can use the following formula:

```
element[i][j] = array[i * number_of_columns + j]
```

where  $i$  is the row index and  $j$  is the column index.

For example, the following code implements a two-dimensional array of integers in row-major order:

```
655 ; Declare a two-dimensional array of integers.  
656 array dw 100 dup(0)  
657  
658 ; Initialize the elements of the array.  
659 mov ebx, 1  
660 mov ecx, 100  
661 mov edi, array  
662 loop:  
663 mov dword ptr [edi], ebx  
664 inc ebx  
665 inc edi  
666 loop loop  
667  
668 ; Access an element of the array.  
669 mov eax, array[1 * 10 + 2]
```

Explanation:

- The array is declared as 100 contiguous DWORDs initialized to 0. This creates a  $10 \times 10$  array since each DWORD is 2 bytes ( $10 * 10 = 100$ ).
- EBX is used as a counter from 1 to 100 to store sequential values in the array.
- EDI points to the start of the array and is incremented each iteration to move through the elements.
- ECX counts iterations from 1 to 100 to fill all elements.
- Array elements are accessed using `rownum*cols + col`. Here `row 1, col 2` is at offset  $10 + 2 = 12$ .

- The elements are stored in row-major order - row 1, then row 2, etc sequentially in memory.

So this shows a typical way to declare, initialize, and access a 2D array in MASM using row-major layout.

## Base-Index Operands

A base-index operand is a type of operand that allows you to access memory using the sum of two register values: the base register and the index register.

Base-index operands are very useful for accessing arrays, because they allow you to calculate the address of an element of an array using the row and column indices of the element.

To use a base-index operand, you must first load the base register with the address of the array.

You can then load the index register with the row and/or column index of the element of the array that you want to access.

Finally, you can use the base-index operand to access the element of the array.

The following example shows how to use a base-index operand to access an element of a two-dimensional array:

```
672 ; Declare a two-dimensional array of integers.  
673 array dw 1000h, 2000h, 3000h  
674  
675 ; Load the base register with the address of the array.  
676 mov ebx, OFFSET array  
677  
678 ; Load the index register with the row and column indices of the element that we want to access.  
679 mov esi, 1 ; row index  
680 mov edi, 2 ; column index  
681  
682 ; Calculate the address of the element using the base-index operand.  
683 mov eax, [ebx + esi * 2 + edi * 4]  
684  
685 ; Display the value of the element.  
686 mov edx, 0 ; service number  
687 mov ecx, 1 ; buffer offset  
688 mov al, [eax] ; buffer byte  
689 int 21h ; system call
```

The above code will display the value 2000h, which is the value of the element at row 1, column 2 of the array.

### Column-Major Order vs. Row-Major Order

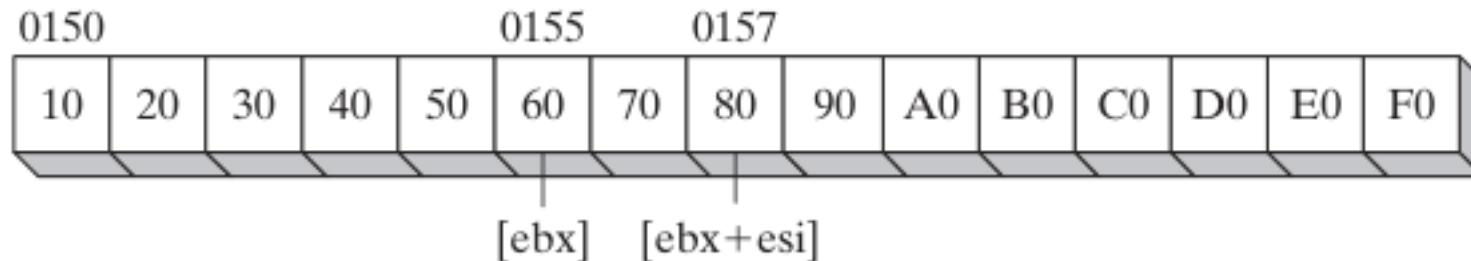
Two-dimensional arrays can be stored in memory in two different ways: column-major order and row-major order.

In **column-major order**, the elements of each column are stored contiguously in memory.

In **row-major order**, the elements of each row are stored contiguously in memory.

Most high-level programming languages use row-major order to store two-dimensional arrays.

Therefore, if you are writing assembly language code that will be used with a high-level language, you should use row-major order to store your two-dimensional arrays.



To access an elements of the array above, you can use the following formula:

```
element[row_index][column_index] = array[row_index * number_of_columns + column_index]
```

where `row_index` is the row index of the element and `column_index` is the column index of the element.

For example, to access the element at row 1, column 2 of the array, you would use the following code:

```
698 mov ebx, OFFSET array ; load the base register with the address of the array  
699 mov esi, 1 ; load the index register with the row index  
700 mov al, [ebx + esi * 2] ; access the element at row 1, column 2
```

The al register will now contain the value of the element at row 1, column 2 of the array.

## Calculating a Row Sum

The calc\_row\_sum procedure calculates the sum of a selected row in a matrix of 8-bit integers. It receives the following inputs:

- EBX: The offset of the matrix in memory.
- EAX: The index of the row to calculate the sum for.
- ECX: The size of each row in the matrix, in bytes.
- The procedure returns the sum of the row in the EAX register.

The procedure works by first calculating the offset of the row in the matrix. This is done by multiplying the row index by the row size.

The procedure then adds this offset to the base address of the matrix to get the address of the first element in the row.

The procedure then iterates over the row, adding each element to the accumulator. The accumulator is initialized to 0 before the loop starts.

The loop iterates until the end of the row is reached.

After the loop finishes, the procedure returns the sum in the accumulator in the EAX register.

Here is a more detailed explanation of the code:

```
707 ;-----  
708 ; calc_row_sum  
709 ; Calculates the sum of a row in a byte matrix.  
710 ; Receives: EBX = table offset, EAX = row index,  
711 ; ECX = row size, in bytes.  
712 ; Returns: EAX holds the sum.  
713 ;-----  
714 calc_row_sum PROC USES ebx ecx edx esi  
715  
716 ; Calculate the offset of the row.  
717 mul ecx ; row index * row size  
718  
719 ; Add the offset to the base address of the matrix to get the address of the first element in the row.  
720 add ebx, eax ; row offset  
721  
722 ; Initialize the accumulator.  
723 mov eax, 0 ; accumulator  
724  
725 ; Set the column index to 0.  
726 mov esi, 0 ; column index  
727  
728 ; Loop over the row, adding each element to the accumulator.  
729 L1:  
730 movzx edx, BYTE PTR[ebx + esi] ; get a byte  
731 add eax, edx ; add to accumulator  
732 inc esi ; next byte in row  
733 loop L1  
734  
735 ; Return the sum in the accumulator.  
736 ret  
737 calc_row_sum ENDP
```

The BYTE PTR operand size in the MOVZX instruction is required to clarify the operand size. The MOVZX instruction converts a byte to a doubleword.

The BYTE PTR operand size tells the assembler that the operand at the address specified by the ebx + esi register is a byte.

The calc\_row\_sum procedure is a useful example of how to use base-index addressing to access elements of a two-dimensional array and to perform common tasks on arrays, such as calculating the sum of a row.

## Scale Factors

A scale factor is a multiplier that is used to scale the index operand when accessing elements of an array using base-index addressing.

The scale factor is required because the size of the elements in an array can vary.

For example, if an array contains bytes, then the scale factor is 1. If an array contains words, then the scale factor is 2. If an array contains doublewords, then the scale factor is 4.

The following table shows the scale factor for different types of data:

Data type	Scale factor
Byte	1
Word	2
Doubleword	4
Quadword	8

To use a scale factor, you simply multiply the index operand by the scale factor before adding it to the base address of the array.

This will give you the address of the element in the array at the specified index.

For example, the following code accesses the element at row 1, column 2 of an array of words:

```

745 ; tableW is an array of words.
746 mov ebx, OFFSET tableW ; load the base register with the address of the array
747 mov esi, 2 ; load the index register with the column index
748 mov ax, [ebx + esi * TYPE tableW] ; access the element at row 1, column 2

```

The TYPE tableW operand size in the MOV instruction tells the assembler that the elements in the array are words.

array are words.

Therefore, the scale factor is 2. The `esi * TYPE tableW` operand is multiplied by 2 before it is added to the base address of the array.

This gives us the address of the element at row 1, column 2.

Scale factors can be useful for writing efficient code to access arrays.

By using scale factors, you can avoid having to keep track of the size of the elements in the array. This can make your code more readable and maintainable.

## Base-Index-Displacement Operands

A base-index-displacement operand is a type of operand that allows you to access memory using the sum of the following values:

- A **displacement**.
- A **base register**.
- An **index register**.
- An **optional scale factor**.

Base-index-displacement operands are well suited for processing two-dimensional arrays. The displacement can be an array name, the base operand can hold the row offset, and the index operand can hold the column offset.

The following example shows how to use a base-index-displacement operand to access an element of a two-dimensional array of doublewords:

two-dimensional array of doublewords:

```
759 ; tableD is a two-dimensional array of doublewords.  
760 ; Rowsize is the size of each row in the array, in bytes.  
761  
762 mov ebx, Rowsize ; load the base register with the row offset  
763 mov esi, 2 ; load the index register with the column offset  
764 mov eax, tableD[ebx + esi * TYPE tableD] ; access the element at row 1, column 2
```

The `tableD[ebx + esi * TYPE tableD]` operand specifies the address of the element at row 1, column 2 of the array.

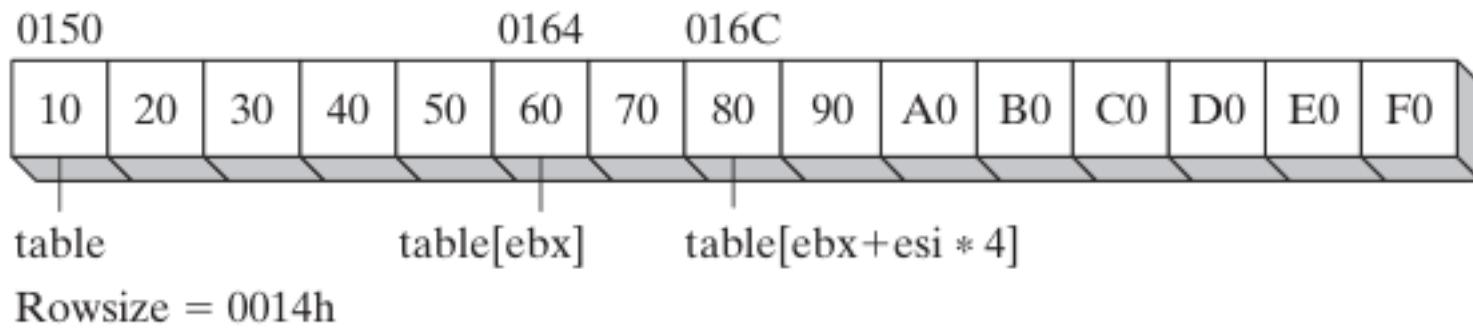
The `ebx` register contains the row offset, and the `esi` register contains the column offset.

The `TYPE tableD` operand size tells the assembler that the elements in the array are doublewords. Therefore, the scale factor is 4.

Base-index-displacement operands can be useful for writing efficient code to access arrays.

By using base-index-displacement operands, you can avoid having to keep track of the size of the elements in the array and the offset of each row in the array.

This can make your code more readable and maintainable.



The diagram you provided shows the positions of the EBX and ESI registers relative to the array tableD. The EBX register contains the row offset, and the ESI register contains the column offset.

The tableD array begins at offset 0150h. The EBX register contains the value 20h, which is the size of each row in the array, in bytes. Therefore, the EBX register points to the beginning of the second row in the array.

The ESI register contains the value 2, which is the column index of the element that we want to access. Therefore, the ESI register points to the third element in the second row of the array.

Base-index-displacement operands are a powerful tool for accessing arrays in assembly language.

## Base-Index Operands in 64-Bit Mode.

The program below is a short program that uses a procedure named get\_tableVal to locate a value in a two-dimensional table of 64-bit integers. The program demonstrates how to use base-index-displacement operands in 64-bit mode.

```
0804 ; Two-dimensional arrays in 64-bit mode (TwoDimArrays.asm)
0805 ; Prototypes for procedures
0806 Crlf proto
0807 WriteInt64 proto
0808 ExitProcess proto
0809
0810 .data
0811     table QWORD 1,2,3,4,5      ; Define a two-dimensional array with 3 rows and 5 columns
0812     RowSize = ($ - table)    ; Calculate the size of one row in bytes
0813
0814     QWORD 6,7,8,9,10        ; Define the second row
0815     QWORD 11,12,13,14,15   ; Define the third row
0816
0817 .code
0818 main PROC
0819     ; Set row and column indices
0820     mov rax, 1              ; Row index (zero-based)
0821     mov rsi, 4              ; Column index (zero-based)
0822
0823     call get_tableVal       ; Call the get_tableVal procedure to retrieve the value
0824     call WriteInt64         ; Display the retrieved value
0825     call Crlf               ; Insert a line break
0826
0827     mov ecx, 0
0828     call ExitProcess        ; End the program
0829
0830 main ENDP
```

The `main` procedure performs the following steps:

It loads the row index (1) into the RAX register.

It loads the column index (4) into the RSI register.

It calls the get\_tableVal procedure to get the value at the specified row and column in the table array.

It calls the WriteInt64 procedure to display the value in the RAX register.

It calls the ExitProcess procedure to end the program.

*The get\_tableVal procedure performs the following steps:*

It loads the row offset into the RBX register.

It multiplies the row offset by the size of a quadword to get the offset of the row in the array.

It adds the column offset to the row offset to get the offset of the element in the array.

It loads the value at the specified offset into the RAX register.

It returns from the procedure.

The get\_tableVal procedure uses a base-index-displacement operand to access the element in the array.

The base operand is the RBX register, which contains the row offset.

The index operand is the RSI register, which contains the column offset.

The scale factor is omitted, because the elements in the array are quadwords, and the size of a quadword is 1.

The program you provided demonstrates how to use base-index-displacement operands in 64-bit mode.

Base-index-displacement operands are a powerful tool for accessing arrays in assembly language.

## *Searching and Sorting Algorithms*

### Bubble sort

The bubble sort algorithm is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. The algorithm starts at the beginning of the array and compares the first two elements.

If the first element is greater than the second element, the two elements are swapped. The algorithm then moves on to the next two elements and repeats the process.

The algorithm continues to iterate through the array until it reaches the end.

The following is a pseudocode implementation of the bubble sort algorithm:

```
0839 bubble_sort(array):
0840     for i in range(len(array) - 1):
0841         for j in range(len(array) - i - 1):
0842             if array[j] > array[j + 1]:
0843                 array[j], array[j + 1] = array[j + 1], array[j]
```

The bubble sort algorithm is a simple and straightforward algorithm, but it is not very efficient for large arrays.

This is because the algorithm has to compare every pair of elements in the array for each iteration. For an array of size  $n$ , the bubble sort algorithm has a time complexity of  $O(n^2)$ .

### ***Analysis of bubble sort performance.***

The following table shows the sort times for various array sizes, assuming that 1000 array elements can be sorted in 0.1 second:

Array Size	Time (seconds)
1,000	0.1
10,000	10.0
100,000	1000
1,000,000	100,000 (27.78 hours)

As you can see, the sort time increases quadratically with the array size. This means that the bubble sort algorithm is not very efficient for large arrays.

The bubble sort algorithm is a simple and straightforward sorting algorithm, but it is not very efficient for large arrays.

If you need to sort a large array, you should use a more efficient sorting algorithm, such as the quicksort algorithm or the merge sort algorithm.

---

```
0847 ; Bubble sort algorithm in MASM
0848 section .data
0849     array: dw 5, 3, 2, 1, 4
0850 section .code
0851     start:
0852
0853     mov esi, array
0854     mov ecx, 5 ; length of the array
0855     L1:
0856     mov edi, esi
0857     add edi, 4
0858     L2:
0859     cmp [esi], [edi]
0860     jg L3 ; swap if esi > edi
0861     xchg [esi], [edi]
0862     add esi, 4
0863     cmp esi, array + ecx * 4 - 4
0864     jne L2
0865     loop L1
0866     ; array is now sorted
0867     exit
```

The array section in the data segment initializes an array with values to be sorted.

The code begins by setting up registers, with esi pointing to the start of the array and ecx

containing the length of the array (in this case, 5).

The outer loop, labeled as L1, iterates through the array. This corresponds to the outer loop counter (cx1) in the notes.

Inside the outer loop, the inner loop labeled as L2 is used to compare and swap elements, corresponding to the inner loop counter (cx2) in the notes.

The comparison is done using cmp, and if the current element ([esi]) is greater than the next element ([edi]), a swap is performed using xchg.

The code ensures that the inner loop (L2) iterates through the entire array by comparing esi to the end of the array (array + ecx \* 4 - 4).

After completing the inner loop for a given pass through the array, it uses loop to decrement the outer loop counter and repeats the process until the outer loop counter is equal to 0.

Once the sorting is complete, the array is in ascending order.

C++ version:

```
0870 int BinSearch(int values[], const int searchVal, int count) {  
0871     int first = 0;  
0872     int last = count - 1;  
0873  
0874     while (first <= last) {  
0875         int mid = (last + first) / 2;  
0876  
0877         if (values[mid] < searchVal)  
0878             first = mid + 1;  
0879         else if (values[mid] > searchVal)  
0880             last = mid - 1;  
0881         else  
0882             return mid; // success  
0883     }  
0884  
0885     return -1; // not found  
0886 }
```

Assembly version:

```
; BinarySearch  
; Searches an array of signed integers for a single value.  
; Receives: Pointer to array, array size, search value.  
; Returns: If a match is found, EAX = the array position of the  
; matching element; otherwise, EAX = -1.
```

```
BinarySearch PROC USES ebx edx esi edi,  
    pArray:PTR DWORD,  
    Count:DWORD,  
    searchVal:DWORD  
  
LOCAL first:DWORD,  
      last:DWORD,  
      mid:DWORD  
  
    mov first, 0  
    mov eax, Count  
    dec eax  
    mov last, eax  
    mov edi, searchVal  
    mov ebx, pArray  
  
L1:  
    mov eax, first  
    cmp eax, last  
    jg L5  
  
    mov eax, last  
    add eax, first  
    shr eax, 1  
    mov mid, eax  
  
    mov esi, mid  
    shl esi, 2  
    mov edx, [ebx+esi]  
  
    cmp edx, edi  
    jge L2  
    mov eax, mid  
    inc eax  
    mov first, eax  
    jmp L4
```

```
L2:  
    cmp edx, edi  
    jle L3  
    mov eax, mid  
    dec eax  
    mov last, eax  
    jmp L4  
  
L3:  
    mov eax, mid  
    jmp L9  
  
L4:  
    jmp L1  
  
L5:  
    mov eax, -1  
  
L9:  
    ret  
  
BinarySearch ENDP
```

## Program 2:

```
; Bubble Sort and Binary Search (BinarySearchTest.asm)  
; Bubble sort an array of signed integers and perform a binary search.  
  
INCLUDE Irvine32.inc  
INCLUDE BinarySearch.inc ; Include procedure prototypes
```

```
LOWVAL = -5000
HIGHVAL = +5000
ARRAY_SIZE = 50

.data
array DWORD ARRAY_SIZE DUP(?)  
  
.code
main PROC
    call Randomize ; Initialize random number generator  
  
    ; Fill an array with random signed integers
    INVOKE FillArray, ADDR array, ARRAY_SIZE, LOWVAL, HIGHVAL  
  
    ; Display the array
    INVOKE PrintArray, ADDR array, ARRAY_SIZE
    call WaitMsg  
  
    ; Perform a bubble sort and redisplay the array
    INVOKE BubbleSort, ADDR array, ARRAY_SIZE
    INVOKE PrintArray, ADDR array, ARRAY_SIZE  
  
    ; Demonstrate a binary search
    call AskForSearchVal  
  
    ; Perform the binary search and display the results
    INVOKE BinarySearch, ADDR array, ARRAY_SIZE, eax
    call ShowResults  
  
    exit
main ENDP  
  
; Prompt the user for a signed integer
AskForSearchVal PROC
    .data
```

```

prompt BYTE "Enter a signed decimal integer in the range of -5000 to +5000 to find in the array:
",0
.code
call Crlf
mov edx, OFFSET prompt
call WriteString
call ReadInt
ret
AskForSearchVal ENDP

; Display the resulting value from the binary search
ShowResults PROC
.data
msg1 BYTE "The value was not found.",0
msg2 BYTE "The value was found at position ",0
.code
.IF eax == -1
    mov edx, OFFSET msg1
    call WriteString
.ELSE
    mov edx, OFFSET msg2
    call WriteString
    call WriteDec
.ENDIF
call Crlf
call Crlf
ret
ShowResults ENDP

END main

```

The provided assembly code, `BinarySearchTest.asm`, is a program that demonstrates the use of the bubble sort and binary search functions to work with an array of signed integers. Here's an overview of what the code does:

## Initialization:

It starts by including necessary libraries and defining constants for the minimum and maximum values (LOWVAL and HIGHVAL) and the size of the array (ARRAY\_SIZE).

The code also defines the array array and contains the .data and .code sections.

## Main Procedure (main):

Calls the Randomize function to initialize the random number generator.

Invokes the FillArray procedure to fill the array with random signed integers within the specified range (LOWVAL to HIGHVAL).

Displays the original content of the array using the PrintArray procedure and waits for a message (WaitMsg).

Performs a bubble sort on the array using the BubbleSort procedure to sort the integers in ascending order.

Displays the sorted array using the PrintArray procedure. Prompts the user to enter a signed integer with the AskForSearchVal procedure.

## AskForSearchVal Procedure:

Prompts the user to enter a signed integer within the specified range. Reads the integer entered by the user and returns it in the EAX register.

### ShowResults Procedure:

Displays the result of the binary search.

If the binary search returns -1, indicating that the value was not found, it prints "The value was not found."

If the binary search returns the position of the value in the array, it prints "The value was found at position X," where X is the position.

In summary, this program generates a random array of signed integers, sorts the array using the bubble sort algorithm, and then performs a binary search on the sorted array to find a user-specified value.

It displays the results of the binary search, indicating whether the value was found and, if so, at what position in the array.

## **STRUCTURES**

Skip...

## ***Structures***

A structure in assembly language is defined using the STRUCT and ENDS directives.

Inside the structure, fields are defined using the same syntax as for ordinary variables. Structures can contain virtually any number of fields:

```
0890 name STRUCT  
0891 field-declarations  
0892 name ENDS
```

For example, the following structure defines an employee structure with fields for ID number, last

name, years of service, and an array of salary history values:

```
0900 Employee STRUCT
0901     IdNum BYTE "00000000"
0902     LastName BYTE 30 DUP(0)
0903     Years WORD 0
0904     SalaryHistory DWORD 0,0,0,0
0905 Employee ENDS
```

The first field, IdNum, is a byte-sized field initialized to the string "00000000".

The second field, LastName, is a 30-byte array of bytes initialized to all zeros.

The third field, Years, is a 2-byte word initialized to zero.

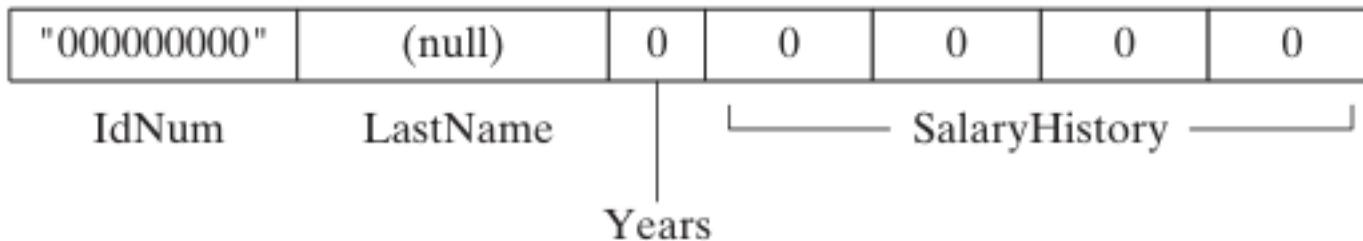
The fourth field, SalaryHistory, is a 4-byte double word initialized to all zeros.

**Note:** The DUP operator is used to initialize the SalaryHistory array to all zeros. The syntax 30 DUP(0) creates an array of 30 bytes, each of which is initialized to zero.

## Structure Memory Layout

In memory, structures are laid out contiguously, with each field starting at the next available address.

The following diagram shows the memory layout of the Employee structure:



Memory Layout:

+-----+	IdNum	(1 byte)
+-----+	LastName	(30 bytes)
+-----+	Years	(2 bytes)
+-----+	SalaryHistory	(4 bytes)

## Using Structures

*Declaring structures:*

```
identifier structureType < initializer-list >
```

The **identifier** is the name of the structure variable.

The **structureType** is the name of the structure type that the variable will be.

The **initializer-list** is a list of values that will be used to initialize the structure fields. The values in the initializer-list must be in the same order as the fields in the structure definition.

If you do not specify an initializer-list, the structure fields will be initialized to their default values.

Example:

```
Employee worker <>
```

This code declares a structure variable named `worker` of type `Employee`. The `<>` (angle brackets) indicate that the structure fields should be initialized to their default values.

### *Referencing Structure Variables:*

To reference a structure variable, you simply use its name. For example, the following code references the `Years` field of the `worker` structure variable:

Once a structure is defined, you can declare variables of that type using the same syntax as for ordinary variables. For example, the following code declares two `employee` variables:

```
mov dx, worker.Years
```

### ***Referencing Structure Members:***

To reference a structure member, you use the following syntax:

```
structureVariable.memberName
```

The `structureVariable` is the name of the structure variable. The `memberName` is the name of the structure member.

Example:

```
mov dx, worker.SalaryHistory
```

This code references the `SalaryHistory` field of the `worker` structure variable.

### ***Using the OFFSET Operator***

The `OFFSET` operator can be used to obtain the address of a structure member. The syntax for the `OFFSET` operator is as follows:

## **OFFSET structureVariable.memberName**

The `structureVariable` is the name of the structure variable. The `memberName` is the name of the structure member.

Example:

```
mov edx, OFFSET worker.LastName
```

This code obtains the address of the `LastName` field of the `worker` structure variable.

=====

*Declaring another structure:*

```
0926 Employee1 Employee  
0927 Employee2 Employee
```

You can then access the fields of these variables using the dot operator `(.)`. For example, the following code sets the `IdNum` field of `Employee1` to 123456789:

```
mov Employee1.IdNum, 123456789
```

You can also access the fields of a structure using the displacement operator ([]). For example, the following code sets the Years field of Employee2 to 5:

```
mov Employee2[2], 5
```

**Note:** The displacement operator is calculated relative to the beginning of the structure. In the previous example, the displacement of the Years field is 2, because it is the second field in the structure.

## Aligning Structure Fields

To achieve the best memory I/O performance, structure members should be aligned to addresses matching their data types.

This means that a byte member should be aligned on a byte boundary, a word member should be aligned on a word boundary, and so on.

If structure members are not aligned correctly, the CPU will require more time to access them.

The following table lists the alignments used by the Microsoft C and C++ compilers and by Win32 API functions:

<b>Member Type</b>	<b>Alignment</b>
BYTE, SBYTE	8-bit (byte) boundary
WORD, SWORD	16-bit (word) boundary
DWORD, SDWORD	32-bit (doubleword) boundary
QWORD	64-bit (quadword) boundary
REAL4	32-bit (doubleword) boundary
REAL8	64-bit (quadword) boundary
Structure	Largest alignment requirement of any member
Union	Alignment requirement of the first member

The key points are:

- Basic data types like BYTE, WORD, DWORD etc have natural alignment to their size boundary.
- 32-bit and 64-bit floating point values align to doubleword and quadword respectively.
- Structures align to the largest alignment needed by any member.

- Unions align to the first member.

Properly aligning data prevents performance issues from unaligned memory accesses. The assembler takes care of alignment automatically based on the data type.

*To align structure fields in assembly language, you can use the ALIGN directive.*

The ALIGN directive sets the address alignment of the next field or variable.

For example, the following code aligns the myVar variable to a doubleword boundary:

```
0940 .data  
0941     ALIGN DWORD  
0942     myVar DWORD ?
```

*You can also use the ALIGN directive to align structure fields.*

For example, the following code correctly defines the Employee structure, using ALIGN to put Years on a word boundary and SalaryHistory on a doubleword boundary:

```
0947 Employee STRUCT  
0948     IdNum BYTE "00000000" ; 9  
0949     LastName BYTE 30 DUP(0) ; 30  
0950     ALIGN WORD ; 1 byte added  
0951     Years WORD 0 ; 2  
0952     ALIGN DWORD ; 2 bytes added  
0953     SalaryHistory DWORD 0,0,0,0 ; 16  
0954 Employee ENDS ; 60 total
```

### *Why is Aligning Structure Fields Important?*

Aligning structure fields is important for performance because the CPU can access aligned data more efficiently.

When structure fields are not aligned, the CPU may have to perform additional instructions to access the data.

This can lead to a significant performance decrease, especially in applications that manipulate large amounts of data.

## *Indirect and Indexed Operands Structs*

### Indirect Operands

Indirect operands are used to address memory locations through the contents of a register. This can be useful for accessing structure members, since it allows you to use a register to store the address of the structure.

To use an indirect operand, you need to use the **PTR operator**. The PTR operator tells the assembler that the operand is indirect.

For example, the following code uses an indirect operand to access the Years field of the worker structure variable:

```
0992 mov esi, OFFSET worker
0993 mov ax, (Employee PTR [esi]).Years
```

This code first moves the address of the worker structure variable into the esi register.

Then, it uses the PTR operator to tell the assembler that the [esi] operand is indirect.

This means that the assembler will load the contents of the esi register and use that as the address of the Years field.

## Indexed Operands

Indexed operands are used to address memory locations by adding an offset to the contents of a register.

This can be useful for accessing arrays of structures, since it allows you to use a register to store the index of the element in the array.

To use an indexed operand, you need to use the [register + offset] syntax.

The register is the register that contains the index of the element in the array.

The offset is the offset of the structure field from the beginning of the structure.

For example, the following code uses an indexed operand to access the Years field of the employee in index position 1 of the department array:

```
1000 .data
1001     department Employee 5 DUP(<>)
1002 .code
1003     mov esi, TYPE Employee
1004     ; index = 1
1005     mov department[esi].Years, 4
```

This code first moves the size of the Employee structure into the esi register.

Then, it uses the [esi] syntax to access the element in index position 1 of the department array. Finally, it sets the Years field of that element to 4.

## Looping through an Array

Indirect and indexed operands can be used to loop through an array of structures.

The following code shows how to loop through the AllPoints array and assign coordinates to each element:



```
1010 ; Loop Through Array
1011 (AllPoints.asm)
1012 INCLUDE Irvine32.inc
1013 NumPoints = 3
1014 .data
1015     ALIGN WORD
1016     AllPoints COORD NumPoints DUP(<0,0>)
1017 .code
1018     main PROC
1019     mov edi, 0
1020     ; array index
1021     mov ecx, NumPoints
1022     ; loop counter
1023     mov ax, 1
1024     ; starting X, Y values
1025     L1:
1026     mov (COORD PTR AllPoints[edi]).X, ax
1027     mov (COORD PTR AllPoints[edi]).Y, ax
1028     add edi, TYPE COORD
1029     inc ax
1030     loop L1
1031     exit
1032     main ENDP
1033 END main
```

This code first moves the value 0 into the edi register. This will be the array index. Then, it moves the value of the NumPoints variable into the ecx register. This will be the loop counter.

Next, the code moves the value 1 into the ax register. This will be the starting X and Y values for the coordinates.

Then, the code enters a loop. Inside the loop, it first uses an indirect operand to access the X field of the element in index position edi of the AllPoints array. It then sets the X field to the value in the ax register.

Next, the code uses an indirect operand to access the Y field of the element in index position edi of the AllPoints array. It then sets the Y field to the value in the ax register.

After that, the code increments the edi register by the size of the COORD structure. This will move the array index to the next element in the array.

Finally, the code increments the ax register by 1. This will increment the X and Y values for the next coordinates.

The loop will continue until the ecx register reaches zero. At that point, the loop will exit and the program will terminate.

### ***Conclusion***

Indirect and indexed operands are powerful tools that can be used to access structure members and arrays of structures.

## *Performance of Aligned and Misaligned Structs*

Aligned structure members can be accessed by the CPU more efficiently than misaligned structure members. This is because the CPU can perform fewer instructions to access aligned data.

The performance impact of misaligned structure members depends on the CPU architecture and the specific structure layout. However, in general, misaligned structure members can lead to a decrease in performance.

The following code shows a simple test to compare the performance of aligned and misaligned structure members:

```
1035 .data
1036     ALIGN DWORD
1037     startTime DWORD ?
1038     ; align startTime
1039     emp Employee <>
1040     ; or: emp EmployeeBad <>
1041 .code
1042     call
1043     GetMSeconds
1044     ; get starting time
1045     mov
1046     startTime,eax
1047     mov
1048     ecx,0FFFFFFFh
1049     ; loop counter
1050     L1:
1051     mov
1052     emp.Years,5
1053     mov
1054     emp.SalaryHistory,35000
1055     loop
1056     L1
1057     call
1058     GetMSeconds
1059     ; get starting time
1060     sub
1061     eax,startTime
1062     call
1063     WriteDec
1064     ; display elapsed time
```

This code gets the system time, executes a loop that accesses structure fields, and calculates the elapsed time. The variable emp can be declared as an Employee or EmployeeBad object.

The Employee structure is aligned, while the EmployeeBad structure is not aligned.

When the code is executed using the Employee structure, the elapsed time is 6141 milliseconds.

When the code is executed using the EmployeeBad structure, the elapsed time is 6203 milliseconds.

The difference in elapsed time is small (62 milliseconds), but it is still measurable.

This suggests that even a small amount of misalignment can have a negative impact on performance.

It is important to note that the performance impact of misaligned structure members can vary depending on the CPU architecture and the specific structure layout.

In some cases, the performance impact may be more significant than the 62 milliseconds observed in the previous example.

### ***Conclusion***

It is generally a good practice to align structure members. This will help to ensure that the CPU can access the structure members efficiently, which can lead to improved performance.

## **Example: Displaying the System Time**

The following is a step-by-step explanation of the program ShowTime.asm in depth:

```
1067 ; Structures (ShowTime.ASM)
1068 INCLUDE Irvine32.inc
1069 .data
1070     sysTime SYSTEMTIME <>
1071     XYPos COORD <10,5>
1072     consoleHandle DWORD ?
1073     colonStr BYTE ":",0
1074 .code
1075     main PROC
1076     ; Get the standard output handle for the Win32 Console.
1077     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
1078     mov consoleHandle,eax
```

The first part of the program defines the data structures that will be used.

The sysTime variable is a SYSTEMTIME structure, which will be used to store the system time.

The XYPos variable is a COORD structure, which will be used to store the cursor position.

The consoleHandle variable is a DWORD variable, which will be used to store the handle to the standard output handle.

The GetStdHandle function is used to retrieve the handle to the standard output handle.

The handle to the standard output handle is used by the SetConsoleCursorPosition and WriteString functions to write data to the console.

```
1083 ; Set the cursor position and get the system time.  
1084 INVOKE SetConsoleCursorPosition, consoleHandle, XYPos  
1085 INVOKE GetLocalTime, ADDR sysTime
```

The next part of the program sets the cursor position to the specified coordinates and retrieves the system time.

The SetConsoleCursorPosition function is used to set the cursor position to the specified coordinates. The GetLocalTime function is used to retrieve the system time.

```
1090 ; Display the system time (hh:mm:ss).
1091 movzx eax,sysTime.wHour
1092 ; hours
1093 call WriteDec
1094 mov edx,OFFSET colonStr
1095 ; ":" 
1096 call WriteString
1097 movzx eax,sysTime.wMinute
1098 ; minutes
1099 call WriteDec
1100 call WriteString
1101 movzx eax,sysTime.wSecond
1102 ; seconds
1103 call WriteDec
1104 call Crlf
```

The next part of the program displays the system time to the console.

The WriteDec function is used to write a decimal number to the console.

The WriteString function is used to write a string to the console.

The Crlf function is used to write a carriage return and line feed to the console.

```
1108 call WaitMsg
1109 ; "Press any key..."
1110 exit
1111 main ENDP
1112 END main
```

The last part of the program displays a message to the console and waits for the user to press a key.

The `WaitMsg` function is used to display a message to the console and wait for the user to press a key.

### ***Conclusion***

The `ShowTime.asm` program is a simple example of how to use structures in assembly language. By using structures, you can group related data together and make your code more readable and maintainable.

## ***Structures Containing Structures***

Structures can contain instances of other structures. This is called a nested structure.

For example, a `Rectangle` structure can be defined in terms of its upper-left and lower-right corners, both `COORD` structures:

```
1116 Rectangle STRUCT
1117     UpperLeft COORD <>
1118     LowerRight COORD <>
1119 Rectangle ENDS
```

This means that a **Rectangle structure** will contain **two COORD structures**, one for the upper-left corner and one for the lower-right corner.

Rectangle variables can be declared without overrides or by overriding individual COORD fields. The following examples show how to declare Rectangle variables:

```
1123 rect1 Rectangle < >
1124 rect2 Rectangle { }
1125 rect3 Rectangle { {10,10}, {50,20} }
1126 rect4 Rectangle < <10,10>, <50,20> >
```

The first two declarations, rect1 and rect2, will create Rectangle variables with the default values for the UpperLeft and LowerRight fields.

The third declaration, rect3, will create a Rectangle variable with the specified values for the UpperLeft and LowerRight fields.

The fourth declaration, rect4, is an alternative way to declare a Rectangle variable with the specified values for the UpperLeft and LowerRight fields.

Once a Rectangle variable has been declared, you can access its fields using the dot notation.

For example, the following code moves the value 10 to the X coordinate of the upper-left corner of the rect1 variable:

```
mov rect1.UpperLeft.X, 10
```

You can also access a structure field using an indirect operand. For example, the following code moves the value 10 to the Y coordinate of the upper-left corner of the structure pointed to by the esi register:

```
1137 mov esi,OFFSET rect1
1138 mov (Rectangle PTR [esi]).UpperLeft.Y, 10
```

The OFFSET operator can be used to return pointers to individual structure fields, including nested fields. For example, the following code moves the value 50 to the X coordinate of the lower-right corner of the rect2 variable:

```
1141 mov edi,OFFSET rect2.LowerRight
1142 mov (COORD PTR [edi]).X, 50
```

Example program:

```
1149 ; Drunkard's Walk
1150 ; Drunkard's walk program simulates a professor's random path in an imaginary grid.
1151 ; The professor starts at coordinates 25, 25 and wanders around the immediate area.
1152
1153 INCLUDE Irvine32.inc
1154
1155 ; Constants
1156 WalkMax = 50      ; Maximum number of steps
1157 StartX = 25       ; Starting X-coordinate
1158 StartY = 25       ; Starting Y-coordinate
1159
1160 ; Define a structure to store the path and number of steps
1161 DrunkardWalk STRUCT
1162     path COORD WalkMax DUP(<0,0>) ; Array of COORD objects for the path
1163     pathsUsed WORD 0                 ; Number of steps taken
1164 DrunkardWalk ENDS
1165
1166 ; Prototypes
1167 DisplayPosition PROTO currX:WORD, currY:WORD
1168
1169 .data
1170 aWalk DrunkardWalk <> ; Create an instance of the DrunkardWalk structure
1171
1172 .code
1173 main PROC
1174     mov esi,OFFSET aWalk ; Get the address of the aWalk structure
1175     call TakeDrunkenWalk ; Simulate the professor's walk
1176     exit
1177 main ENDP
```

```
1179 ;-----
1180 TakeDrunkenWalk PROC
1181 LOCAL currX:WORD, currY:WORD
1182
1183 ; Takes a walk in random directions (north, south, east, west).
1184 ; Receives: ESI points to a DrunkardWalk structure
1185 ; Returns: the structure is initialized with random values
1186 ;-----
1187
1188 pushad ; Preserve registers
1189
1190 ; Initialize EDI with the address of the path array
1191 mov edi,esi
1192 add edi,OFFSET DrunkardWalk.path
1193
1194 mov ecx,WalkMax ; Set loop counter
1195 mov currX,StartX ; Initialize current X-location
1196 mov currY,StartY ; Initialize current Y-location
1197
1198 Again:
1199 ; Insert the current location in the array.
1200 mov ax,currX
1201 mov (COORD PTR [edi]).X,ax ; Store X-coordinate
1202 mov ax,currY
1203 mov (COORD PTR [edi]).Y,ax ; Store Y-coordinate
1204
1205 INVOKE DisplayPosition, currX, currY ; Display the current position
1206
1207 mov eax,4 ; Choose a random direction (0-3)
1208 call RandomRange
```

```
1210 .IF eax == 0
1211     ; North
1212     dec currY
1213 .ELSEIF eax == 1
1214     ; South
1215     inc currY
1216 .ELSEIF eax == 2
1217     ; West
1218     dec currX
1219 .ELSE
1220     ; East (EAX = 3)
1221     inc currX
1222 .ENDIF
1223
1224 add edi,TYPE COORD ; Move to the next COORD
1225 loop Again
1226
1227 Finish:
1228 mov (DrunkardWalk PTR [esi]).pathsUsed, WalkMax ; Set pathsUsed to WalkMax
1229 popad ; Restore registers
1230 ret
1231 TakeDrunkenWalk ENDP
1232
```

```
1233 ;-----  
1234 DisplayPosition PROC currX:WORD, currY:WORD  
1235 ; Display the current X and Y positions.  
1236 ;-----  
1237  
1238 .data  
1239 commaStr BYTE ",",0 ; Comma string  
1240  
1241 .code  
1242 pushad ; Preserve registers  
1243  
1244 ; Display the current X position  
1245 movzx eax,currX  
1246 call WriteDec  
1247  
1248 mov edx,OFFSET commaStr ; Load the comma string  
1249 call WriteString  
1250  
1251 ; Display the current Y position  
1252 movzx eax,currY  
1253 call WriteDec  
1254  
1255 call Crlf ; Move to the next line  
1256  
1257 popad ; Restore registers  
1258 ret  
1259 DisplayPosition ENDP  
1260  
1261 END main
```

I'll provide an explanation of the "Drunkard's Walk" program, which simulates a professor's random path in an imaginary grid:

#### ***Include Directives:***

The program begins with an INCLUDE directive to include the Irvine32.inc library, which provides functions for console I/O and other useful features. Constants:

Constants like WalkMax, StartX, and StartY are defined at the beginning. These constants determine the maximum number of steps in the walk and the starting coordinates of the professor.

#### ***Structure Definition:***

The program defines a structure called DrunkardWalk using the STRUCT and ENDS directives. This structure contains two components: path: An array of COORD objects, which will store the professor's path. pathsUsed: A word to keep track of the number of steps taken.

#### ***Prototypes:***

The DisplayPosition function's prototype is declared. It's used to display the current position (X and Y coordinates) of the professor. Data Section:

The .data section is used to define data, including a variable aWalk, which is an instance of the DrunkardWalk structure.

#### ***Code Section:***

The .code section contains the program's main logic. main Procedure:

The main procedure is the program's entry point. It initializes esi with the address of the aWalk structure. Then, it calls the TakeDrunkenWalk procedure to simulate the professor's walk.

#### ***TakeDrunkenWalk Procedure:***

This procedure simulates the professor's walk. It initializes local variables currX and currY to the starting coordinates and uses a loop to simulate the walk. Inside the loop: The current coordinates are inserted into the path array.

The DisplayPosition function is called to display the current position. A random direction (north, south, east, or west) is chosen for the next step. The loop continues until the maximum number of steps (WalkMax) is reached.

#### ***DisplayPosition Procedure:***

This procedure displays the current X and Y positions with proper formatting. It uses WriteDec to display the coordinates and WriteString to add a comma between X and Y. It also adds a line break with Crlf to format the output.

#### ***Program Termination:***

After the professor's walk is completed, the program sets the pathsUsed field of the awalk structure to WalkMax to indicate the number of steps taken.

#### ***End of Program:***

The END main statement marks the end of the program. In summary, this program uses a structure to simulate a random walk for a professor. It keeps track of the professor's path and displays the X and Y coordinates at each step.

The professor starts at a specified position, and the program randomly selects the next step's direction until the maximum number of steps is reached. The structure helps organize and manage the data related to the walk.

## ***Unions in ASM***

In assembly language, unions are used to group fields together at the same offset.

The size of a union is determined by its largest field.

To declare a union, you use the **UNION** and **ENDS** directives, like this:

```
1265 myUnion UNION
1266     field1 DWORD 0
1267     field2 WORD 0
1268     field3 BYTE 0
1269 myUnion ENDS
```

You don't have to indent.

Fields in a union follow the same rules as structures, but each field can have only one initializer.

## Nesting Unions in Structures

You can nest a union inside a structure in two ways:

Declare the union inside the structure:

```
1277 MyStructure STRUCT
1278     UNION MyUnion
1279         field1 DWORD ?
1280         field2 WORD ?
1281         field3 BYTE ?
1282     ENDS
1283 MyField DWORD ?
1284 MyStructure ENDS
```

Use the union's name in a declaration inside the structure:

```
1290 MyStructure STRUCT
1291     MyUnion MyField
1292         field1 DWORD ?
1293         field2 WORD ?
1294         field3 BYTE ?
1295 MyStructure ENDS
```

### Declaring and Using Union Variables:

Declaring a union variable is similar to declaring a structure variable.

However, you can only have one initializer:

```
myVar MyUnion <1234h>
```

To use a specific field of a union variable, you supply the field's name. For example:

```
mov myVar.field1, eax  
mov myVar.field2, bx
```

## Unions Can Contain Structures

Unions can also contain structures.

For instance, you can have a union that can hold different types of structures based on the situation. Example - KEY\_EVENT\_RECORD:

In the Windows API, there's a structure called KEY\_EVENT\_RECORD that contains a union named uChar. It's used to handle keyboard input events.

Depending on the situation, it can contain either a Unicode character or an ASCII character.

The union lets you use the appropriate character based on the event type.

Unions and structures are helpful for organizing data in assembly language programs.

Unions allow you to store data of different sizes in the same memory space, which can be useful in various situations.

## **Macros in ASM**

Certainly, I'll simplify and clarify the information about macros in assembly language.

Overview of Macros:

A macro is a named block of assembly language statements. It can be called multiple times in a program.

When you invoke a macro, a copy of its code is inserted directly into the program where it was invoked.

This is known as inline expansion, and there's no actual CALL instruction involved. Placement of Macros:

Macro definitions are usually placed at the beginning of the program's source code or in a separate file included with an INCLUDE directive.

Macros are expanded during the assembler's preprocessing step.

The assembler replaces every invocation of the macro with a copy of the macro's source code. If a program defines a macro but never calls it, the macro code does not appear in the compiled program.

Defining Macros:

A macro is defined using the **MACRO** and **ENDM** directives.

The syntax is as follows:

```
1311 macroname MACRO parameter-1, parameter-2...
1312 statement-list
1313 ENDM
```

Parameters are named placeholders for arguments passed to the macro.

Parameters can be any text, integers, variable names, or other values. The preprocessor treats them as text.

Parameters don't have type information; type checking occurs during assembly.

#### Example - mPutchar Macro:

The mPutchar macro takes a single parameter named char. It pushes eax, moves the character char into al, and calls the WriteChar procedure.

Finally, it pops eax to restore the original value.

When you use this macro, it replaces the macro invocation with these statements, and the character is displayed on the console.

In essence, macros are a way to create reusable blocks of code that can be inserted directly into your program. They make your code more modular and easier to maintain.

```
1316 mPutchar MACRO char
1317     push eax          ; Push the value in the eax register
1318     mov al, char      ; Move the character 'char' into the al register
1319     call WriteChar   ; Call the WriteChar procedure to display the character
1320     pop eax          ; Pop the original value back into eax
1321 ENDM
1322
1323
1324 ;You'd use it like this:
1325
1326
1327 .code
1328 ; ... Your code here ...
1329 mPutchar 'A' ; Display the character 'A' on the console
1330 ; ... Your code here ...
```

### Invoking Macros:

To use a macro, insert its name in your program, and you can provide arguments.

The syntax for invoking a macro is: **macroname argument-1, argument-2, ...** macroname must be a previously defined macro name in your source code.

Each argument replaces a parameter in the macro. The order of arguments must match the order of parameters.

You can pass a different number of arguments than the number of parameters in the macro. If you pass too many arguments, the assembler warns you.

If you pass too few, the unfilled parameters remain empty.

For example, if you have a macro called mPutchar that displays characters on the console, you can invoke it like this: mPutchar 'A'.

The macro call is expanded to the code that displays 'A' on the console.

### **Debugging Macros:**

Debugging programs with macros can be challenging.

Check the listing file (.LST) after assembling to ensure that each macro is expanded as intended.

In the Visual Studio debugger, you can view the disassembly to see how each macro call is expanded into actual code. This can help with debugging.

### **Additional Macro Features:**

You can use the REQ qualifier to specify that a macro parameter is required.

If the macro is invoked without an argument for a required parameter, the assembler shows an error message.

For example, in the mPutchar macro, you can specify that the char parameter is required:

## mPutchar MACRO char:REQ

To exclude comments from appearing in macro expansions, use double semicolons (;;).

This way, comments within a macro definition won't show up when the macro is expanded. For example, you can add comments like this: **;; reminder: char must contain 8 bits** within your macro, and they won't appear in the expanded code.

In summary, invoking macros is like using predefined functions in your program. You provide arguments to the macro, which then replaces them with its predefined instructions.

Debugging macros can be done by checking the listing file and using the debugger. Additionally, you can specify required parameters and control which comments appear in macro expansions.

```
1340 mPutchar MACRO char:REQ
1341     push eax          ; Save the value in the eax register
1342     mov al, char       ; Move the character from 'char' to the al register
1343     call WriteChar     ; Call the WriteChar procedure to display the character
1344     pop eax           ; Restore the original value in eax
1345 ENDM
```

This macro essentially saves the value in eax, displays the character specified by the char argument, and then restores eax to its original value. Remember that the REQ qualifier indicates that the char parameter is required when invoking the macro.

## ***ECHO and LOCAL Directives***

### **ECHO directive**

The ECHO directive writes a string to standard output as the program is assembled. This can be useful for debugging or for printing informative messages to the user.

Example:

```
1349 mPutchar MACRO char:REQ
1350 ECHO Expanding the mPutchar macro
1351 push
1352 eax
1353 mov
1354 al,char
1355 call
1356 WriteChar
1357 pop
1358 eax
1359 ENDM
```

When this macro is expanded, the assembler will print the message "Expanding the mPutchar macro" to standard output.

## LOCAL Directive

Macros can often contain labels. If a macro is invoked multiple times in a program, the labels in the macro will be redefined each time. This can lead to errors if the same label is used in multiple places in the program.

The LOCAL directive can be used to avoid this problem. When a label is declared as LOCAL, the assembler will generate a unique name for the label each time the macro is expanded.

Example:

```
1363 makeString MACRO text
1364 LOCAL string
1365 .data
1366     string BYTE text,0
1367 ENDM
```

In this example, the makeString macro declares a variable named string. If the macro is invoked multiple times, the assembler will generate a unique name for the string variable each time.

To use the LOCAL directive, simply add the LOCAL keyword before the label declaration. For example:

**LOCAL label**

The LOCAL keyword can be used for any type of label, including code labels and data labels.

The ECHO and LOCAL directives can be useful for debugging macros and avoiding label redefinition errors.

## Macros Containing Code and Data

Macros can contain both code and data. This means that a macro can define both instructions and variables. Example:

```
1375 mWrite MACRO text
1376     LOCAL string    ; Define a local label
1377     .data           ; Switch to the data section
1378     string BYTE text, 0    ; Define the string
1379
1380     .code           ; Switch back to the code section
1381     push edx       ; Push edx onto the stack
1382     mov edx, OFFSET string    ; Load the address of the string into edx
1383     call WriteString      ; Call the WriteString procedure with the address of the string
1384     pop edx         ; Restore the value of edx
1385 ENDM
```

This macro defines a function called mWrite() that displays a string on the console. The macro takes a single parameter, text, which is the string to display.

The macro first defines a local label named string. This label is used to identify the string in the .data section of the program.

The macro then defines the code for the `mWrite()` function. This code pushes the address of the string onto the stack, calls the `WriteString()` function to display the string, and then pops the address of the stack.

## Nested Macros

A macro invoked from another macro is called a nested macro. When the assembler's preprocessor encounters a call to a nested macro, it expands the macro in place.

This means that the parameters passed to the enclosing macro are passed directly to its nested macros. Example:

```
1390 mWriteln MACRO text
1391     mWrite text ; Invoke the mWrite macro with the given text
1392     call Crlf      ; Call the Crlf procedure to add a new line
1393 ENDM
```

This `mWriteln` macro simplifies the process of writing a line of text. It calls the `mWrite` macro to display the provided text and then adds a new line by calling the `Crlf` procedure.

This macro defines a function called `mWriteln()` that writes a string to the console and appends an end of line. The macro invokes the `mWrite()` macro to display the string, and then calls the `Crlf()` procedure to append an end of line.

The `text` parameter is passed directly to the `mWrite()` macro. Example:

```
mWriteln "My Sample Macro Program"
```

When the assembler expands this statement, it will first expand the `mWriteln()` macro. This will result in the following code:

```
1406 mWrite "My Sample Macro Program"  
1407 call Crlf
```

This code will then be expanded to the following instructions:

```
1412 push edx          ; Push the value in the edx register onto the stack  
1413 mov edx, OFFSET ??0002 ; Load the address of label ??0002 into edx  
1414 call WriteString ; Call the WriteString procedure with the address in edx  
1415 pop edx           ; Pop the value from the stack back into edx  
1416 call Crlf         ; Call the Crlf procedure to add a new line
```

### *Tips for Creating Macros:*

Keep macros short and simple. This will make them easier to understand and maintain.

Use a modular approach when creating macros. This means breaking down complex macros into smaller, simpler macros. This will make the macros more reusable and flexible.

Use the LOCAL directive to avoid label redefinition errors.

Document your macros clearly. This will help other programmers understand how to use them.

Macros can be a powerful tool for simplifying and improving code. However, it is important to use them carefully and to follow the tips above to avoid problems.

### Irvine32/64 library and Macros:

```
1420 ; Using the Book's Macro Library (32-Bit Mode Only)
1421 ; To enable the library, include the following line after your existing INCLUDE:
1422 INCLUDE Macros.inc
1423
1424 ; Macros in the Macros.inc Library:
1425 ; mDump - Displays a variable using its name and default attributes.
1426 ; mDumpMem - Displays a range of memory.
1427 ; mGotoxy - Sets the cursor position in the console window buffer.
1428 ; mReadString - Reads a string from the keyboard.
1429 ; mShow - Displays a variable or register in various formats.
1430 ; mShowRegister - Displays a 32-bit register's name and contents in hexadecimal.
1431 ; mWrite - Writes a string literal to the console window.
1432 ; mWriteSpace - Writes one or more spaces to the console window.
1433 ; mWriteString - Writes a string variable's contents to the console window.
1434
1435 ; Example usage of the mDumpMem macro:
1436 ; mDumpMem OFFSET array, LENGTHOF array, TYPE array
1437 ; Displays the 'array' variable using its default attributes.
```

In summary, the "Macros.inc" library contains various macros that simplify common tasks in assembly programming, such as displaying variables, manipulating cursor positions, reading from the keyboard, and more.

You can enable this library by including it in your program, and then you can use the provided macros to streamline your code.

=====

Let's expand on them a bit:

## **mDumpMem macro**

The **mDumpMem macro** is used to display a block of memory in the console window. It requires three arguments: the memory offset, the number of items to display, and the size of each memory component. The macro internally invokes the DumpMem library procedure, passing the three arguments to it (ESI, ECX, and EBX, respectively). Here's a cleaned-up version of the explanation:

```
1440 mDumpMem MACRO address:REQ, itemCount:REQ, componentSize:REQ
1441
1442     push ebx
1443     push ecx
1444     push esi
1445
1446     mov esi, address
1447     mov ecx, itemCount
1448     mov ebx, componentSize
1449
1450     call DumpMem
1451
1452     pop esi
1453     pop ecx
1454     pop ebx
1455
1456 ENDM
```

The `mDumpMem` macro displays a memory dump using the `DumpMem` procedure.

It takes 3 parameters:

- `address` - Offset of memory block to dump
- `itemCount` - Number of components to display
- `componentSize` - Size in bytes of each component

It avoids passing EBX, ECX, ESI to prevent corrupting registers used internally.

Example usage:

```
| mDumpMem OFFSET array, LENGTHOF array, TYPE array
```

This would display the contents of the array memory block, with number of items and size based on the array's declared size and type.

It shows both the memory offset and hex dump of the values, with the componentSize determining the format.

So `mDumpMem` provides a convenient way to dump memory blocks for debugging.

---

## **mDump macro**

The `mDump` macro displays the address and hexadecimal contents of a variable.

It takes two parameters:

- `varName` - Name of the variable to dump
- `useLabel` - Optional label to display

The size and format match the variable's declared type.

If `useLabel` is passed a non-blank value, it will print the variable name.

```
1463 mDump MACRO varName:REQ, useLabel  
1464  
1465 call Crlf  
1466  
1467 IFNB <useLabel>  
1468     mWrite "Variable name: &varName"  
1469 ENDIF  
1470  
1471 mDumpMem OFFSET varName, LENGTHOF varName, TYPE varName  
1472  
1473 ENDM
```

The `&varName` substitution operator inserts the actual variable name into the string.

`IFNB` checks if `useLabel` was passed a non-blank value.

`mDumpMem` is called to print the hex dump using the variable's attributes.

So `mDump` provides a convenient way to quickly dump variables for debugging.

=====

## mGotoxy macro

The mGotoxy macro positions the cursor in the console window.

It takes two BYTE parameters for the X and Y coordinates.

Avoid passing DH and DL to prevent register conflicts.

```
1477 mGotoxy MACRO X:REQ, Y:REQ  
1478  
1479     push edx  
1480  
1481     mov dh,Y  
1482     mov dl,X  
1483  
1484     call Gotoxy  
1485  
1486     pop edx  
1487  
1488 ENDM
```

**Register conflict example:**

If DH and DL are passed as arguments, the expanded code would be:

```
1492 push edx
1493 mov dh,d1 ; Y value overwritten
1494 mov dl,dh ; X value now incorrect
1495 call Gotoxy
1496 pop edx
```

DH is overwritten by DL before it can be copied to DL.

So the macro documentation warns not to pass DH/DL to avoid this problem.

In summary, mGotoxy sets the cursor position, but care must be taken with register arguments to prevent conflicts. The macro code/docs make this clear.

=====

## **mReadString macro**

The mReadString macro reads keyboard input into a buffer.

It takes one parameter:

- **varName** - Name of the buffer to store the input string

Avoids using ECX and EDX internally to prevent register conflicts.

```
1500 mReadString MACRO varName:REQ  
1501  
1502     push ecx  
1503     push edx  
1504  
1505     mov edx,OFFSET varName  
1506     mov ecx,SIZEOF varName  
1507  
1508     call ReadString  
1509  
1510     pop edx  
1511     pop ecx  
1512  
1513 ENDM
```

```
1517 firstName BYTE 30 DUP(?)  
1518  
1519 mReadString firstName
```

This would call ReadString to input a string from the keyboard into the firstName buffer.

So mReadString encapsulates the details of calling ReadString to simplify inputting strings.

=====

## mShow macro

The mShow macro displays a register or variable's name and contents in different formats. It is useful for debugging.

mShow takes a register/variable name followed by format specifiers:

- H - Hexadecimal
- D - Decimal (unsigned by default)
- I - Signed decimal
- B - Binary
- N - Newline

You can combine multiple formats like "HDB" and add multiple newlines. The default is "HIN".

Display AX in multiple formats:

```
1524 mov ax,4096
1525 mShow AX          ; HIN (hex, signed decimal, newline)
1526 mShow AX,DBN      ; Decimal, binary, newline
```

```
1529 ;Output
1530 AX = 1000h = 4096d
1531 AX = 4096d = 00010000 00000000b
```

Display multiple registers on one line:

```
1540 mov ax,1  
1541 mov bx,2  
1542 mov cx,3  
1543 mov dx,4  
1544  
1545 mShow AX,D  
1546 mShow BX,D  
1547 mShow CX,D  
1548 mShow DX,DN
```

```
1552 ;Output  
1553 AX = 1d BX = 2d CX = 3d DX = 4d
```

Display variable in decimal with newline:

```
1555 mydword DWORD ?  
1556  
1557 mShow mydword,DN
```

---

**mShowRegister macro**

`mShowRegister` displays a 32-bit register's name and hexadecimal contents. It is useful for debugging.

It takes two parameters:

- `regName` - String to display as the register name
- `regValue` - The 32-bit register value

### `mShowRegister EBX, ebx`

Displays:

EBX=7FFD9000

### Implementation

The macro does the following:

- Declares a local string variable `tempStr` to hold the label
- Pushes `EAX` and `EDX` to preserve registers used internally
- Builds the label string with the name and '='
- Calls `WriteString` to display the label
- Pops `EDX` to restore it
- Moves the register value into `EAX`
- Calls `WriteHex` to display the hex value
- Pops `EAX` to restore it

Some key points:

- Local string variable avoids modifying caller's code
- Register pushing/popping prevents corruption

- WriteString and WriteHex handle the output
- Substitution inserts regName and regValue from caller

So mShowRegister encapsulates the details of displaying a register's name and value for debugging. The caller simply specifies the name and register.

=====

The mWriteSpace macro writes one or more spaces to the console window. It takes an optional integer parameter specifying the number of spaces to write. The default value is one.

The mWriteString macro writes the contents of a string variable to the console window. It takes a single parameter, which is the name of the string variable to write.

Here is a more detailed explanation of how these macros work:

## **mWriteSpace**

The mWriteSpace macro defines a local label named spaces. This label is used to identify a string of spaces in the .data section of the program.

The macro then defines the code for the mWriteSpace() function. This code pushes the address of the spaces string onto the stack, calls the WriteString() function to display the string, and then pops the address off the stack.

Example:

```
mWriteSpace 5
```

When the assembler expands this statement, it will first expand the `mWriteSpace()` macro. This will result in the following code:

```
1571 push edx
1572 mov edx,OFFSET spaces
1573 call WriteString
1574 pop edx
```

This code will then be expanded to the following instructions:

```
1577 push edx
1578 mov edx,5
1579 rep movsb
1580 pop edx
```

The `rep movsb` instruction will copy 5 bytes from the source (the EDX register) to the destination (the console window). The source will be incremented by one after each byte is copied, and the destination will be incremented by one after each byte is copied.

=====

```
mWriteString
```

The `mWriteString` macro defines a function called `mWriteString()` that writes the contents of a string variable to the console window. The macro takes a single parameter, which is the name of the string variable to write.

The macro first saves the EDX register on the stack. This is because the `WriteString()` function uses the EDX register to store the address of the string to write.

The macro then loads the address of the string variable into the EDX register. Finally, the macro calls the `WriteString()` function and then pops the EDX register from the stack.

Example:

```
1583 .data
1584     str1 BYTE "Please enter your name: ",0
1585 .code
1586     mWriteString str1
```

When the assembler expands this statement, it will first expand the `mWriteString()` macro. This will result in the following code:

```
1590 push edx
1591 mov edx,OFFSET str1
1592 call WriteString
1593 pop edx
```

This code will then be expanded to the following instructions:

```
1595 push edx
1596 mov edx,OFFSET str1
1597 call WriteString
1598 pop edx
```

The WriteString() function will then write the contents of the str1 variable to the console window.

## Conclusion

The mWriteSpace and mWriteString macros can be useful for simplifying code and making it more readable. For example, the following code:

```
WriteString("Please enter your name: ");
```

can be rewritten using the macros as follows:

```
mWriteString "Please enter your name: "
```

Here is the Wraps.asm program explained with the code rewritten for clarity:

### Purpose:

Demonstrates use of wrapper macros for common procedures.

**Macros used:**

- mGotoxy - Set cursor position
- mWrite - Display formatted output
- mWriteString - Display a string
- mReadString - Input a string
- mDumpMem - Hex dump memory

```
1610 INCLUDE Irvine32.inc
1611 INCLUDE Macros.inc
1612 .data
1613     array DWORD 1,2,3,4,5,6,7,8
1614     firstName BYTE 31 DUP(?)
1615     lastName BYTE 31 DUP(?)
1616 .code
1617     main PROC
1618         mGotoxy 0,0          ; Position cursor
1619         ; Display heading
1620         mWrite <"Sample Macro Program",0dh,0ah>
1621         ; Input first name
1622         mGotoxy 0,5
1623         mWrite "Please enter your first name: "
1624         mReadString firstName
1625         ; Input last name
1626         call Crlf
1627         mWrite "Please enter your last name: "
1628         mReadString lastName
1629         ; Display name
1630         call Crlf
1631         mWrite "Your name is "
1632         mWriteString firstName
1633         mWriteSpace
1634         mWriteString lastName
1635         ; Display array
1636         call Crlf
1637         mDumpMem OFFSET array, LENGTHOF array, TYPE array
1638         exit
1639     main ENDP
1640 END main
```

**Here are the review questions rewritten:**

1. When a macro is invoked, does the assembler automatically insert CALL and RET instructions in the generated code? Explain.
2. Where is macro expansion handled - at assembly time or runtime?
3. What is the main advantage of using macros with parameters compared to macros without parameters?
4. Can a macro definition appear before or after the code that invokes it, as long as it is in the code segment? Explain.
5. How does replacing a procedure call with a macro invocation typically affect code size if the macro is called multiple times? Explain.
6. Can a macro contain data definitions like DW and DB? Explain.

**Here are answers to the review questions:**

1. False - CALL and RET instructions are not automatically inserted when a macro is invoked. The macro expansion is inserted directly into the code.
2. True - Macro expansion is handled by the assembler's preprocessor before the code is assembled.
3. Macros with parameters are more flexible since they can accept arguments from the caller. This avoids having to modify the macro definition for different uses.
4. True - As long as it is in the code segment, a macro can appear anywhere, before or after its usage. The assembler handles macros separately during preprocessing.
5. True - If a macro is invoked multiple times, the code will be expanded/duplicated each time, increasing the overall code size compared to calling a single procedure.
6. False - Macros can contain data definitions, though the data is inserted wherever the macro is expanded.

# ***Conditional Assembly Directives***

Here is an organized summary of the key conditional assembly directives:

- **IF, ELSE, ENDIF** - Conditionally assemble code blocks based on a condition
- **IRP, IRPC** - Repeat a code block for each parameter value
- **REPT** - Repeat a block a specified number of times
- **WHILE, ENDW** - Repeat a block while a condition is true
- **EXITM** - Exit macro expansion early
- **LOCAL** - Declare local macro symbols

These allow macros to contain conditional logic, repetition, local symbols, early exit, and other advanced logic.

The important thing is that they work at assembly time, not runtime. The assembler evaluates them to determine what code to include or exclude in the final program. This makes macros much more powerful than simple text substitution.

## **Conditional-assembly directives**

Conditional-assembly directives can be used to control the assembly of code based on certain conditions.

This can be useful for creating macros that are more flexible and adaptable. The general syntax for conditional-assembly directives is as follows:

```
1645 IF condition  
1646     statements  
1647  
1648 [ELSE  
1649     statements]  
1650  
1651 ENDIF
```

The IF directive checks the condition specified in its argument. If the condition is true, the statements following the IF directive are assembled.

If the condition is false, the statements following the ELSE directive are assembled (if one is present).

Here is a table of the most common conditional-assembly directives:

Directive	Description
IF <i>expression</i>	Permits assembly if the value of <i>expression</i> is true (nonzero). Possible relational operators are LT, GT, EQ, NE, LE, and GE.
IFB < <i>argument</i> >	Permits assembly if <i>argument</i> is blank. The argument name must be enclosed in angle brackets (<>).
IFNB < <i>argument</i> >	Permits assembly if <i>argument</i> is not blank. The argument name must be enclosed in angle brackets (<>).
IFIDN < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are equal (identical). Uses a case-sensitive comparison.
IFIDNI < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are equal. Uses a case-insensitive comparison.
IFDIF < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are unequal. Uses a case-sensitive comparison.
IFDIFI < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are unequal. Uses a case-insensitive comparison.
IFDEF <i>name</i>	Permits assembly if <i>name</i> has been defined.
IFNDEF <i>name</i>	Permits assembly if <i>name</i> has not been defined.
ENDIF	Ends a block that was begun using one of the conditional-assembly directives.
ELSE	Terminates assembly of the previous statements if the condition is true. If the condition is false, ELSE assembles statements up to the next ENDIF.
ELSEIF <i>expression</i>	Assembles all statements up to ENDIF if the condition specified by a previous conditional directive is false and the value of the current expression is true.
EXITM	Exits a macro immediately, preventing any following macro statements from being expanded.

Table 2:

Directive	Description
.IF	Assembles the following statements if the condition is true.
.ELSE	Assembles the following statements if the condition is false (and no previous .ELSE directive was true).
.ENDIF	Marks the end of a conditional-assembly block.

The following example shows how to use conditional-assembly directives to create a macro that can be used to print a message to the console:

```
1655 ; PrintMessage Macro
1656 ; Displays a message using the mWrite macro if the message is provided.
1657 ; Parameters:
1658 ; - message: The message to be displayed
1659 ;
1660 ; Usage example:
1661 ; PrintMessage "Hello, world!"
1662
1663 PrintMessage MACRO message
1664     .IF message
1665         mWrite message
1666     .ENDIF
1667 ENDM
```

This macro takes a single parameter, `message`, which is the message to be printed to the console.

The IF directive checks to see if the `message` parameter is empty. If it is not empty, the `mWrite` macro is used to print the message to the console.

Here is an example of how to use the `PrintMessage` macro:

```
PrintMessage "Hello, world!"
```

This will print the message "Hello, world!" to the console.

Conditional-assembly directives can be used to create macros that are more flexible and adaptable.

For example, the PrintMessage macro could be modified to support different types of messages, such as error messages, warning messages, and informational messages.

It is important to note that conditional-assembly directives are evaluated at assembly time, not at runtime.

This means that the condition specified in a conditional-assembly directive must be constant and cannot be evaluated based on runtime values.

## Checking for Missing Arguments

To prevent errors caused by missing arguments in a macro, you can use conditional directives like IFB (if blank) or IFNB (if not blank) to check if an argument is provided. Here's a simplified explanation and code example:

```
1675 mWriteString MACRO string
1676     IFB <string>
1677         ECHO -----
1678         ECHO * Error: Parameter missing in mWriteString
1679         ECHO * (No code generated)
1680         ECHO -----
1681         EXITM
1682     ENDIF
1683
1684     push edx
1685     mov edx, OFFSET string
1686     call WriteString
1687     pop edx
1688 ENDM
```

In assembly language macros, it's important to handle missing arguments to avoid errors during macro expansion. A missing argument can lead to invalid instructions when the macro is expanded.

To address this issue, you can use conditional directives:

IFB (if blank) returns true if a macro argument is blank, meaning it's not provided. IFNB (if not blank) returns true if a macro argument is not blank, indicating it's provided.

Let's take an example where we have a macro called `mWriteString`. This macro displays a string using the `WriteString` procedure, but it needs a string argument to work correctly.

In this example, if the string argument is missing, the macro will display an error message during assembly and won't generate any code. This helps ensure that your macros are used correctly with the required arguments.

## Default Argument Initializers

Macros can have default argument initializers.

This means that if a macro argument is missing when the macro is called, the default argument will be used instead.

The syntax for a default argument initializer is as follows:

**paramname := <argument>**

Spaces before and after the operators are optional.

For example, the following macro has a default argument initializer for the text parameter:

```
1697 ; mWriteln Macro
1698 ; This macro writes a string to the console followed by a carriage return and line feed (CRLF).
1699
1700 mWriteln MACRO text:=<" ">
1701     ; Check if a text argument is provided. If not, use a space as the default.
1702     IFB <text>
1703         mWrite " " ; Display a space if no text is provided
1704     ELSE
1705         mWrite text ; Display the provided text
1706     ENDIF
1707
1708     ; Call the Crlf procedure to move to the next line.
1709     call Crlf
1710 ENDM
```

This macro first checks if a text argument is provided. If it's missing, it uses a space as the default value.

Then, it calls the `mWrite` macro to display the text and adds a line break by invoking the `Crlf` procedure.

If the `mWriteln` macro is called with no arguments, the default argument initializer (" ") will be used for the `text` parameter.

This will cause the macro to print a single space followed by an end of line to the console.

## Boolean Expressions

The assembler allows the following relational operators to be used in constant Boolean expressions:

```
1715 <:      ;Less than
1716 >:      ;Greater than
1717 ==:     ;Equal to
1718 !=:     ;Not equal to
1719 <=:     ;Less than or equal to
1720 >=:     ;Greater than or equal to
```

These operators can be used in conjunction with the IF and other conditional directives to control the assembly of code.

For example, the following code uses the IF directive to check if the value of the x variable is greater than 10:

```
1724 IF x > 10
1725     ; Assemble the following code if x is greater than 10
1726     ; ...
1727 ENDIF
```

In this code snippet, the instructions within the IF block will be assembled only if the condition  $x > 10$  is true.

If  $x$  is indeed greater than 10, the code within the IF block will be processed; otherwise, it will be skipped.

## IF, ELSE, and ENDIF directives

The **IF, ELSE, and ENDIF directives** are used to control the assembly of code based on certain conditions.

The IF directive checks the condition specified in its argument. If the condition is true, the statements following the IF directive are assembled.

If the condition is false, the statements following the ELSE directive are assembled (if one is present).

The ENDIF directive marks the end of a conditional-assembly block.

The following is an example of how to use the IF, ELSE, and ENDIF directives:

```
1732 IF userAge > 18
1733     ; Assemble this code if the user's age is greater than 18
1734     mov edx, OFFSET adultMessage
1735 ELSE
1736     ; Assemble this code if the user's age is 18 or younger
1737     mov edx, OFFSET minorMessage
1738 ENDIF
```

In this example, the code chooses different messages to display based on whether the userAge is greater than 18 or not. If it's greater than 18, it displays the "adultMessage," otherwise, it displays the "minorMessage."

## mGotoxyConst macro

The mGotoxyConst macro is an example of a macro that uses the IF, ELSE, and ENDIF directives to control the assembly of code.

The macro takes two parameters, X and Y, which must be constant expressions.

The macro checks to see if the values of X and Y are within the valid ranges of 0 to 79 and 0 to 24, respectively.

If either of the values is outside of the valid range, the macro displays a warning message and sets a flag.

If the flag is set, the macro exits. Otherwise, the macro assembles the code to move the cursor to the specified coordinates.

The following is an example of how to use the mGotoxyConst macro:

**mGotoxyConst 10, 20**

This will move the cursor to column 10, row 20.

The IF, ELSE, and ENDIF directives can be used to create more complex and flexible macros.

## The IFIDN and IFIDNI Directives

```
1748 mReadBuf MACRO bufferPtr, maxChars
1749 ;
1750 ; Reads from the keyboard into a buffer.
1751 ; Receives: offset of the buffer, count of the maximum
1752 ; number of characters that can be entered. The
1753 ; second argument cannot be edx or EDX.
1754 ;-----
1755 IFIDNI <maxChars>,<EDX> ; Check if the second argument is EDX (case-insensitive)
1756     ECHO Warning: Second argument to mReadBuf cannot be EDX
1757     ECHO ****
1758     EXITM ; Exit the macro
1759 ENDIF
1760
1761 push ecx
1762 push edx
1763 mov edx, bufferPtr ; Move the buffer offset to EDX
1764 mov ecx, maxChars ; Move the maximum character count to ECX
1765 call ReadString
1766 pop edx
1767 pop ecx
1768 ENDM
```

This code defines a macro `mReadBuf` that reads from the keyboard into a buffer. It checks if the second argument, `maxChars`, is equal to `EDX` in a case-insensitive manner using `IFIDNI`.

If it is, it displays a warning message and exits the macro. If not, it proceeds to read from the keyboard into the specified buffer.

## **IFIDNI:**

IFIDNI is a conditional assembly directive.

It is used to check if two symbols (or macro parameter names) are equal in a case-insensitive manner.

In the code, it's used to compare maxChars with EDX to ensure that the second argument is not equal to EDX.

If the comparison is true, it means the second argument is EDX, and a warning message is displayed.

If the comparison is false, the code proceeds to read input into the buffer.

## **ECHO:**

ECHO is used to write a message to the console during assembly. In the code, it's used to display a warning message when the second argument is EDX.

## **EXITM:**

EXITM is used to exit a macro.

In the code, if the second argument is EDX, the macro execution is halted using EXITM. push and pop:

These instructions are used to push and pop values onto and from the stack, respectively.

In the code, `push ecx` and `push edx` are used to save the values of registers ECX and EDX on the stack. This is done to protect the original values of these registers.

Later, `pop edx` and `pop ecx` are used to restore the original values of these registers before exiting the macro.

#### **mov:**

The `mov` instruction is used to move data between registers and memory locations.

In the code, `mov edx, bufferPtr` is used to load the address of the buffer into the EDX register, and `mov ecx, maxChars` is used to load the value of `maxChars` into ECX before calling `ReadString`.

Overall, the code ensures that the second argument passed to the `mReadBuf` macro is not EDX and issues a warning if it is.

If the argument is not EDX, the code continues to read input into the specified buffer. The use of conditional assembly directives and stack manipulation ensures proper execution of the macro.

## **Summing a matrix row**

```
1783 mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
1784     push ebx
1785     push ecx
1786     push esi
1787
1788     mov eax, index
1789     mov ebx, arrayOffset
1790     mov ecx, rowSize
1791     shr ecx, (TYPE eltType / 2) ; Adjust rowSize for the element type
1792
1793     add ebx, eax ; Calculate the row offset
1794     mov eax, 0      ; Initialize accumulator
1795     mov esi, 0      ; Initialize column index
1796
1797 L1:
1798     IFIDNI <eltType>, <DWORD>
1799         mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
1800     ELSE
1801         movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
1802     ENDIF
1803
1804     add eax, edx
1805     inc esi
1806     loop L1
1807
1808     pop esi
1809     pop ecx
1810     pop ebx
1811 ENDM
```

**Here's what the macro does:**

It takes four parameters: index, arrayOffset, rowSize, and eltType.

The rowSize is adjusted based on the eltType parameter, ensuring it represents the number of elements in each row.

The macro initializes the registers and sets up the required variables.

It calculates the row offset and initializes the accumulator.

Inside the loop labeled L1, it uses an IFIDNI conditional to check if the eltType is DWORD.

Depending on this condition, it reads the elements from memory correctly.

It accumulates the elements into eax and increments the column index.

The loop continues until all elements are processed.

Finally, the macro cleans up by popping the registers from the stack.

This simplified version of the macro calculates the sum of a row in an array of different element types, taking into account the size of each element type.

```
1815 ; Define the mCalc_row_sum macro
1816 mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
1817     LOCAL L1
1818
1819     ; Save registers
1820     push ebx
1821     push ecx
1822     push esi
1823
1824     ; Set up required registers
1825     mov eax, index
1826     mov ebx, arrayOffset
1827     mov ecx, rowSize
1828
1829     ; Calculate the row offset
1830     mul ecx          ; row index * row size
1831     add ebx, eax      ; row offset
1832
1833     ; Prepare the loop counter
1834     shr ecx, (TYPE eltType / 2) ; Calculate the number of elements in a row
1835
1836     ; Initialize accumulator and column index
1837     mov eax, 0          ; Accumulator
1838     mov esi, 0          ; Column index
1839
```

```
1840 L1:  
1841 ; Check if eltType is DWORD  
1842 IFIDNI <eltType>, <DWORD>  
1843     mov edx, eltType PTR[ebx + esi*(TYPE eltType)]  
1844 ELSE  
1845     ; Assuming eltType is BYTE or WORD  
1846     movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]  
1847 ENDIF  
1848  
1849 add eax, edx          ; Add to accumulator  
1850 inc esi               ; Move to the next column  
1851  
1852 loop L1  
1853  
1854 ; Restore registers  
1855 pop esi  
1856 pop ecx  
1857 pop ebx  
1858  
1859 ENDM  
1860
```

```
1861 .data
1862 ; Define data arrays
1863 tableB BYTE 10h, 20h, 30h, 40h, 50h
1864 RowSizeB = ($ - tableB)
1865 tableW WORD 10h, 20h, 30h, 40h, 50h
1866 RowSizeW = ($ - tableW)
1867 tableD DWORD 10h, 20h, 30h, 40h, 50h
1868 RowSizeD = ($ - tableD)
1869 index DWORD ?

1870
1871 .code
1872 ; Call the mCalc_row_sum macro for different arrays and data types
1873 mCalc_row_sum index, OFFSET tableB, RowSizeB, BYTE
1874 mCalc_row_sum index, OFFSET tableW, RowSizeW, WORD
1875 mCalc_row_sum index, OFFSET tableD, RowSizeD, DWORD
```

Certainly, here's an explanation of the provided code:

The code introduces a macro called `mCalc_row_sum`, which is designed to calculate the sum of a row in a two-dimensional array. This macro takes four parameters: `index`, `arrayOffset`, `rowSize`, and `eltType`.

Inside the macro, registers `ebx`, `ecx`, and `esi` are pushed onto the stack to ensure they are preserved and not affected by the macro's operations.

The `index` parameter represents the row index, `arrayOffset` is the offset of the array, `rowSize` indicates the number of bytes in each table row, and `eltType` specifies the array type, which can be

BYTE, WORD, or DWORD.

The row offset is calculated by multiplying the `rowSize` with the index and adding it to `arrayOffset`. This is done to find the starting address of the row within the array.

To determine the number of elements in a row (whether they are bytes, words, or double words), the macro uses the `eltType`.

If it's DWORD, no scaling is required. If it's BYTE or WORD, the `ecx` register is shifted to the right by 1 or 2 bits, respectively, to adjust it to the number of elements in a row.

The accumulator (`eax`) and the column index (`esi`) are initialized to 0, as the macro iterates over the row.

Within a loop labeled L1, the macro loads an element from memory based on the element type (`eltType`). If the element type is DWORD, it uses a simple `mov` instruction.

If the element type is BYTE or WORD, it uses `movzx` to zero-extend the value.

The element value is added to the accumulator (`eax`), and the column index is incremented (`esi`) to move to the next element in the row.

The loop continues until all elements in the row have been processed.

After the loop, the registers are popped to restore their original values.

The `.data` section defines three different arrays (`tableB`, `tableW`, and `tableD`) with different data types (BYTE, WORD, and DWORD) and calculates the size of a row for each array.

The .code section demonstrates how to use the mCalc\_row\_sum macro with these arrays, specifying the appropriate data type for each call.

Overall, this macro allows you to easily calculate the sum of a row in a 2D array with different data types, making your code more versatile and readable.

## *Special Operators for Macros*

In assembly language macros, there are four special operators that enhance their flexibility and usability:

### Substitution Operator (&):

The substitution operator (&) is a valuable tool in macros. It helps resolve ambiguous references to parameter names within a macro.

For example, consider the mShowRegister macro, which displays the name and hexadecimal contents of a 32-bit register.

When calling this macro with a register name, like ECX, the macro produces output that includes the register name and its value.

1880 .code

1881 mShowRegister ECX

**Creating a String Variable:** If you want to create a string variable inside a macro that includes the register name, using just regName within the string won't work as expected. The preprocessor might treat it as a regular string and not replace it with the argument passed to the macro.

**Using the Substitution Operator (&):** To force the preprocessor to insert the macro argument (e.g., ECX) into the string literal, you can use the substitution operator &. This operator ensures that the macro argument is correctly incorporated into the string. Here's an example of how you can define the tempStr variable with the & operator:

```
1885 mShowRegister MACRO regName  
1886 .data  
1887 tempStr BYTE " &regName=",0
```

In summary, the substitution operator & is a powerful tool for resolving parameter references within macros, making them more versatile and efficient in handling various inputs.

## Expansion Operator (%) in Macros

In assembly language, the expansion operator (%) plays a vital role in macros. It can be used in several ways to evaluate expressions and expand text macros:

**Evaluating Expressions:** When used with TEXTEQU, the % operator evaluates a constant expression and converts the result to an integer.

For example, if you have a variable count = 10, you can use % to calculate (5 + count) and get the integer result, which is then represented as text:

```
1895 count = 10
1896 sumVal TEXTEQU %(5 + count) ; This results in "15"
```

**Flexibility for Passing Arguments:** The % operator offers flexibility in passing arguments to macros. If a macro expects a constant integer argument, you can use the % operator to pass an integer expression. For example:

```
mGotoxyConst %(5 * 10), %(3 + 4)
```

In this case, the expressions within %(...) are evaluated to their integer values, which are then passed to the macro.

**Expanding Macros on a Line:** When the expansion operator (%) is the first character on a source code line, it instructs the preprocessor to expand all text macros and macro functions found on that line.

This can be useful for creating dynamic text during assembly. For example, to display the size of an array, you can use TEXTEQU to create a text macro, and then expand it on the next line:

```
1905 TempStr TEXTEQU %(SIZEOF array)
1906 %
1907 ECHO The array contains TempStr bytes
```

This approach allows for dynamic text generation during assembly.

**Displaying Line Numbers:** In some cases, macros can display the line number from which they were called to help with debugging.

For instance, the LINENUM text macro references @LINE, a predefined assembler operator that returns the current source code line number.

When an error condition is detected in the macro, the **line number can be displayed in an error message**, making it easier to identify and fix issues in the source code.

In summary, the expansion operator (%) is a versatile tool in macros, enabling the evaluation of expressions, dynamic text generation, and enhanced debugging by displaying line numbers in error messages.

## Literal-Text Operator (< >)

The literal-text operator (< >) is a tool that allows you to group characters and symbols into a single text literal.

Its main purpose is to prevent the preprocessor from treating these characters as separate arguments or operators.

This is particularly useful when you have a string that contains special characters like commas, percent signs, ampersands, or semicolons.

These special characters could otherwise be misinterpreted by the preprocessor.

For example, consider the mWrite macro, which expects a string literal as its argument.

If you pass it the following string without using the literal-text operator:

```
mWrite "Line three", 0dh, 0ah
```

The preprocessor would consider this as three separate arguments. In this case, text after the first comma would be discarded because the macro expects only one argument. To prevent this, you can surround the string with the literal-text operator:

```
mWrite <"Line three", 0dh, 0ah>
```

By doing this, the preprocessor treats all text enclosed within the brackets as a single macro argument.

## Literal-Character Operator (!)

The literal-character operator (!) serves a similar purpose to the literal-text operator.

It's used to instruct the preprocessor to treat a predefined operator as a regular character.

This is useful when you need to include special characters within a text literal without them being misinterpreted by the preprocessor.

For example, consider the definition of the BadYValue symbol:

```
BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>
```

Here, the ! operator is used to prevent the > symbol from being treated as a text delimiter. It ensures that the entire text within the < > brackets is preserved as a single text literal.

### Example: Using %, &, and ! Operators

To illustrate these operators, let's consider an example. Suppose you have a symbol called BadYValue, and you want to create a macro called ShowWarning.

This macro takes a text argument, encloses it in quotes, and then passes it to the mWrite macro. You can achieve this using the substitution operator (&) as follows:

```
1930 ShowWarning MACRO message
1931 mWrite "&message"
1932 ENDM
```

Now, when you invoke ShowWarning and pass it the expression %BadYValue, the % operator evaluates (dereferences) BadYValue, turning it into its string representation. The program then displays the warning message correctly:

```
ShowWarning %BadYValue
```

As a result, the program runs and displays the warning message as intended: "Warning: Y-coordinate is > 24."

In summary, the literal-text operator and the literal-character operator are tools to control how the preprocessor interprets and handles special characters within your assembly code, allowing you to maintain the desired structure and functionality of your macros and text literals.

## Macro Functions

A macro function is similar to a regular macro procedure, but with a key difference: it always returns a constant value, either an integer or a string, using the EXITM directive. Let's look at an example to understand this better:

```
1940 IsDefined MACRO symbol
1941     IFDEF symbol
1942         EXITM <-1>; True
1943     ELSE
1944         EXITM <0>; False
1945     ENDIF
1946 ENDM
```

In this example, the IsDefined macro function checks whether a given symbol has been defined. If the symbol is defined, it returns true (represented by -1); otherwise, it returns false (0).

### **Invoking a Macro Function:**

When you want to use a macro function, you need to enclose its argument list in parentheses.

For instance, let's call the IsDefined macro and pass it the symbol RealMode, which may or may not

have been defined:

```
1950 IF IsDefined(RealMode)
1951     mov ax, @data
1952     mov ds, ax
1953 ENDIF
```

If the assembler has already encountered a definition of RealMode before this point in the assembly process, it will assemble the two instructions as shown.

You can also use the macro function within other macros, like this Startup macro:

```
1970 Startup MACRO
1971     IF IsDefined(RealMode)
1972         mov ax, @data
1973         mov ds, ax
1974     ENDIF
1975 ENDM
```

The IsDefined macro can be valuable when you're designing programs for different memory models. For example, you can use it to determine which include file to use based on whether Real Mode is defined.

## Defining the RealMode Symbol

To use the IsDefined macro effectively, you need to define the RealMode symbol appropriately. There are a couple of ways to do this. One is to add the following line at the beginning of your program:

are a couple of ways to do this. One is to add the following line at the beginning of your program:

```
RealMode = 1
```

Alternatively, you can define symbols using the assembler's command-line options. For example, this command defines the RealMode symbol with a value of 1:

```
ML -c -DRealMode=1 myProg.asm
```

This allows you to control whether the RealMode symbol is defined or not when assembling your program.

#### HelloNew Program:

The HelloNew.asm program provided is an example that demonstrates the usage of the macros described.

It displays a message on the screen and chooses the appropriate include file based on whether RealMode is defined.

The program is adaptable to both 16-bit Real Mode and 32-bit Protected Mode.

In summary, macro functions return constant values, either integers or strings, and are useful for conditional assembly based on the existence of defined symbols, making your assembly code more versatile and flexible.

## Questions

### **What is the purpose of the IFB directive?**

The purpose of the IFB directive is to check if a macro argument is blank (empty). It returns true if the argument is empty and false if it contains any content. It's often used to handle cases where a macro expects specific arguments and needs to respond differently when arguments are missing.

### **What is the purpose of the IFIDN directive?**

The purpose of the IFIDN directive is to perform a case-sensitive match between two symbols (or macro parameter names) and determine if they are equal. It returns true if the symbols are the same and false if they are different. This directive is helpful for ensuring that certain conditions are met within a macro.

### **Which directive stops all further expansion of a macro?**

The directive that stops all further expansion of a macro is the EXITM directive. When EXITM is encountered within a macro, it halts the macro's execution, preventing any more macro expansion or code generation.

### **How is IFIDNI different from IFIDN?**

The key difference between IFIDNI and IFIDN is their case sensitivity. IFIDNI performs a case-insensitive match, meaning it treats symbols or names as equal regardless of letter case. In contrast, IFIDN is case-sensitive and only returns true if the symbols or names match exactly, including their letter case.

### **What is the purpose of the IFDEF directive?**

The purpose of the IFDEF directive is to check whether a particular symbol (usually a macro or a variable) has been defined earlier in the code. It returns true if the symbol is defined and false if it is not. IFDEF is commonly used to conditionally include or exclude code blocks based on the

existence of specific symbols in the assembly program.

## ***REPEAT Blocks***

MASM provides several looping directives for generating repeated blocks of statements: WHILE, REPEAT, FOR, and FORC.

These directives operate at assembly time and use constant values for loop conditions and counters.

### **WHILE Directive:**

The WHILE directive repeats a block of statements as long as a specific constant expression remains true. It has the following syntax:

```
1987 WHILE constExpression  
1988     statements  
1989 ENDM
```

For example, you can use the WHILE directive to generate Fibonacci numbers within a specific range, like so:

```
1992 .data
1993     val1 = 1
1994     val2 = 1
1995     DWORD val1 ; First two values
1996     DWORD val2
1997     val3 = val1 + val2
1998
1999 WHILE val3 LT 0F0000000h
2000     DWORD val3
2001     val1 = val2
2002     val2 = val3
2003     val3 = val1 + val2
2004 ENDM
```

This code generates Fibonacci numbers and stores them as assembly-time constants until the value exceeds 0F0000000h.

### REPEAT Directive:

The REPEAT directive repeats a statement block a fixed number of times at assembly time, based on an unsigned constant integer expression.

```
2008 REPEAT constExpression  
2009     statements  
2010 ENDM
```

It's used when you need to repeat a block of code a predetermined number of times, similar to the DUP directive.

### REPEAT Directive Example: Creating an Array

In this example, we use the REPEAT directive to create an array of WeatherReadings. Each WeatherReadings struct contains a location string and arrays for rainfall and humidity readings. The loop repeats for a total of WEEKS\_PER\_YEAR times:

```
2050 WEEKS_PER_YEAR = 52  
2051  
2052 WeatherReadings STRUCT  
2053     location BYTE 50 DUP(0)  
2054     REPEAT WEEKS_PER_YEAR  
2055         LOCAL rainfall, humidity  
2056         rainfall DWORD ?  
2057         humidity DWORD ?  
2058     ENDM  
2059 WeatherReadings ENDS
```

This code defines a structured array for recording weather readings over the course of a year.

## FOR Directive:

The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols. Each symbol in the list represents one iteration of the loop.

```
2015 FOR parameter, <arg1, arg2, arg3, ...>
2016   statements
2017 ENDM
```

It's useful when you want to perform a set of operations for each item in a list of symbols.

### **FOR Directive Example: Student Enrollment**

In this example, the FOR directive is used to create multiple SEMESTER objects for student enrollment in different semesters. The loop iterates over a list of semester names and generates corresponding SEMESTER objects:

```
2065 .data
2066 FOR semName, <Fall2013, Spring2014, Summer2014, Fall2014>
2067   semName SEMESTER <>
2068 ENDM
```

This code generates SEMESTER objects with different names for each semester.

## **FORC Directive:**

The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string represents one iteration of the loop.

```
2020 FORC parameter, <string>
2021     statements
2022 ENDM
```

It's handy when you need to process a block of code for each character in a string.

## **Student Enrollment:**

You can use the FOR directive to create multiple SEMESTER objects, each with a different name from a list of symbols. This can be useful for managing student enrollments over multiple semesters.

## **Character Lookup Table:**

The FORC directive can be used to generate a character lookup table. In this example, a table of non-alphabetic characters is created by iterating through a string of special characters. These looping directives offer flexibility and structure for generating repetitive code in assembly language programs, making it easier to manage and control complex operations.

## FORC Directive Example: Character Lookup Table

In this example, the FORC directive is used to create a character lookup table for non-alphabetic characters. Each character in the string is processed to generate a corresponding entry in the lookup table:

```
2070 Delimiters LABEL BYTE
2071 FORC code, <@#$%^&*!<!>>
2072     BYTE "&code"
2073 ENDM
```

This code generates a lookup table containing the ASCII values and corresponding characters for various special symbols.

---

In this example, we create a linked list data structure using the ListNode structure, which contains a data area (NodeData) and a pointer to the next node (NextPtr).

The program defines and populates multiple instances of ListNode objects within a loop to create a linked list.

Here's the revised and expanded code with added explanations:

```
2077 INCLUDE Irvine32.inc
2078
2079 ; Define the ListNode structure
2080 ListNode STRUCT
2081     NodeData DWORD ?
2082     NextPtr DWORD ?
2083 ListNode ENDS
2084
2085 TotalNodeCount = 15
2086 NULL = 0
2087 Counter = 0
2088
2089 .data
2090 LinkedList LABEL PTR ListNode
2091
2092 ; Use the REPEAT directive to create a linked list
2093 REPEAT TotalNodeCount
2094     Counter = Counter + 1
2095     ; Create a new ListNode with data and link to the next node
2096     ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
2097 ENDM
2098
2099 ; Create a tail node to mark the end of the list
2100 ListNode <0, 0>
2101
2102 .code
2103 main PROC
2104     mov esi, OFFSET LinkedList ; Initialize the pointer to the start of the list
```

```
2105  
2106 NextNode:  
2107     ; Check for the tail node (end of the list).  
2108     mov eax, (ListNode PTR [esi]).NextPtr  
2109     cmp eax, NULL  
2110     je quit ; If NextPtr is NULL, exit the loop  
2111  
2112     ; Display the node data.  
2113     mov eax, (ListNode PTR [esi]).NodeData  
2114     call WriteDec ; Display the integer value  
2115     call Crlf      ; Move to the next line for the next value  
2116  
2117     ; Get the pointer to the next node.  
2118     mov esi, (ListNode PTR [esi]).NextPtr  
2119     jmp NextNode  
2120  
2121 quit:  
2122     exit  
2123  
2124 main ENDP  
2125  
2126 END main
```

Program to illustrate what we have learnt above:



```
2132 INCLUDE Irvine32.inc
2133
2134 ; Define a macro that generates bytes based on a list of values
2135 mGenerateBytes MACRO values
2136     LOCAL L1
2137     FOR val, <values>
2138         BYTE val
2139     ENDM
2140 ENDM
2141
2142 .data
2143 byteArray BYTE 0, 0, 0, 0
2144
2145 .code
2146 main PROC
2147     ; Question 7 - Macro Expansion
2148     ; a
2149     mGenerateBytes <100, 20, 30>
2150     mov eax, OFFSET byteArray
2151     call DisplayByteArray
2152
2153     ; b
2154     mGenerateBytes <AL, 20>
2155     mov eax, OFFSET byteArray
2156     call DisplayByteArray
2157
```

```
2158 ; C
2159 byteVal = 42
2160 countVal = 5
2161 mGenerateBytes <byteVal, countVal>
2162 mov eax, OFFSET byteArray
2163 call DisplayByteArray
2164
2165 ; Question 8 - Linked List Scenario
2166 TotalNodeCount = 15
2167 NULL = 0
2168 Counter = 0
2169 LinkedList LABEL PTR ListNode
2170
2171 REPEAT TotalNodeCount
2172     Counter = Counter + 1
2173     ListNode <Counter, ($ + SIZEOF ListNode)>
2174 ENDM
2175
2176 ; Traverse and display the linked list
2177 mov esi, OFFSET LinkedList
2178
```

```
2179 NextNode:  
2180     mov eax, (ListNode PTR [esi]).NextPtr  
2181     cmp eax, NULL  
2182     je quit  
2183  
2184     mov eax, (ListNode PTR [esi]).NodeData  
2185     call WriteDec  
2186     call Crlf  
2187  
2188     mov esi, (ListNode PTR [esi]).NextPtr  
2189     jmp NextNode  
2190  
2191 quit:  
2192     exit  
2193  
2194 main ENDP  
2195  
2196 ; Helper function to display the contents of the byteArray  
2197 DisplayByteArray PROC  
2198     mov ecx, LENGTHOF byteArray  
2199     mov esi, 0  
2200     mov edx, OFFSET byteArray  
2201  
2202 DisplayLoop:  
2203     mov al, [edx + esi]  
2204     call WriteHex  
2205     call Crlf  
2206     inc esi  
2207     loop DisplayLoop  
2208     ret  
2209 DisplayByteArray ENDP
```

2210 It's gonna be **END main** you know... 😂 😊

## ***WinAPI in ASM***

Skip....

## ***WinAPI in ASM Intro***

When a Windows application launches, it can create either a console window or a graphical window. In our project files, we've used the following option with the LINK command to specify a console-based application:

```
/SUBSYSTEM:CONSOLE
```

A console program resembles an MS-DOS window but with additional features, as we'll explore shortly.

It includes a single input buffer for queuing input records, which contain data about input events such as keyboard input, mouse clicks, and user actions like resizing the console window.

Additionally, it features one or more screen buffers, which are two-dimensional arrays containing character and color data that affect the appearance of text in the console window.

## **Win32 API Reference Information**

Designed for:  
Microsoft®  
Windows® XP  
Windows® 2000

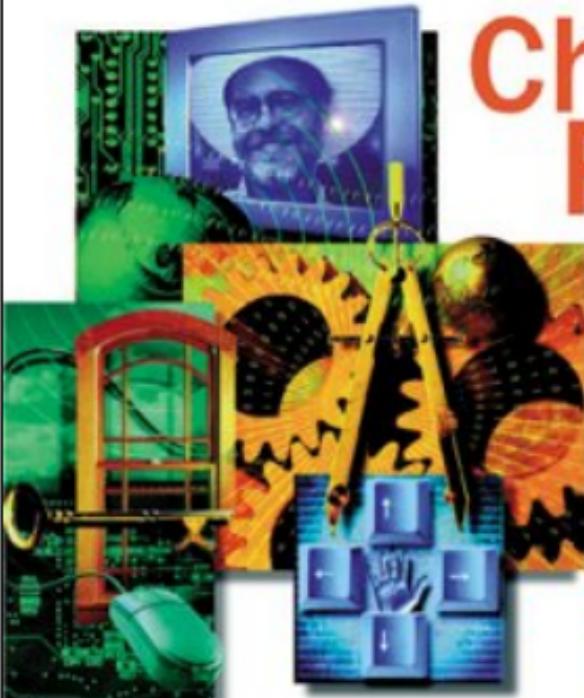
CD-ROM  
Included

Microsoft® Programming Series

Completely  
Revised and  
Updated!

# Programming **Windows**<sup>®</sup>

Fifth Edition



Charles  
Petzold

The definitive  
guide to the  
Win32® API

Microsoft Press

Here is a summary of the key points:

- This section introduces a subset of Win32 API functions with simple examples, but does not cover every detail due to space constraints.
- The Microsoft MSDN website contains full documentation on the Win32 APIs. Make sure to filter for "Platform SDK" when searching.



- The sample programs include lists of function names in kernel32.lib and user32.lib libraries for reference.
- Win32 API functions often use named constants like TIME\_ZONE\_ID\_UNKNOWN.

```
#include <Windows.h>  
// What is the  
// Windows API?
```

- Some constants are defined in SmallWin.inc, others can be found by referring to Windows header files like WinNT.h on the book's website.
- The header files define groups of related constants used by the Win32 functions.



## Windows API

- This overview provides a starting point on using Win32 APIs in assembly, but full details can be found in the Microsoft documentation and header files.
- The example code illustrates simple usage of some key functions.

When calling functions in the Win32 API, two character sets are commonly used: the 8-bit ASCII/ANSI character set and the 16-bit Unicode set, which is available in recent Windows versions.

What is Character Encoding?

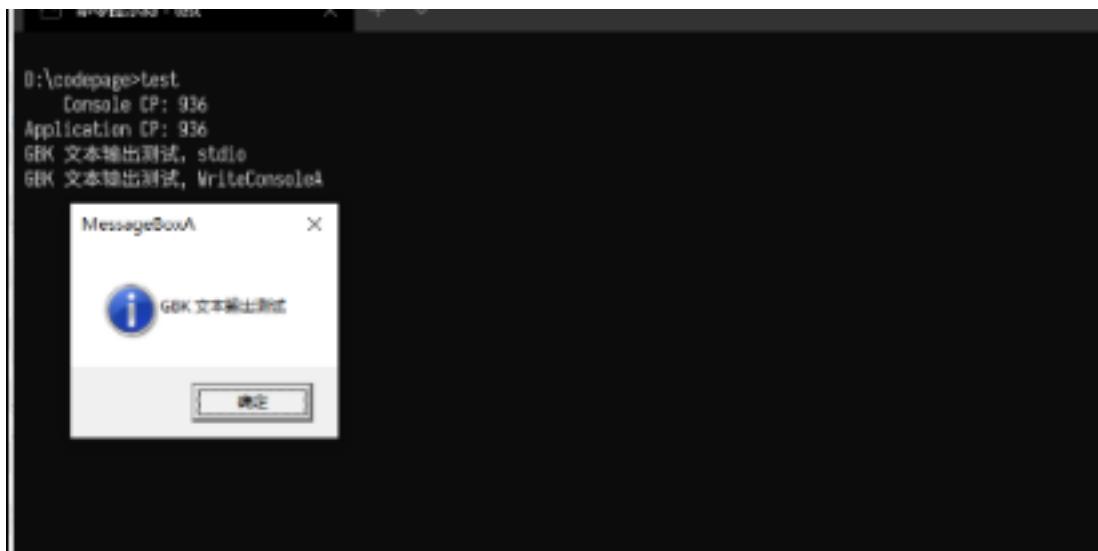
## Character Sets

UTF-8 | UTF-16 | ASCII

Win32 functions related to text come in two versions: one ending with 'A' (for 8-bit ANSI characters) and the other ending with 'W' (for wide character sets, including Unicode). For example, there are two versions of the WriteConsole function:

- **WriteConsoleA**
- **WriteConsoleW**

It's important to note that function names ending with 'W' are not supported in Windows 95 or 98.



In modern Windows versions, Unicode is the native character set. If you call a function like `WriteConsoleA`, the operating system performs character conversion from ANSI to Unicode and then calls `WriteConsoleW`.



In Microsoft's MSDN Library documentation, the trailing 'A' or 'W' is typically omitted from the

function names.

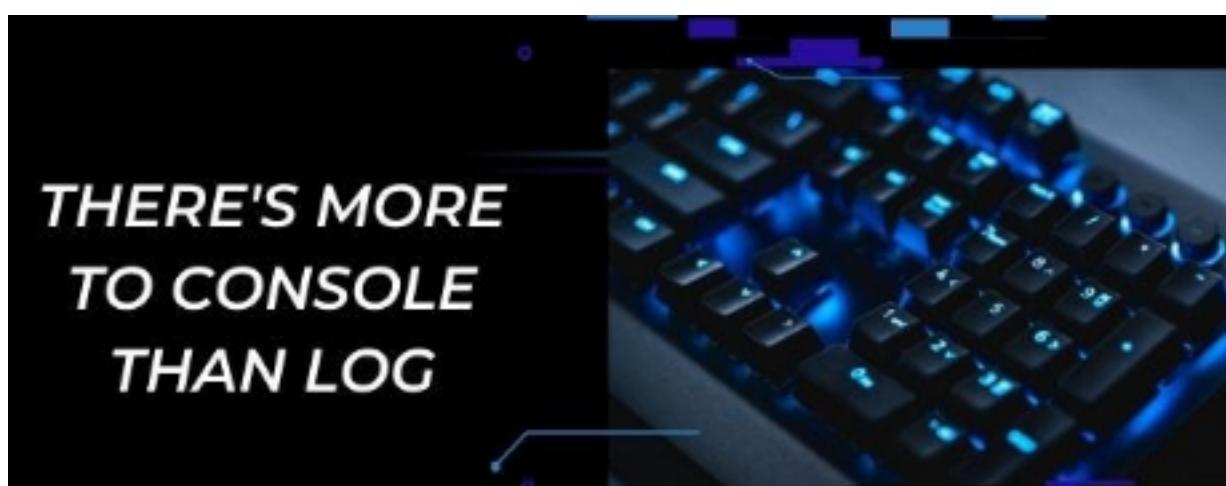
In the program's include files provided with this book, function names like WriteConsoleA are redefined as follows:

```
WriteConsole EQU <WriteConsoleA>
```

This definition allows you to call WriteConsole using the generic name.

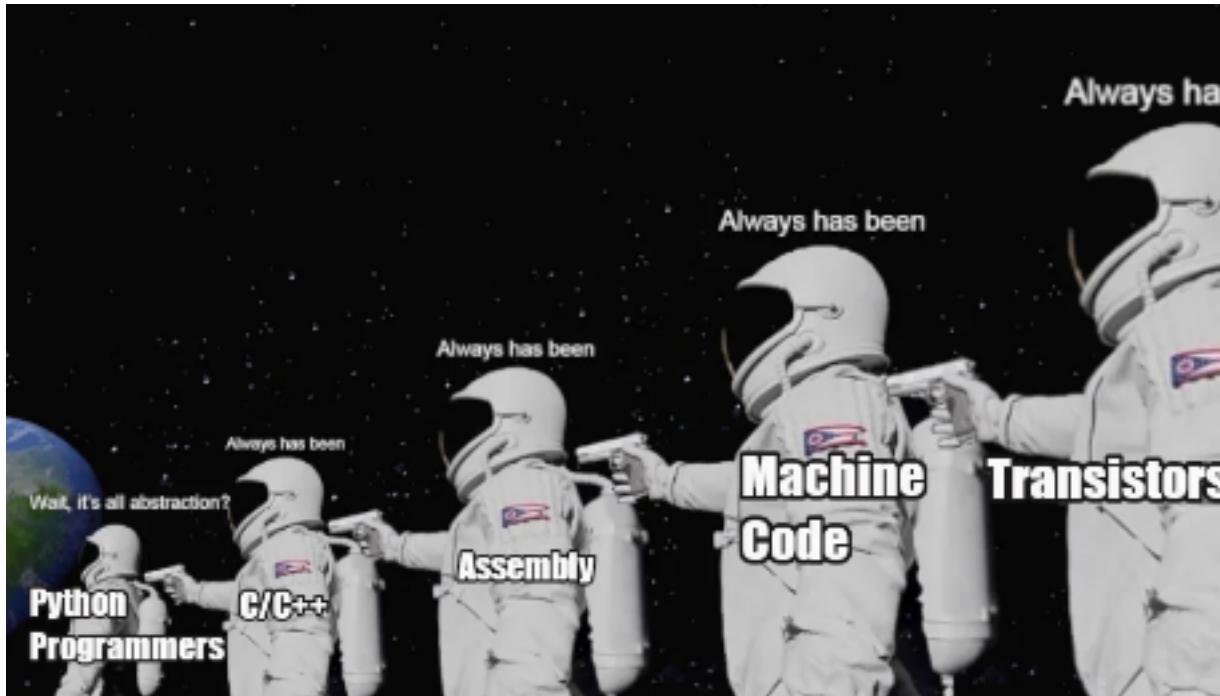
## High-Level and Low-Level Console Access

There are two levels of access to the console, each offering a trade-off between simplicity and complete control:

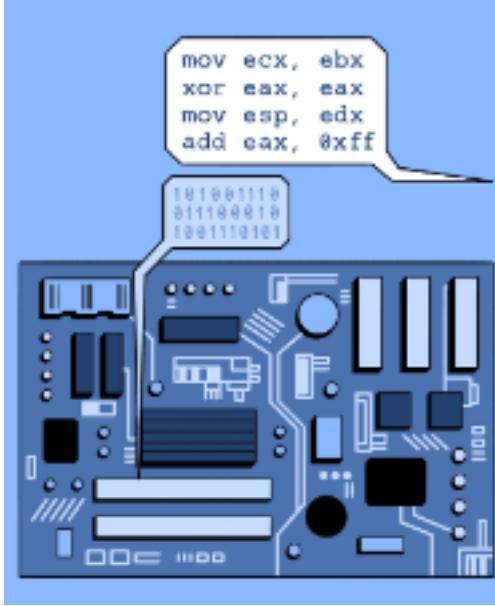


**High-Level Console Functions:** These functions read a stream of characters from the console's input buffer and write character data to the console's screen buffer. Both input and output can be redirected to read from or write to text files.

redirected to read from or write to text files.



**Low-Level Console Functions:** These functions provide detailed information about keyboard and mouse events, as well as user interactions with the console window (e.g., dragging, resizing). They also enable precise control over the window's size, position, and text colors.



# Assembly Language

[ə-'sem-blē 'laŋ-gwij]

Low-level programming language intended to communicate directly with a computer's hardware.

This summary should provide you with a clear understanding of character sets and the distinctions between high-level and low-level console access in Windows API programming. If you have any further questions or need more information, please feel free to ask.

## Windows Data Types

The MASM translations of the MS-Windows data types in Table 11-1 are as follows:

MS-Windows Type	MASM Type	Description
BOOL, BOOLEAN	DWORD	A boolean value (TRUE or FALSE)
BYTE	BYTE	An 8-bit unsigned integer
CHAR	BYTE	An 8-bit Windows ANSI character

In other words, the following MASM types are equivalent to the corresponding MS-Windows types:

- 
- 01 DWORD = BOOL = BOOLEAN
  - 02 BYTE = CHAR

It is important to note that the **HANDLE type in MS-Windows is also a DWORD**. This means that a HANDLE variable can be used to store a handle to any type of object, such as a window, a file, or a memory region.

Here is an example of how to declare and use a HANDLE variable in MASM:

```
07 handleVariable: DWORD  
08  
09 ; Get a handle to the console window.  
10 invoke GetConsoleWindow, handleVariable  
11  
12 ; Use the handle to write a message to the console.  
13 invoke WriteConsole, handleVariable, addr message, length message, bytesWritten, NULL  
14  
15 ; Close the handle to the console window.  
16 invoke CloseHandle, handleVariable
```

The SmallWin.inc include file contains constant definitions, text equates, and function prototypes for Win32 API programming.

It is automatically included in programs by Irvine32.inc. The file contains definitions for several Win32 data types, including the HANDLE type.

Here are some examples of how to use the SmallWin.inc include file:

```
20 ; Get a handle to the standard input handle.  
21 invoke GetStdHandle, STD_INPUT_HANDLE, handleVariable  
22  
23 ; Get a handle to the standard output handle.  
24 invoke GetStdHandle, STD_OUTPUT_HANDLE, handleVariable  
25  
26 ; Get a handle to the standard error handle.  
27 invoke GetStdHandle, STD_ERROR_HANDLE, handleVariable
```

The SmallWin.inc include file can be used to simplify the development of Win32 API programs in MASM.

*Here is a clear and concise explanation of the MS-Windows data types listed in your notes:*

**BOOL, BOOLEAN:** A boolean value, either TRUE or FALSE.

**BYTE:** An 8-bit unsigned integer, meaning that it can store values from 0 to 255.

**CHAR:** An 8-bit Windows ANSI character. ANSI characters are used in older Windows applications and are encoded using a variety of different character sets, depending on the language and region.

**COORD:** A structure that contains two WORD values, X and Y, which represent the coordinates of a point on the screen.

**SYSTEMTIME:** A structure that contains information about a date and time, including the year, month, day, hour, minute, second, and millisecond.

**COLORREF:** A 32-bit value used to represent a color.

**DWORD:** A 32-bit unsigned integer, meaning that it can store values from 0 to 4,294,967,295.

**HANDLE:** A handle is a reference to an object, such as a window, file, or memory region.

**HFILE:** A handle to a file opened by the OpenFile function.

**INT:** A 32-bit signed integer, meaning that it can store values from -2,147,483,648 to 2,147,483,647.

**LONG:** A 32-bit signed integer, the same as INT.

**LPARAM:** A message parameter used by window procedures and callback functions. LPARAM can be used to store any type of data, but it is typically used to store pointers to structures or other data structures.

**LPCSTR:** A pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters.

**LPCVOID:** A pointer to a constant of any type.

**LPSTR:** A pointer to a null-terminated string of 8-bit Windows (ANSI) characters.

**LPCTSTR:** A pointer to a constant character string that is portable for Unicode and double-byte character sets. Unicode is a modern character encoding that can represent characters from all over the world. Double-byte character sets are used to represent characters in languages such as Chinese and Japanese.

**LPTSTR:** A pointer to a character string that is portable for Unicode and double-byte character sets.

**LPVOID**: A pointer to an unspecified type.

**LRESULT**: A 32-bit value returned from a window procedure or callback function.

**SIZE\_T**: The maximum number of bytes to which a pointer can point.

**UINT**: A 32-bit unsigned integer, the same as **DWORD**.

**WNDPROC**: A pointer to a window procedure. A window procedure is a function that is responsible for handling messages sent to a window.

**WORD**: A 16-bit unsigned integer, meaning that it can store values from 0 to 65,535.

**WPARAM**: A 32-bit value passed as a parameter to a window procedure or callback function. **WPARAM** can be used to store any type of data, but it is typically used to store the message ID or other information about the message.

The **SmallWin.inc** include file contains structure definitions, data type definitions, and function prototypes for Win32 API programming. It is automatically included in MASM programs by the **Irvine32.inc** include file.

### **Structures Explained:**

The **COORD structure** is used to store the coordinates of a point on the screen. It contains two **WORD** members, **X** and **Y**, which represent the horizontal and vertical coordinates of the point, respectively.

The **SYSTEMTIME structure** is used to store information about a date and time. It contains the following members:

- **wYear**: The year.
- **wMonth**: The month.
- **wDayOfWeek**: The day of the week.
- **wDay**: The day of the month.
- **wHour**: The hour.
- **wMinute**: The minute.
- **wSecond**: The second.
- **wMilliseconds**: The millisecond.

## Console handles

Console handles are 32-bit unsigned integers that uniquely identify console devices, such as the keyboard, display, and printer. They are used by Win32 console functions to perform input and output operations.

The three standard console handles are:

**STD\_INPUT\_HANDLE**: The standard input handle is used to read keyboard input.

**STD\_OUTPUT\_HANDLE**: The standard output handle is used to write to the console display.

**STD\_ERROR\_HANDLE**: The standard error handle is used to write error messages to the console display. To get a handle to a console device, you can use the **GetStdHandle function**. This function takes a console handle type as a parameter and returns a handle to the corresponding console device.

Once you have a handle to a console device, you can use it to perform input and output operations. For example, to read a character from the keyboard, you can use the **ReadConsole** function. This function takes a console input handle and a buffer as parameters and reads a specified number of

characters from the console input buffer into the buffer.

To write a character to the console display, you can use the WriteConsole function. This function takes a console output handle, a buffer, and a number of characters to write as parameters and writes the specified number of characters from the buffer to the console display.

You can also use console handles to control the appearance and behavior of the console window. For example, to set the title of the console window, you can use the SetConsoleTitle function. This function takes a console window handle and a title string as parameters and sets the title of the console window to the specified string.

Console handles are an essential part of Win32 console programming. By understanding how to use console handles, you can develop powerful and efficient console-based applications.

The handles are:

#### **AllocConsole**

This function allocates a new console for the calling process. This is useful for applications that need to create their own console, such as console-based games or debugging tools.

#### **CreateConsoleScreenBuffer**

This function creates a new console screen buffer. A console screen buffer is a memory area that stores the text and color attributes for the console display.

#### **ExitProcess**

This function ends a process and all its threads. It is typically used to terminate an application when it is finished running or when an error occurs.

#### **FillConsoleOutputAttribute**

This function sets the text and background color attributes for a specified number of character cells. This can be used to change the appearance of text on the console display.

### **FillConsoleOutputCharacter**

This function writes a character to the screen buffer a specified number of times. This can be used to fill a rectangular area of the console display with a single character.

### **FlushConsoleInputBuffer**

This function flushes the console input buffer. The console input buffer is a memory area that stores keyboard input until it is read by an application. Flushing the console input buffer removes all unread input from the buffer.

### **FreeConsole**

This function detaches the calling process from its console. This is useful for applications that need to run without a console, such as services or background tasks.

### **GenerateConsoleCtrlEvent**

This function sends a specified signal to a console process group that shares the console associated with the calling process. This can be used to notify other applications that the calling process is terminating or that an event has occurred.

### **GetConsoleCP**

This function retrieves the input code page used by the console associated with the calling process. The input code page is a table that maps character codes to characters.

### **GetConsoleCursorInfo**

This function retrieves information about the size and visibility of the cursor for the specified console screen buffer.

### **GetConsoleMode**

This function retrieves the current input mode of a console input buffer or the current output mode of a console screen buffer. The input and output modes control the behavior of the console input and output, respectively.

### **GetConsoleOutputCP**

This function retrieves the output code page used by the console associated with the calling process. The output code page is a table that maps characters to character codes.

### **GetConsoleScreenBufferInfo**

This function retrieves information about the specified console screen buffer.

### **GetConsoleTitle**

This function retrieves the title bar string for the current console window.

### **GetConsoleWindow**

This function retrieves the window handle used by the console associated with the calling process.

### **GetLargestConsoleWindowSize**

This function retrieves the size of the largest possible console window.

### **GetNumberOfConsoleInputEvents**

This function retrieves the number of unread input records in the console's input buffer.

### **GetNumberOfConsoleMouseButtons**

This function retrieves the number of buttons on the mouse used by the current console.

### **GetStdHandle**

This function retrieves a handle for the standard input, standard output, or standard error device.

These handles are typically used by console applications to read keyboard input, write to the console display, and write error messages, respectively.

### **HandlerRoutine**

This is an application-defined function that is used with the SetConsoleCtrlHandler function. The SetConsoleCtrlHandler function allows an application to specify a function to be called when the console receives certain signals, such as a close signal or a termination signal.

### **PeekConsoleInput**

This function reads data from the specified console input buffer without removing it from the buffer. This can be used to check for keyboard input without actually reading it.

### **ReadConsole**

This function reads character input from the console input buffer and removes it from the buffer. This is the most common way to read keyboard input in a console application.

### **ReadConsoleInput**

This function reads data from a console input buffer and removes it from the buffer. This function is similar to the ReadConsole function, but it can also read mouse input and other types of input.

### **ReadConsoleOutput**

This function reads character and color attribute data from a rectangular block of character cells in a console screen buffer. This can be used to read the text and appearance of a rectangular area of the console display.

### **ReadConsoleOutputAttribute**

This function copies a specified number of foreground and background color attributes from consecutive cells of a console screen buffer. This can be used to read the color attributes of a rectangular area of the console display.

## **ReadConsoleOutputCharacter**

This function copies a number of characters from consecutive cells of a console screen buffer. This can be used to read the text of a rectangular area of the console display.

## **ScrollConsoleScreenBuffer**

This function moves a block of data in a screen buffer. This can be used to scroll the console display, or to move text or other data within the screen buffer.

## **SetConsoleActiveScreenBuffer**

This function sets the specified screen buffer to be the currently displayed console screen buffer. This can be used to switch between different screen buffers, or to display a different screen buffer in a different console window.

## **SetConsoleCP**

This function sets the input code page used by the console associated with the calling process. The input code page is a table that maps character codes to characters. This function can be used to change the language of the console input, or to support different character sets.

## **SetConsoleCtrlHandler**

This function adds or removes an application-defined HandlerRoutine from the list of handler functions for the calling process. A HandlerRoutine is a function that is called when the console receives certain signals, such as a close signal or a termination signal. This function can be used to implement custom behavior when the console receives these signals.

## **SetConsoleCursorInfo**

This function sets the size and visibility of the cursor for the specified console screen buffer. This function can be used to change the appearance of the cursor, or to hide the cursor altogether.

## **SetConsoleCursorPosition**

This function sets the cursor position in the specified console screen buffer. This function can be used to move the cursor to a specific location on the console display.

## **SetConsoleMode**

This function sets the input mode of a console's input buffer or the output mode of a console screen buffer. The input and output modes control the behavior of the console input and output, respectively. This function can be used to change the behavior of the console keyboard, mouse, and other input devices, or to change the appearance of the console display.

## **SetConsoleOutputCP**

This function sets the output code page used by the console associated with the calling process. The output code page is a table that maps characters to character codes. This function can be used to change the language of the console output, or to support different character sets.

## **SetConsoleScreenBufferSize**

This function changes the size of the specified console screen buffer. This function can be used to increase or decrease the size of the console display, or to accommodate different screen sizes.

## **SetConsoleTextAttribute**

This function sets the foreground (text) and background color attributes of characters written to the screen buffer. This function can be used to change the appearance of text on the console display.

## **SetConsoleTitle**

This function sets the title bar string for the current console window. This can be used to change the title of the console window, or to identify the console window in a list of windows.

## **SetConsoleWindowInfo**

This function sets the current size and position of a console screen buffer's window. This function can be used to resize or move the console window, or to fit the console window to a specific screen area.

### **SetStdHandle**

This function sets the handle for the standard input, standard output, or standard error device. These handles are typically used by console applications to read keyboard input, write to the console display, and write error messages, respectively.

### **WriteConsole**

This function writes a character string to a console screen buffer beginning at the current cursor location. This is the most common way to write text to the console display.

### **WriteConsoleInput**

This function writes data directly to the console input buffer. This function can be used to simulate keyboard input, or to send other types of input to the console.

### **WriteConsoleOutput**

This function writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer. This function can be used to write text and color attribute data to a specific area of the console display.

### **WriteConsoleOutputAttribute**

This function copies a number of foreground and background color attributes to consecutive cells of a console screen buffer. This function can be used to change the color attribute of a specific area of the console display.

### **WriteConsoleOutputCharacter**

This function copies a number of characters to consecutive cells of a console screen buffer. This

function can be used to write text to a specific area of the console display.

## *Displaying a Message Box*

In Win32 console applications, you can use the MessageBoxA function to display a message box to the user. The MessageBoxA function takes four parameters:

**hWnd:** The handle to the window that owns the message box. If this parameter is NULL, the message box will be created as a top-level window.

**lpText:** A pointer to the text to display in the message box.

**lpCaption:** A pointer to the caption of the message box.

**uType:** A bit-mapped integer that specifies the type of message box to display. The uType parameter can be used to specify the buttons to display, the icon to display, and the default button.

The following table shows some of the possible values for the uType parameter:

<b>Value</b>	<b>Integer Value</b>	<b>Description</b>
MB_OK	0	Display a message box with an OK button.
MB_OKCANCEL	1	Display a message box with OK and Cancel buttons.
MB_YESNO	4	Display a message box with Yes and No buttons.
MB_YESNOCANCEL	3	Display a message box with Yes, No, and Cancel buttons.
MB_RETRYCANCEL	5	Display a message box with Retry and Cancel buttons.
MB_ABORTRETRYIGNORE	2	Display a message box with Abort, Retry, and Ignore buttons.
MB_CANCELTRYCONTINUE	6	Display a message box with Cancel, Try Again, and Continue buttons.
MB_ICONSTOP	16	Display a message box with a stop-sign icon.
MB_ICONQUESTION	32	Display a message box with a question-mark icon.
MB_ICONINFORMATION	64	Display a message box with an information-symbol icon.
MB_ICONEXCLAMATION	48	Display a message box with an exclamation-point icon.

The table describes various message box constants and their corresponding descriptions.

These constants are often used in programming to customize the appearance and behavior of message boxes, which are dialog boxes used to display information or request user input in applications.

Clearer Table:

<b>Value</b>	<b>Description</b>
MB_OK	Displays message box with OK button
MB_OKCANCEL	Displays message box with OK and Cancel buttons
MB_YESNO	Displays message box with Yes and No buttons
MB_YESNOCANCEL	Displays message box with Yes, No, and Cancel buttons
MB_RETRYCANCEL	Displays message box with Retry and Cancel buttons
MB_ABORTTRYIGNORE	Displays message box with Abort, Retry, and Ignore buttons
MB_CANCELTRYCONTINUE	Displays message box with Cancel, Try Again, and Continue buttons
MB_ICONSTOP	Displays message box with stop-sign icon
MB_ICONQUESTION	Displays message box with question-mark icon
MB_ICONINFORMATION	Displays message box with information-symbol icon
MB_ICONEXCLAMATION	Displays message box with exclamation-point icon

Yes, the values listed in the table are often used as integer constants to specify the "uType" parameter when creating message boxes in programming.

The "uType" parameter is an integer value that determines the type and behavior of the message box.

You can use the uTypes in your programming language like:

```
30 import ctypes
31
32 # Display a message box with an OK button
33 ctypes.windll.user32.MessageBoxW(0, 'This is an example', 'MessageBox', 0x00000000)
34
35 # Display a message box with Yes and No buttons
36 ctypes.windll.user32.MessageBoxW(0, 'Question?', 'MessageBox', 0x00000004)
37
38 # Display a message box with an exclamation-point icon
39 ctypes.windll.user32.MessageBoxW(0, 'Warning!', 'MessageBox', 0x00000030)
```

Or

```
45 #include <windows.h>
46
47 int main() {
48     ;Display a message box with an OK button
49     MessageBoxW(NULL, L"This is an example", L"MessageBox", MB_OK);
50
51     ;Display a message box with Yes and No buttons
52     int result = MessageBoxW(NULL, L"Question?", L"MessageBox", MB_YESNO);
53
54     ;Check the user's response
55     if (result == IDYES) {
56         ;User clicked Yes
57     } else if (result == IDNO) {
58         ;User clicked No
59     }
60
61     ;Display a message box with an exclamation-point icon
62     MessageBoxW(NULL, L"Warning!", L"MessageBox", MB_ICONEXCLAMATION);
63
64     return 0;
65 }
```

In this C program, we use different message box types, including MB\_OK, MB\_YESNO, and MB\_ICONEXCLAMATION.

You can specify the desired message box type by passing the corresponding integer value as the third parameter to the `MessageBoxW` function.

The program also checks the user's response to the "Yes" and "No" buttons by examining the result returned by the function.

Or

```
070 #include <windows.h>
071
072 int main() {
073     ;Display a message box with an OK button
074     MessageBoxW(NULL, L"This is an example", L"MessageBox", 0);
075
076     ;Display a message box with Yes and No buttons
077     int result = MessageBoxW(NULL, L"Question?", L"MessageBox", 4);
078
079     ;Check the user's response
080     if (result == 6) {
081         ;User clicked Yes
082     } else if (result == 7) {
083         ;User clicked No
084     }
085
086     ;Display a message box with an exclamation-point icon
087     MessageBoxW(NULL, L"Warning!", L"MessageBox", 48);
088
089     return 0;
090 }
```

In this code, we're using the integer values directly to specify the message box types.

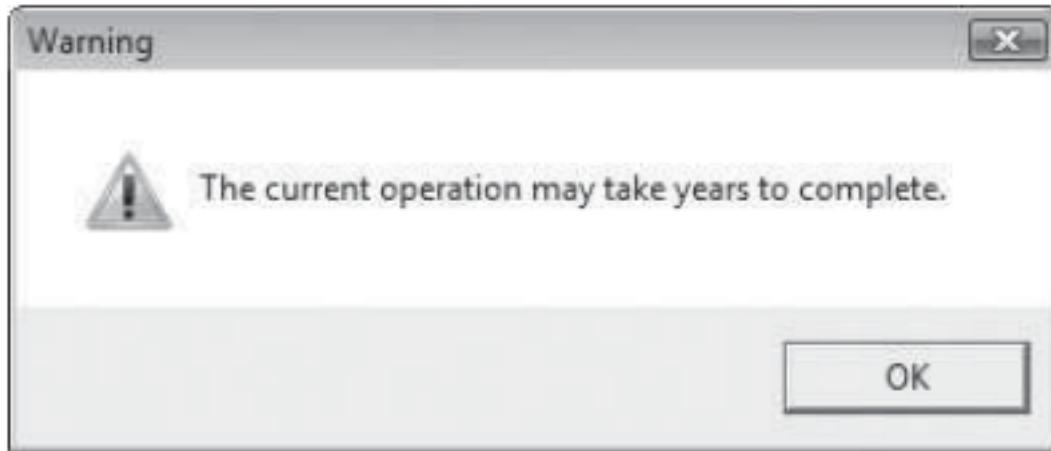
For example, 0 corresponds to MB\_OK, 4 corresponds to MB\_YESNO, and 48 corresponds to

`MB_ICONEXCLAMATION`.

You can use these values to create message boxes with the desired type and behavior in your C program.

## Demonstration Program

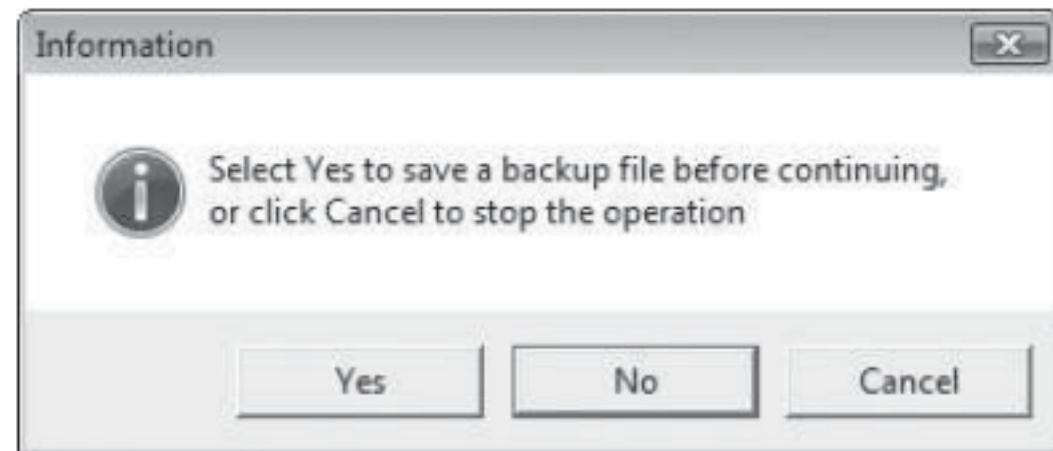
We will demonstrate a short program that demonstrates some capabilities of the `MessageBoxA` function. The first function call displays a warning message:



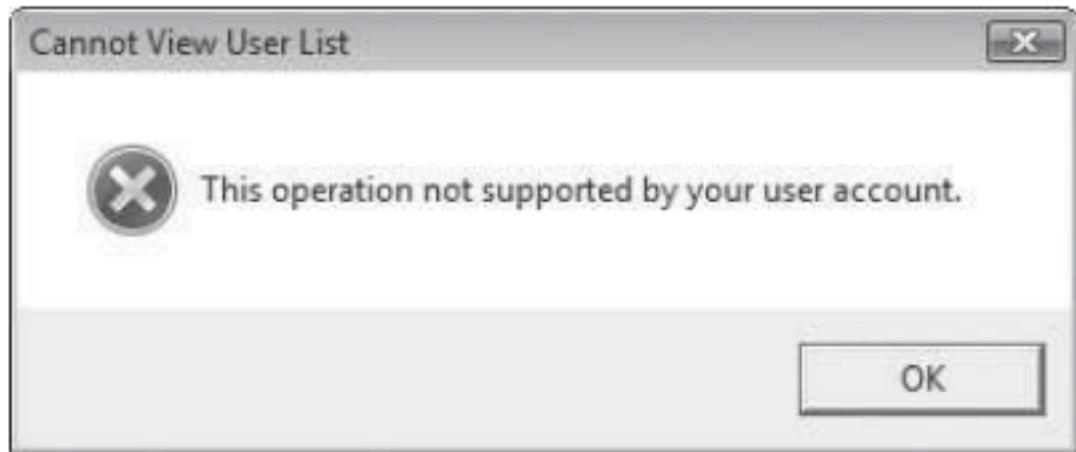
The second function call displays a question icon and Yes/No buttons. If the user selects the Yes button, the program could use the return value to select a course of action:



The third function call displays an information icon with three buttons:



The fourth function call displays a stop icon with an OK button:



```
; Demonstrate MessageBoxA (MessageBox.asm)
INCLUDE Irvine32.inc

.data
captionW BYTE "Warning",0
warningMsg BYTE "The current operation may take years to complete.",0

captionQ BYTE "Question",0
questionMsg BYTE "A matching user account was not found.",0dh,0ah,"Do you wish to continue?",0

captionC BYTE "Information",0
infoMsg BYTE "Select Yes to save a backup file before continuing.",0dh,0ah,"or click Cancel to stop
the operation",0

captionH BYTE "Cannot View User List",0
haltMsg BYTE "This operation not supported by your user account.",0

.code
main PROC
```

```
; Display Exclamation icon with OK button
Invoke MessageBox, NULL, ADDR warningMsg, ADDR captionW, MB_OK + MB_ICONEXCLAMATION

; Display Question icon with Yes/No buttons
Invoke MessageBox, NULL, ADDR questionMsg, ADDR captionQ, MB_YESNO + MB_ICONQUESTION

; interpret the button clicked by the user
cmp eax, IDYES ; YES button clicked?

; Display Information icon with Yes/No/Cancel buttons
Invoke MessageBox, NULL, ADDR infoMsg, ADDR captionC, MB_YESNOCANCEL + MB_ICONINFORMATION +
MB_DEFBUTTON2

; Display stop icon with OK button
Invoke MessageBox, NULL, ADDR haltMsg, ADDR captionH, MB_OK + MB_ICONSTOP

exit
main ENDP
END main
```

The provided code is a demonstration program in assembly language for the Windows API. It showcases the use of the `MessageBox` function to create message boxes with different icons and button options. Here's a clear and concise explanation of the code:

The program starts with including the `Irvine32` library, which provides various macros and functions for 32-bit assembly programming.

In the `.data` section, several strings are defined for different message boxes, including captions and messages.

These strings are null-terminated with a '0' byte at the end. The .code section contains the main procedure (main PROC) where the demonstration begins.

Four message boxes are created using the INVOKE macro, each with different options: A warning message box with an exclamation icon and an OK button.

A question message box with a question icon and Yes/No buttons. An information message box with an information icon and Yes/No/Cancel buttons, with the default button set to No.

A stop message box with a stop icon and an OK button. After displaying each message box, the program checks the result (in the eax register) to determine which button was clicked by the user.

It compares the result with predefined constants like IDYES to check if the "Yes" button was clicked.

Finally, the program exits after displaying all the message boxes.

In summary, this program demonstrates how to create message boxes with different icons and button options using the MessageBox function from the Windows API.

It also shows how to interpret the user's response by comparing the result with predefined constants for button IDs.

## ***Console Input***

```
140 ; Read From the Console (ReadConsole.asm)
141 INCLUDE Irvine32.inc
142
143 BufSize = 80
144
145 .data
146 buffer BYTE BufSize DUP(?), 0, 0
147 stdInHandle HANDLE ?
148 bytesRead DWORD ?
149
150 .code
151 main PROC
152     ; Get handle to standard input
153     INVOKE GetStdHandle, STD_INPUT_HANDLE
154     mov stdInHandle, eax
155
156     ; Wait for user input
157     INVOKE ReadConsole, stdInHandle, ADDR buffer, BufSize, ADDR bytesRead, 0
158
159     ; Display the buffer
160     mov esi, OFFSET buffer
161     mov ecx, bytesRead
162     mov ebx, TYPE buffer
163     call DumpMem
164
165     exit
166 main ENDP
167
168 END main
```

**Console Input Buffer:** In Win32 console programming, there exists an input buffer that stores input event records. These input events include keystrokes, mouse movements, and mouse-button clicks.

High-level input functions like ReadConsole process this input data and return a stream of characters to the program.

**ReadConsole Function:** The ReadConsole function is a Win32 API function used to read text input from the console and store it in a buffer.

It takes several parameters, including the console input handle, a pointer to a character buffer, the number of characters to read, a pointer to store the count of characters read, and a reserved parameter.

The provided code is an example program that demonstrates how to use the ReadConsole function to read characters entered by the user in a console application.

It starts by defining the size of the buffer (BufSize) and declaring the necessary data variables, including a buffer for storing the input, a handle for standard input (stdInHandle), and a variable for the number of bytes read (bytesRead).

In the main procedure, it retrieves the standard input handle using the GetStdHandle function, which returns a handle to the standard input.

It then calls the ReadConsole function to read input from the user.

The function parameters include the standard input handle, the buffer for storing input, the maximum number of characters to read, a pointer to store the count of characters read, and a value of 0 for

the reserved parameter.

After reading the input, it displays the content of the buffer using the DumpMem function, which is part of the Irvine32 library.

The DumpMem function is used to display the buffer's content in both hexadecimal and ASCII representations.

The program can read and display user input, including any end-of-line characters (0Dh and 0Ah) inserted when the user presses the Enter key.

In summary, this code demonstrates how to read and display user input from the console using the ReadConsole function in a Win32 console application. It provides an example of handling console input in assembly language.

## ***Checking for Errors***

In Windows API programming, it's crucial to check for errors when using various API functions. If an API function returns an error value (typically NULL), you can call the GetLastError API function to obtain more information about the error. GetLastError returns a 32-bit integer error code in the EAX register.

### ***GetLastError Function:***

GetLastError is used to retrieve error information after a Windows API function call. It returns a 32-bit integer error code in the EAX register, which can be used to identify the specific error.

If a function returns NULL or an error code, you can call GetLastError to get more details about

what went wrong.

### ***FormatMessage Function:***

After obtaining the error code from `GetLastError`, you might want to retrieve a human-readable error message to better understand the error.

`FormatMessage` is used for this purpose. It formats a message based on the error code. Its parameters are:

**`dwFlags`:** Formatting options, specifying how to interpret the `lpSource` parameter. Recommended values include `FORMAT_MESSAGE_ALLOCATE_BUFFER` and `FORMAT_MESSAGE_FROM_SYSTEM`.

**`lpSource`:** Location of the message definition. For system error messages, set it to `NULL` (0).

**`dwMsgID`:** The error code obtained from `GetLastError`.

**`dwLanguageID`:** Language identifier. Set to zero for a language-neutral or user's default locale message.

**`lpBuffer (output parameter)`:** Pointer to a buffer that receives the null-terminated message string. If `FORMAT_MESSAGE_ALLOCATE_BUFFER` is used, the buffer is allocated automatically.

**`nSize`:** Buffer size, which can be set to 0 if using the recommended `dwFlags` options. **`va_list`:** Pointer to an array of values that can be inserted in a formatted message. Not used for error messages.

### ***Sample Use of GetLastError and FormatMessage:***

First, you call `GetLastError` to obtain the error code. Then, you invoke `FormatMessage` to retrieve the corresponding error message based on the error code. Finally, you can display or handle the error message as needed.

### **LocalFree Function:**

After using FormatMessage to obtain the error message, it's important to release the storage allocated by FormatMessage.

You can use LocalFree for this purpose. The code provided demonstrates the use of GetLastError and FormatMessage to obtain and display error messages in a Windows API program.

It checks for errors, retrieves the error message, and frees allocated memory after use.

Note: Error codes and messages are essential for debugging and providing meaningful feedback to users when errors occur in Windows API applications.

```
.data
 messageId DWORD ?
 pErrorMsg DWORD ? ; points to error message

.code
 call GetLastError
 mov messageId, eax

INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId,
0, ADDR pErrorMsg, 0, NULL

; After calling FormatMessage, you can use the error message in pErrorMsg
; ... (perform error handling or display the error message as needed)
; Don't forget to free the allocated memory when done
```

```
INVOKE LocalFree, pErrorMsg
```

This code checks for errors by calling GetLastError, retrieves the error message using FormatMessage, and then handles or displays the error message. Finally, it frees the allocated memory using LocalFree.

```
.data
    messageId DWORD ?
    pErrorMsg DWORD ? ; points to error message
.code
    call GetLastError
    mov messageId, eax

    INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + FORMAT_MESSAGE_FROM_SYSTEM, NULL,
messageId, 0, ADDR pErrorMsg, 0, NULL

    ; After calling FormatMessage, you can use the error message in pErrorMsg
    ; ... (perform error handling or display the error message as needed)

    ; Don't forget to free the allocated memory when done
    INVOKE LocalFree, pErrorMsg
WriteWindowsMsg PROC USES eax edx
; Displays a string containing the most recent error
; generated by MS-Windows.
; Receives: nothing
; Returns: nothing
.data
    WriteWindowsMsg_1 BYTE "Error ",0
    WriteWindowsMsg_2 BYTE ":",0
    pErrorMsg DWORD ?
    ; points to the error message
```

```
 messageId DWORD ?  
.code  
; Get the most recent error code  
call GetLastErrorHandler  
mov messageId, eax  
; Display the error number as "Error X: "  
mov edx, OFFSET WriteWindowsMsg_1  
call WriteString  
call WriteDec  
mov edx, OFFSET WriteWindowsMsg_2  
call WriteString  
; Get the corresponding error message string  
INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \  
FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId, 0, ADDR pErrorMsg, 0, 0  
; Display the error message generated by MS-Windows  
mov edx, pErrorMsg  
call WriteString  
; Free the error message string  
INVOKE LocalFree, pErrorMsg  
ret  
WriteWindowsMsg ENDP
```

### ***Explanation:***

The WriteWindowsMsg procedure is designed to display error messages generated by the MS-Windows operating system.

It starts by calling GetLastErrorHandler to obtain the most recent error code and stores it in the messageId variable.

The procedure then displays the error number in the format "Error X: " where X is the error code.

It uses the WriteString procedure to display the "Error " and the WriteDec procedure to display the

error code.

After that, it retrieves the corresponding error message string using FormatMessage. The FORMAT\_MESSAGE\_ALLOCATE\_BUFFER and FORMAT\_MESSAGE\_FROM\_SYSTEM flags are used to allocate memory for the message and obtain it from the system.

The obtained error message is displayed using WriteString.

Finally, it frees the memory allocated for the error message using LocalFree.

This procedure is a convenient way to retrieve and display error messages when working with Windows API functions, making it easier to diagnose issues in your applications.

## ***Single Character Input***

### **Single-Character Input and Irvine32 Keyboard Procedures**

In console mode on MS-Windows, handling single-character input involves dealing with the keyboard device driver, scan codes, virtual-key codes, and the message queue. Here's an explanation of the process and the relevant Irvine32 keyboard procedures:

#### **Keyboard Input Process:**

MS-Windows provides a device driver for the installed keyboard. When a key is pressed, it sends an 8-bit scan code to the computer's keyboard port.

Upon releasing the key, a second scan code is transmitted. MS-Windows translates these scan codes into 16-bit virtual-key codes, which are device-independent values that identify the key's purpose.

A message containing the scan code, virtual-key code, and related information is created by MS-Windows and placed in the message queue.

The message eventually reaches the currently executing program thread, identified by the console input handle.

### **Irvine32 Keyboard Procedures:**

The Irvine32 library provides two related procedures for handling keyboard input: ReadChar and ReadKey.

**ReadChar** waits for an ASCII character to be typed at the keyboard and returns the character in the AL register.

**ReadKey** performs a no-wait keyboard check. If no key is waiting in the console input buffer, it sets the Zero flag.

If a key is found, the Zero flag is clear, and AL contains either zero or an ASCII code. The upper halves of EAX and EDX are overwritten.

### **Using ReadKey and Control Key State:**

In ReadKey, if AL contains zero, the user may have pressed a special key (e.g., function key, cursor arrow).

AH register contains the keyboard scan code, which can be matched to a list of keyboard keys. DX contains the virtual-key code.

EBX contains state information about the states of the keyboard control keys.

### **Control Key State Values:**

- **CAPSLOCK\_ON**: The CAPS LOCK light is on.
- **ENHANCED\_KEY**: The key is enhanced.
- **LEFT\_ALT\_PRESSED**: The left ALT key is pressed. **LEFT\_CTRL\_PRESSED**: The left CTRL key is pressed.
- **NUMLOCK\_ON**: The NUM LOCK light is on.
- **RIGHT\_ALT\_PRESSED**: The right ALT key is pressed.
- **RIGHT\_CTRL\_PRESSED**: The right CTRL key is pressed.
- **SCROLLLOCK\_ON**: The SCROLL LOCK light is on.
- **SHIFT\_PRESSED**: The SHIFT key is pressed.

You can use these control key state values to determine the state of control keys while processing keyboard input.

The **ReadChar** and **.ReadKey** procedures in the Irvine32 library simplify handling keyboard input in your assembly programs, making it easier to respond to user interactions.

### **Testing Keyboard Input with ReadKey and GetKeyState**

This section covers testing keyboard input using **ReadKey** and **GetKeyState**, including a program that reports the state of the CapsLock key and another program that checks the state of the NumLock and Left Shift keys.

## Testing Keyboard Input with ReadKey:

The program tests ReadKey by waiting for a keypress and then reporting the state of the CapsLock key.

A **delay factor** is included when calling ReadKey to allow MS-Windows to process its message loop.

If ReadKey returns a non-zero value (a keypress has occurred), the program tests the value of EBX using the **CAPSLOCK\_ON** constant to check the state of the CapsLock key.

It then displays a message indicating whether CapsLock is ON or OFF.

## GetKeyState for Keyboard State Testing:

The GetKeyState API function allows you to test the state of individual keyboard keys. You pass it a virtual key value, like those identified in Table 11-4.

TABLE 11-4 Testing Keys with GetKeyState.

Key	Virtual Key Symbol	Bit to Test in EAX
NumLock	VK_NUMLOCK	0
Scroll Lock	VK_SCROLL	0
Left Shift	VK_LSHIFT	15
Right Shift	VK_RSHIFT	15
Left Ctrl	VK_LCONTROL	15
Right Ctrl	VK_RCONTROL	15
Left Menu	VK_LMENU	15
Right Menu	VK_RMENU	15

It returns a value in EAX, and you need to test the value to determine the state of the key.

The program demonstrates using GetKeyState to check the state of the NumLock and Left Shift keys:

It calls GetKeyState with VK\_NUMLOCK and checks if the lowest bit (bit 0) of AL is set.

If it is set, it indicates that NumLock is ON.

It then calls GetKeyState with VK\_LSHIFT and checks the high bit (bit 31) of EAX to determine if the Left Shift key is currently pressed.

Depending on the test results, it displays appropriate messages to report the state of the keys.

```
227 ;Testing Keyboard Input with ReadKey (TestReadkey.asm)
228 INCLUDE Irvine32.inc
229 INCLUDE Macros.inc
230
231 .code
232 main PROC
233 L1:
234     mov eax, 10      ; Delay for message processing
235     call Delay
236     call ReadKey    ; Wait for a keypress
237     jz L1
238
239     test ebx, CAPSLOCK_ON
240     jz L2
241     mWrite <"CapsLock is ON", 0dh, 0ah>
242     jmp L3
243 L2:
244     mWrite <"CapsLock is OFF", 0dh, 0ah>
245 L3:
246     exit
247 main ENDP
248 END main
```

Program 2:

```
251 ;GetKeyState
252 INCLUDE Irvine32.inc
253 INCLUDE Macros.inc
254
255 .code
256 main PROC
257     INVOKE GetKeyState, VK_NUMLOCK
258     test al, 1
259     .IF !Zero?
260         mWrite <"The NumLock key is ON", 0dh, 0ah>
261     .ENDIF
262
263     INVOKE GetKeyState, VK_LSHIFT
264     test eax, 80000000h
265     .IF !Zero?
266         mWrite <"The Left Shift key is currently DOWN", 0dh, 0ah>
267     .ENDIF
268
269     exit
270 main ENDP
271 END main
```

# **Console Output**

It appears you're looking for an explanation of the WriteConsole function and some associated data structures. Below, I'll provide explanations for WriteConsole and the COORD and SMALL\_RECT structures.

## **WriteConsole Function:**

WriteConsole is a Win32 function used to write a string to the console window at the current cursor position. It is used for console output. Here's a breakdown of its parameters:

- **hConsoleOutput:** This is the handle to the console output stream.
- **lpBuffer:** It's a pointer to the array of characters you want to write. **nNumberOfCharsToWrite:** This parameter holds the length of the array you want to write.
- **lpNumberOfCharsWritten:** It's a pointer to an integer that will receive the number of characters written when the function returns.
- **lpReserved:** This parameter is not used, so you can set it to zero.

WriteConsole writes the string and advances the cursor just past the last character written. It can handle standard ASCII control characters like tabs, carriage returns, and line feeds, and the string doesn't need to be null-terminated.

## **COORD Structure:**

The COORD structure is used in various Win32 console functions. It represents the coordinates of a character cell in the console screen buffer.

The origin of this coordinate system is at the top left cell of the console screen. The structure has two fields:

has two fields:

- **X**: This field is of type WORD and represents the X-coordinate.
- **Y**: This field is also of type WORD and represents the Y-coordinate.

### ***SMALL\_RECT Structure:***

The SMALL\_RECT structure is another data structure used in Win32 console functions. It specifies a rectangular region by defining the upper left and lower right corners of the rectangle within the console window.

It's useful for specifying character cells in the console window. The structure has the following fields:

- **Left**: This field is of type WORD and represents the left coordinate of the rectangle.
- **Top**: This field, also of type WORD, represents the top coordinate.
- **Right**: This field, again of type WORD, represents the right coordinate.
- **Bottom**: This field, once more of type WORD, represents the bottom coordinate.

These data structures are used in various console-related Win32 functions to manage and manipulate console windows. The WriteConsole function is particularly useful for writing content to the console.

### **Example 1: WriteConsole function**

```
275 ; Win32 Console Example #1(Console1.asm)
276 ; This program calls the following Win32 Console functions:
277 ; GetStdHandle, ExitProcess, WriteConsole
278 INCLUDE Irvine32.inc
279 .data
280 endl EQU <0dh,0ah>      ; End of line sequence
281 message LABEL BYTE
282     BYTE "This program is a simple demonstration of"
283     BYTE "console mode output, using the GetStdHandle"
284     BYTE "and WriteConsole functions.",endl
285 messageSize DWORD ($ - message)
286 consoleHandle HANDLE 0      ; Handle to standard output device
287 bytesWritten DWORD ?        ; Number of bytes written
288 .code
289 main PROC
290     ; Get the console output handle:
291     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
292     mov consoleHandle,eax
293
294     ; Write a string to the console:
295     INVOKE WriteConsole,
296             consoleHandle,    ; Console output handle
297             ADDR message,    ; String pointer
298             messageSize,    ; String length
299             ADDR bytesWritten, ; Returns number of bytes written
300             0                ; Not used
301
302     ; Exit the program:
303     INVOKE ExitProcess, 0
304 main ENDP
305 END main
```

The provided program, `Console1.asm`, is a simple example that demonstrates the use of the `GetStdHandle`, `WriteConsole`, and `ExitProcess` functions in a Win32 console application. Here's a breakdown of the code:

The `.data` section begins by defining an `endl` constant that represents the end-of-line sequence (carriage return and line feed).

A message is defined using the `message` label. It contains a multiline text string that the program will write to the console. The `messageSize` variable is used to store the size of the message string.

The `consoleHandle` variable is declared as a `HANDLE` and initialized to 0. This variable will hold the handle to the standard output device (console).

`bytesWritten` is declared as a `DWORD` and will be used to store the number of bytes written by the `WriteConsole` function.

The `.code` section contains the main procedure.

`INVOKE GetStdHandle, STD_OUTPUT_HANDLE` is used to obtain the handle to the standard output (the console). The obtained handle is stored in the `consoleHandle` variable.

The **INVOKE WriteConsole function** is called to write the message string to the console. It takes the following parameters:

- **consoleHandle**: The handle to the console output.
- **ADDR message**: A pointer to the message string.
- **messageSize**: The length of the message string.
- **ADDR bytesWritten**: A pointer to a variable that will receive the number of bytes written.

- `0`: An unused parameter.
- Finally, `INVOKE ExitProcess, 0` is called to exit the program.

When you run this program, it will write the message to the console, and the console window will display the content of the message string.

The output will look like this:

This program is a simple demonstration of  
console mode output, using the `GetStdHandle`  
and `WriteConsole` functions.

This code demonstrates how to use the **Win32 Console functions** to write a message to the console window. The message is stored in the `message` variable and is written to the console using the **WriteConsole function**. Finally, the program exits using `ExitProcess`

### **WriteConsoleOutputCharacter function**

```
311 INCLUDE Irvine32.inc
312
313 .data
314     message BYTE "Hello, World!",0 ; The string to be written
315     coord COORD <5, 5>           ; Starting coordinates in the console
316
317 .code
318 main PROC
319     ; Get the console output handle:
320     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
321     mov edi, eax ; Store the console output handle in EDI
322
323     ; Write the message to the console at the specified coordinates:
324     INVOKE WriteConsoleOutputCharacter, edi, ADDR message, LENGTHOF message - 1, coord, NULL
325
326     ; Exit the program
327     INVOKE ExitProcess, 0
328
329 main ENDP
330
331 END main
```

The `WriteConsoleOutputCharacter` function in Win32 allows you to copy an array of characters to consecutive cells in the console screen buffer at a specified location. Here's a breakdown of its parameters:

- **hConsoleOutput:** This is the handle to the console output. It specifies the console screen buffer where you want to write the characters.
- **lpCharacter:** A pointer to the buffer containing the characters you want to write.

- **nLength:** The size of the buffer, indicating the number of characters to write from the buffer.
- **dwWriteCoord:** This parameter specifies the coordinates of the first cell where you want to start writing. It is of type COORD, which holds X (column) and Y (row) coordinates.
- **lpNumberOfCharsWritten:** A pointer to a DWORD that will receive the count of characters written by the function.

This function writes the characters to the console screen buffer, and if the text reaches the end of a line, it wraps around to the next line.

It's important to note that this function doesn't change the attribute values in the screen buffer, and it ignores ASCII control codes such as tab, carriage return, and line feed.

If the function is successful, it returns a non-zero value, and the number of characters written is stored in lpNumberOfCharsWritten. If it fails, it returns zero.

You can use WriteConsoleOutputCharacter to write text directly to the console, which can be useful for more advanced console applications.

## *File Handling*

### CreateFile Function

The CreateFile function is used to create a new file or open an existing file. It returns a handle to the open file if successful, otherwise, it returns INVALID\_HANDLE\_VALUE.

```
335 CreateFile PROTO,  
336     lpFilename: PTR BYTE,  
337     dwDesiredAccess: DWORD,  
338     dwShareMode: DWORD,  
339     lpSecurityAttributes: DWORD,  
340     dwCreationDisposition: DWORD,  
341     dwFlagsAndAttributes: DWORD,  
342     hTemplateFile: DWORD
```

### Parameters:

- **lpFilename:** Points to the null-terminated string containing the filename.
- **dwDesiredAccess:** Specifies the type of access (read, write, read/write, device query, etc.).
- **dwShareMode:** Controls how multiple programs can access the file while it's open.
- **lpSecurityAttributes:** Points to a security structure controlling security rights.
- **dwCreationDisposition:** Specifies what to do when the file exists or doesn't exist.
- **dwFlagsAndAttributes:** Contains bit flags specifying file attributes like archive, encrypted, hidden, etc.
- **hTemplateFile:** An optional handle to a template file for attributes and extended attributes.

### **dwDesiredAccess** Parameter Options:

The dwDesiredAccess parameter specifies the type of access to the file. You can choose from the following options or specific flag values:

- **0:** Device query access, to check device attributes or file existence.
- **GENERIC\_READ:** Read access for reading from the file.

- **GENERIC\_WRITE**: Write access for writing to the file.

### ***dwCreationDisposition Parameter Options:***

The dwCreationDisposition parameter specifies actions on existing and non-existing files.

Choose one of the following options:

- **CREATE\_NEW**: Creates a new file, fails if it already exists.
  - **CREATE\_ALWAYS**: Creates a new file, overwrites if it exists.
  - **OPEN\_EXISTING**: Opens an existing file, fails if it doesn't exist.
  - **OPEN\_ALWAYS**: Opens the file if it exists, creates if it doesn't.
  - **TRUNCATE\_EXISTING**: Opens the file and truncates it to size zero, fails if it doesn't exist.
- 

Let's delve deeper into the CreateFile function and provide some code examples.

**lpFilename**: This is the path to the file you want to create or open. It can be a fully qualified filename (including the drive and path) or just the filename. For example, "C:\myfolder\myfile.txt" or "myfile.txt".

**dwDesiredAccess**: Specifies the type of access to the file. You can use a combination of these flags (bitwise OR) to specify the desired access:

- **GENERIC\_READ**: Read access.
- **GENERIC\_WRITE**: Write access.

**0:** Device query access (useful for checking device attributes or file existence). **dwShareMode:** This parameter controls how other processes can access the file while it is open. It can take one or a combination of these flags (bitwise OR):

- **FILE\_SHARE\_READ:** Other processes can read the file.
- **FILE\_SHARE\_WRITE:** Other processes can write to the file.
- **0:** No sharing allowed.

**lpSecurityAttributes:** This parameter allows you to specify a security structure, but you can usually set it to NULL if you don't need to set specific security attributes.

**dwCreationDisposition:** Specifies what to do when the file exists or doesn't exist. You can choose from these options:

- **CREATE\_NEW:** Creates a new file. If the file already exists, the function fails.
- **CREATE\_ALWAYS:** Creates a new file. If it exists, it overwrites it.
- **OPEN\_EXISTING:** Opens an existing file. If it doesn't exist, the function fails.
- **OPEN\_ALWAYS:** Opens the file if it exists or creates it if it doesn't.
- **TRUNCATE\_EXISTING:** Opens the file and truncates it to size zero. Fails if the file doesn't exist.

**dwFlagsAndAttributes:** This parameter allows you to set various file attributes. Common attributes include FILE\_ATTRIBUTE\_NORMAL, FILE\_ATTRIBUTE\_ARCHIVE, FILE\_ATTRIBUTE\_HIDDEN, etc.

**hTemplateFile:** You can typically set this to NULL. It's an optional handle to a template file that can supply file attributes and extended attributes for the file being created.

Here's an example of how you might use the CreateFile function in assembly language to create or open a text file and get a handle to it:

```
344 .data
345     filePath BYTE "myfile.txt", 0
346     handle HANDLE ?
347
348 .code
349     ; Create or open the file for writing
350     INVOKE CreateFile, ADDR filePath, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
351
352     ; Check if the file handle is valid
353     cmp eax, INVALID_HANDLE_VALUE
354     je fileCreationFailed
355
356     ; Store the file handle
357     mov handle, eax
358
359     ; Now you can write to the file using the handle
360
361     ; Close the file when done
362     INVOKE CloseHandle, handle
363
364 fileCreationFailed:
365     ; Handle the case where file creation/opening failed
366     ; This could include error checking and cleanup
```

This code opens or creates the file "myfile.txt" for writing and checks if the operation was successful.

### Combined Code:

Here's a single code example that demonstrates the creation and opening of files using CreateFile, and reading from a file using ReadFile. It uses the file "mydata.txt" for illustration:

```
370 .data
371     filePath BYTE "mydata.txt", 0
372     handle HANDLE ?
373     bytesRead DWORD ?
374     buffer BYTE 128 DUP(?)
375 .code
376     ; Create or open the file for writing
377     INVOKE CreateFile, ADDR filePath, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
378     ; Check if the file handle is valid
379     cmp eax, INVALID_HANDLE_VALUE
380     je fileCreationFailed
381     ; Store the file handle
382     mov handle, eax
383     ; Write some data to the file (assuming data is in the buffer)
384     ; Close the file handle
385     INVOKE CloseHandle, handle
386     ; Reopen the file for reading
387     INVOKE CreateFile, ADDR filePath, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
388     ; Check if the file handle is valid
389     cmp eax, INVALID_HANDLE_VALUE
390     je fileOpenFailed
391     ; Store the file handle
392     mov handle, eax
393     ; Read data from the file
394     INVOKE ReadFile, handle, ADDR buffer, 128, ADDR bytesRead, 0
395     ; Handle the data read from the file
396     ; Close the file handle
397     INVOKE CloseHandle, handle
398 fileOpenFailed:
399 fileCreationFailed:
400     ; Handle any failures during file creation or opening
```

### **CreateFile Function:**

The CreateFile function is used to either create a new file or open an existing file. It returns a handle to the open file if successful or INVALID\_HANDLE\_VALUE if it fails. Several parameters determine how the file is accessed and what happens in various scenarios.

**lpFileName:** This parameter points to a null-terminated string representing the file's name and location.

**dwDesiredAccess:** Specifies the type of access, such as read, write, or both. It uses flags like GENERIC\_READ and GENERIC\_WRITE.

**dwShareMode:** It controls the sharing of the file among multiple programs. You can specify sharing options using constants like FILE\_SHARE\_READ and FILE\_SHARE\_WRITE.

**lpSecurityAttributes:** This parameter can point to a security structure that controls security rights, but for most use cases, it's set to NULL.

**dwCreationDisposition:** Determines what happens when the file already exists or not. It uses options like CREATE\_NEW, CREATE\_ALWAYS, OPEN\_EXISTING, etc.

**dwFlagsAndAttributes:** Contains attributes that define the file, like being hidden or read-only, and can be a combination of various attribute flags.

**hTemplateFile:** Optional and is used for specifying a template file that provides attributes and extended attributes for the new file.

### **File Access and Sharing:**

**dwDesiredAccess** defines the type of access you want. **GENERIC\_READ** allows reading, **GENERIC\_WRITE** permits writing, and you can combine them for both read and write access.

**dwShareMode** controls how the file can be shared among different programs. It includes options like **FILE\_SHARE\_READ** and **FILE\_SHARE\_WRITE**, which determine whether other processes can read or write to the file simultaneously.

### **Creation and Opening Scenarios:**

**dwCreationDisposition** specifies what should happen when opening a file: **CREATE\_NEW**: Creates a new file, failing if it already exists. **CREATE\_ALWAYS**: Creates a new file, overwriting an existing one. **OPEN\_EXISTING**: Opens an existing file. **OPEN\_ALWAYS**: Opens the file if it exists or creates a new one if it doesn't. **TRUNCATE\_EXISTING**: Opens the file and truncates it to size zero. Attributes:

**dwFlagsAndAttributes** allows you to set file attributes like **FILE\_ATTRIBUTE\_ARCHIVE**, **FILE\_ATTRIBUTE\_HIDDEN**, **FILE\_ATTRIBUTE\_NORMAL**, and **FILE\_ATTRIBUTE\_READONLY**. **CloseHandle Function**:

After working with a file, it's crucial to close the handle using the **CloseHandle** function. This releases system resources and ensures data integrity.

### **ReadFile Function:**

**ReadFile** is used to read data from a file. It requires the file handle, a buffer to store the data, the number of bytes to read, and a pointer to a variable that will hold the number of bytes actually read.

**Synchronous reading** is achieved by setting **lpOverlapped** to **NULL**. In practice, you would use these functions in a sequence, like opening a file, reading or writing data, and then closing the file handle to ensure proper file handling.

The code example provided earlier demonstrates a simple file creation, writing, and reading scenario, where you can see these functions in action.

## ***WriteFile***

### **WriteFile Function:**

The WriteFile function is used to write data to a file or an output handle. The handle can represent a file or another output destination like the screen buffer.

The function writes data to the file starting at the position indicated by the file's internal position pointer.

After the write operation is completed, the file's position pointer is adjusted by the number of bytes actually written.

**hFile:** This is the handle to the file or output destination where the data should be written.

**lpBuffer:** It's a pointer to the buffer containing the data you want to write.

**nNumberOfBytesToWrite:** Specifies how many bytes should be written to the file.

**lpNumberOfBytesWritten:** A pointer to an integer that will hold the number of bytes actually written after the operation is completed.

**lpOverlapped:** This should be set to NULL for synchronous operation. It's used for asynchronous operations. The return value is zero if the function fails, and it's a non-zero value if the write operation is successful.

### **SetFilePointer Function:**

The SetFilePointer function is used to move the position pointer of an open file. This function is handy for appending data to a file or for performing random-access record processing. It's often used to navigate within a file.

**hFile:** The file handle represents the file you want to move the pointer within.

**lDistanceToMove:** This is the number of bytes you want to move the pointer. It can be positive or negative, allowing you to move forward or backward within the file.

**lpDistanceToMoveHigh:** This is a pointer to a variable that contains the upper 32 bits of the distance. It's used for handling large file sizes, and if it's set to NULL, only the value in lDistanceToMove is considered.

**dwMoveMethod:** Specifies the starting point for moving the file pointer and can take one of three values: FILE\_BEGIN (absolute file positioning), FILE\_CURRENT (relative to the current file position), and FILE\_END (relative to the end of the file).

For example, to prepare to append data to the end of a file, you can use FILE\_END as the move method:

```
407 INVOKE SetFilePointer,  
408 fileHandle, ; file handle  
409 0, ; distance low  
410 0, ; distance high  
411 FILE_END ; move method
```

These functions are crucial for managing file access and data writing in Windows programming. They are often used in sequence, with SetFilePointer positioning the file pointer to the desired location, and WriteFile writing data to that location. Proper usage of these functions ensures efficient file manipulation in Windows applications.

```
417 ;-----
418 ; CreateOutputFile PROC
419 ;
420 ; Creates a new file and opens it in output mode.
421 ;
422 ; Receives: EDX points to the filename.
423 ;
424 ; Returns: If the file was created successfully, EAX
425 ; contains a valid file handle. Otherwise, EAX
426 ; equals INVALID_HANDLE_VALUE.
427 ;
428 ;-----
429 INVOKE CreateFile,
430     edx, GENERIC_WRITE, DO_NOT_SHARE, NULL,
431     CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
432 ret
433 CreateOutputFile ENDP
```

```
435 ;-----  
436 ; OpenFile PROC  
437 ;  
438 ; Opens a new text file and opens for input.  
439 ;  
440 ; Receives: EDX points to the filename.  
441 ;  
442 ; Returns: If the file was opened successfully, EAX  
443 ; contains a valid file handle. Otherwise, EAX equals  
444 ; INVALID_HANDLE_VALUE.  
445 ;  
446 ;-----  
447 INVOKE CreateFile,  
448     edx, GENERIC_READ, DO_NOT_SHARE, NULL,  
449     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0  
450 ret  
451 OpenFile ENDP
```

```
453 ;-----
454 ; WriteToFile PROC
455 ;
456 ; Writes a buffer to an output file.
457 ;
458 ; Receives: EAX = file handle, EDX = buffer offset,
459 ; ECX = number of bytes to write
460 ;
461 ; Returns: EAX = number of bytes written to the file.
462 ; If the value returned in EAX is less than the
463 ; argument passed in ECX, an error likely occurred.
464 ;
465 ;-----
466 .data
467 WriteToFile_1 DWORD ?
468 ; number of bytes written
469 .code
470 INVOKE WriteFile,
471     eax,          ; file handle
472     edx,          ; buffer pointer
473     ecx,          ; number of bytes to write
474     ADDR WriteToFile_1, ; number of bytes written
475     0             ; overlapped execution flag
476 mov eax, WriteToFile_1 ; return value
477 ret
478 WriteToFile ENDP
```

```
480 ;-----  
481 ; ReadFromFile PROC  
482;  
483 ; Reads an input file into a buffer.  
484;  
485 ; Receives: EAX = file handle, EDX = buffer offset,  
486 ; ECX = number of bytes to read  
487;  
488 ; Returns: If CF = 0, EAX = number of bytes read; if  
489 ; CF = 1, EAX contains the system error code returned  
490 ; by the GetLastError Win32 API function.  
491;  
492 ;-----  
493 .data  
494 ReadFromFile_1 DWORD ?  
495 ; number of bytes read  
496 .code  
497 INVOKE ReadFile,  
498     eax,          ; file handle  
499     edx,          ; buffer pointer  
500     ecx,          ; max bytes to read  
501     ADDR ReadFromFile_1, ; number of bytes read  
502     0              ; overlapped execution flag  
503 mov eax, ReadFromFile_1  
504 ret  
505 ReadFromFile ENDP
```

```
507 ;-----  
508 ; CloseFile PROC  
509 ;  
510 ; Closes a file using its handle as an identifier.  
511 ;  
512 ; Receives: EAX = file handle  
513 ;  
514 ; Returns: EAX = nonzero if the file is successfully closed.  
515 ;  
516 ;-----  
517 INVOKE CloseHandle, eax  
518 ret  
519 CloseFile ENDP
```

That was the first program to test your knowledge, now let's do the second one:

```
524 ; Creating a File (CreateFile.asm)
525 INCLUDE Irvine32.inc
526 BUFFER_SIZE = 501
527
528 .data
529 buffer BYTE BUFFER_SIZE DUP(?)
530 filename BYTE "output.txt",0
531 fileHandle HANDLE ?
532 stringLength DWORD ?
533 bytesWritten DWORD ?
534 str1 BYTE "Cannot create file",0dh,0ah,0
535 str2 BYTE "Bytes written to file [output.txt]:",0
536 str3 BYTE "Enter up to 500 characters and press [Enter]: ",0dh,0ah,0
537
538 .code
539 main PROC
540     ; Create a new text file.
541     mov edx, OFFSET filename      ; Load the address of the filename.
542     call CreateOutputFile        ; Call the CreateOutputFile procedure.
543     mov fileHandle, eax         ; Store the file handle in fileHandle.
544
545     ; Check for errors.
546     cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
547     jne file_ok                 ; If not equal, jump to file_ok.
548
549     ; If there's an error, display the error message and exit.
550     mov edx, OFFSET str1         ; Load the address of the error message.
551     call WriteString            ; Call WriteString to display the error message.
552     jmp quit                    ; Jump to quit to exit.
```



```
554 file_ok:  
555     ; Ask the user to input a string.  
556     mov edx, OFFSET str3          ; Load the address of the input prompt.  
557     call WriteString            ; Call WriteString to display the input prompt.  
558  
559     mov ecx, BUFFER_SIZE        ; Load the maximum buffer size.  
560  
561     ; Input a string.  
562     mov edx, OFFSET buffer      ; Load the address of the buffer.  
563     call ReadString            ; Call ReadString to get user input.  
564     mov stringLength, eax       ; Store the length of the entered string.  
565     ; Write the buffer to the output file.  
566     mov eax, fileHandle         ; Load the file handle.  
567     mov edx, OFFSET buffer      ; Load the address of the buffer.  
568     mov ecx, stringLength       ; Load the length of the string.  
569     call WriteToFile           ; Call WriteToFile to write to the file.  
570     mov bytesWritten, eax       ; Store the number of bytes written.  
571     ; Close the file.  
572     call CloseFile              ; Call CloseFile to close the file.  
573     ; Display the return value.  
574     mov edx, OFFSET str2          ; Load the address of the output message.  
575     call WriteString            ; Call WriteString to display the message.  
576     mov eax, bytesWritten       ; Load the number of bytes written.  
577     call WriteDec               ; Call WriteDec to display the value.  
578     call Crlf                  ; Call Crlf to add a new line.  
579 quit:  
580     exit  
581 main ENDP  
582 END main
```

That's the second program.

Let's try another program:

```
587 ; Reading a File (ReadFile.asm)
588 ; Opens, reads, and displays a text file using
589 ; procedures from Irvine32.lib.
590 INCLUDE Irvine32.inc
591 INCLUDE macros.inc
592 BUFFER_SIZE = 5000
593
594 .data
595 buffer BYTE BUFFER_SIZE DUP(?)
596 filename BYTE 80 DUP(0)
597 fileHandle HANDLE ?
598
599 .code
600 main PROC
601     ; Let the user input a filename.
602     mWrite "Enter an input filename: " ; Display the input prompt.
603     mov edx, OFFSET filename ; Load the address of the filename.
604     mov ecx, SIZEOF filename ; Load the size of the filename.
605     call ReadString ; Call ReadString to get user input.
606
607     ; Open the file for input.
608     mov edx, OFFSET filename ; Load the address of the filename.
609     call OpenInputFile ; Call OpenInputFile to open the file.
610     mov fileHandle, eax ; Store the file handle in fileHandle.
611
612     ; Check for errors when opening the file.
613     cmp eax, INVALID_HANDLE_VALUE ; Compare the result to INVALID_HANDLE_VALUE.
614     jne file_ok ; If not equal, jump to file_ok.
```



```
616 ; If there's an error, display the error message and exit.  
617 mWrite <"Cannot open file", 0dh, 0ah> ; Display the error message.  
618 jmp quit ; Jump to quit to exit.  
619  
620 file_ok:  
621 ; Read the file into a buffer.  
622 mov edx, OFFSET buffer ; Load the address of the buffer.  
623 mov ecx, BUFFER_SIZE ; Load the buffer size.  
624 call ReadFromFile ; Call ReadFromFile to read the file.  
625  
626 jnc check_buffer_size ; If no error, jump to check_buffer_size.  
627  
628 ; If there's an error, display an error message.  
629 mWrite "Error reading file. " ; Display the error message.  
630 call WriteWindowsMsg ; Call WriteWindowsMsg to display the Windows error message.  
631 jmp close_file ; Jump to close_file to close the file.  
632  
633 check_buffer_size:  
634 cmp eax, BUFFER_SIZE ; Compare the result to BUFFER_SIZE.  
635 jb buf_size_ok ; If less, jump to buf_size_ok.  
636  
637 ; If the buffer is too small for the file, display an error message and exit.  
638 mWrite <"Error: Buffer too small for the file", 0dh, 0ah> ; Display the error message.  
639 jmp quit ; Jump to quit to exit.  
640  
641 buf_size_ok:  
642 mov buffer[eax], 0 ; Insert a null terminator.  
643 mWrite "File size: " ; Display a message about the file size.  
644 call WriteDec ; Call WriteDec to display the file size.  
645 call Crlf ; Call Crlf to add a new line.
```

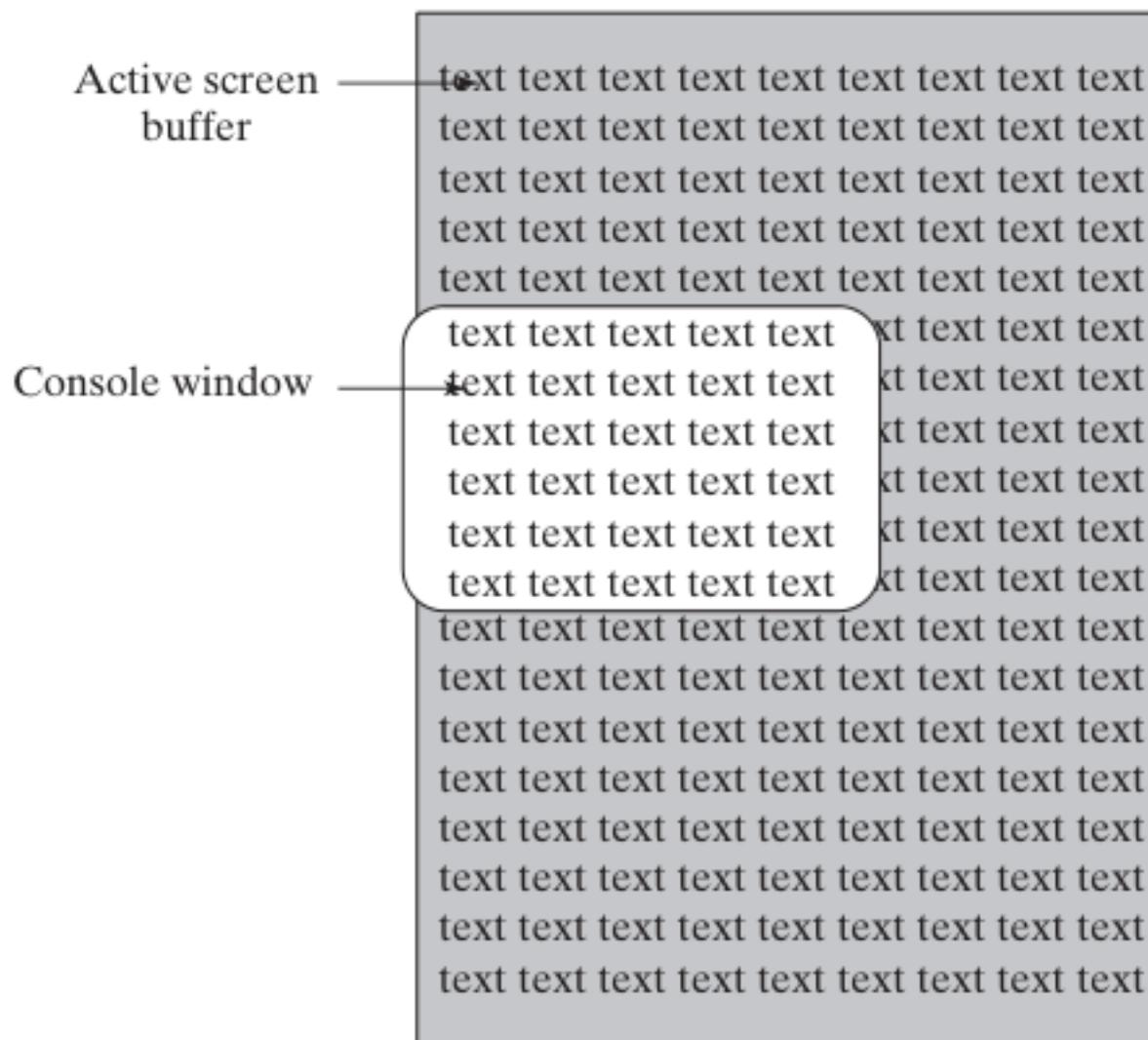
```
646  
647     ; Display the buffer.  
648     mWrite <"Buffer:", 0dh, 0ah, 0dh, 0ah> ; Display the buffer message.  
649     mov edx, OFFSET buffer ; Load the address of the buffer.  
650     call WriteString ; Call WriteString to display the buffer.  
651     call Crlf ; Call Crlf to add a new line.  
652  
653 close_file:  
654     mov eax, fileHandle ; Load the file handle.  
655     call CloseFile ; Call CloseFile to close the file.  
656  
657 quit:  
658     exit ; Exit the program.  
659 main ENDP  
660  
661 END main
```

That's the 3rd program.

## *Console Window Manipulation*

I'll simplify the notes and provide commented code for each of the functions:

## Screen buffer and console window.



The image you sent shows a screen buffer and console window. The screen buffer is a memory area that stores the text and color attributes for the console display. The console window is the window that displays the console buffer.

To manipulate the screen buffer in assembly WinAPI, you can use the following functions:

**WriteConsoleOutput()**: Writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer.

**ReadConsoleOutput()**: Reads character and color attribute data from a specified rectangular block of character cells in a console screen buffer.

**SetConsoleCursorPosition()**: Sets the cursor position in the specified console screen buffer. The following code shows an example of how to use the WriteConsoleOutput function to write text to the screen buffer:

```
700 ; Get a handle to the console screen buffer.  
701 mov eax, STD_OUTPUT_HANDLE  
702 invoke GetStdHandle  
703 mov ebx, eax  
704  
705 ; Set the cursor position.  
706 mov ecx, 0 ; X coordinate  
707 mov edx, 0 ; Y coordinate  
708 invoke SetConsoleCursorPosition  
709 mov esi, ebx  
710  
711 ; Write the text to the screen buffer.  
712 mov edi, 0 ; X coordinate  
713 mov edi, 0 ; Y coordinate  
714 mov al, 'A'  
715 invoke WriteConsoleOutput  
716  
717 ; Exit the program.  
718 mov eax, 0  
719 invoke ExitProcess
```

This code will write the character 'A' to the screen buffer at the top-left corner of the console window.

You can use the ReadConsoleOutput function to read text from the screen buffer. For example, the

following code shows how to read a single character from the screen buffer:

```
722 ; Get a handle to the console screen buffer.  
723 mov eax, STD_OUTPUT_HANDLE  
724 invoke GetStdHandle  
725 mov ebx, eax  
726  
727 ; Set the cursor position.  
728 mov ecx, 0 ; X coordinate  
729 mov edx, 0 ; Y coordinate  
730 invoke SetConsoleCursorPosition  
731 mov esi, ebx  
732  
733 ; Read a single character from the screen buffer.  
734 mov edi, 0 ; X coordinate  
735 mov edi, 0 ; Y coordinate  
736 mov al, 1 ; Number of characters to read  
737 invoke ReadConsoleOutput  
738  
739 ; Exit the program.  
740 mov eax, 0  
741 invoke ExitProcess
```

This code will read a single character from the screen buffer at the top-left corner of the console window.

You can use the `SetConsoleCursorPosition` function to set the cursor position in the screen buffer. For example, the following code shows how to set the cursor position to the middle of the console window:

```
745 ; Get a handle to the console screen buffer.  
746 mov eax, STD_OUTPUT_HANDLE  
747 invoke GetStdHandle  
748 mov ebx, eax  
749  
750 ; Set the cursor position.  
751 mov ecx, 40 ; X coordinate  
752 mov edx, 25 ; Y coordinate  
753 invoke SetConsoleCursorPosition  
754 mov esi, ebx  
755  
756 ; Exit the program.  
757 mov eax, 0  
758 invoke ExitProcess
```

This code will set the cursor position to the middle of the console window.

[SetConsoleTitle](#), [GetConsoleScreenBufferInfo](#), and [SetConsoleWindowInfo](#).

```
665 ; SetConsoleTitle function to change the console window's title
666 .data
667 titleStr BYTE "New Console Title",0
668
669 .code
670 ; Invoke SetConsoleTitle with the specified title string
671 INVOKE SetConsoleTitle, ADDR titleStr
```

This code demonstrates how to change the console window's title using the SetConsoleTitle function.

```
675 ; GetConsoleScreenBufferInfo function to retrieve information about the console window
676 .data
677 consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>
678 outHandle HANDLE ?
679
680 .code
681 ; Invoke GetConsoleScreenBufferInfo to retrieve information about the console window
682 INVOKE GetConsoleScreenBufferInfo, outHandle, ADDR consoleInfo
```

This code shows how to use the **GetConsoleScreenBufferInfo** function to obtain information about the console window, including screen buffer size, cursor position, and other details. The retrieved information is stored in the consoleInfo structure.

```
685 ; SetConsoleWindowInfo function to set the console window's size and position
686 .data
687>windowRect SMALL_RECT <0, 0, 79, 24> ; Example window rectangle
688
689.code
690 ; Invoke SetConsoleWindowInfo to set the console window's size and position
691Invoke SetConsoleWindowInfo, outHandle, TRUE, ADDR windowRect
```

This code demonstrates how to use the **SetConsoleWindowInfo** function to set the size and position of the console window relative to the screen buffer. The `windowRect` structure defines the new window dimensions and position.

## CONSOLE\_SCREEN\_BUFFER\_INFO structure.

Name	Value	Type
consoleInfo	{dwSize={X=0x0078 Y=0x0032 } dwCursorPosition= CONSOLE_SCREEN_BUFFER_INFO {X=0x0078 Y=0x0032 } wAttributes=0x0007 srWindow={Left=0x0000 Top=0x0000 Right=0x004f Bottom=0x0018 } dwMaximumWindowSize={X=0x0078 Y=0x0032 }}	CONSOLE_SCREEN_BUFFER_INFO
dwSize	{X=0x0078 Y=0x0032 }	COORD
X	0x0078	unsigned short
Y	0x0032	unsigned short
dwCursorPosition	{X=0x0014 Y=0x0005 }	COORD
X	0x0014	unsigned short
Y	0x0005	unsigned short
wAttributes	0x0007	unsigned short
srWindow	{Left=0x0000 Top=0x0000 Right=0x004f Bottom=0x0018 }	SMALL_RECT
Left	0x0000	unsigned short
Top	0x0000	unsigned short
Right	0x004f	unsigned short
Bottom	0x0018	unsigned short
dwMaximumWindowSize	{X=0x0078 Y=0x0032 }	COORD
X	0x0078	unsigned short
Y	0x0032	unsigned short

I'll provide a simplified and commented version of the Scroll.asm program:

```
767 INCLUDE Irvine32.inc
768
769 .data
770 message BYTE ": This line of text was written to the screen buffer",0dh,0ah
771 messageSize DWORD ($-message)
772 outHandle HANDLE 0 ; Standard output handle
773 bytesWritten DWORD ?
774 lineNumber DWORD 0
775 windowRect SMALL_RECT <0,0,60,11> ; Left, top, right, bottom
776
777 .code
778 main PROC
779     ; Get the standard output handle
780     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
781     mov outHandle, eax
782
783     .REPEAT
784         ; Display the line number
785         mov eax, lineNumber
786         call WriteDec
787
788         ; Write the message to the console
789         INVOKE WriteConsole, outHandle, ADDR message, messageSize, ADDR bytesWritten, 0
790
791         ; Increment the line number
792         inc lineNumber
793
794     .UNTIL lineNumber > 50
795
796     ; Resize and reposition the console window
797     INVOKE SetConsoleWindowInfo, outHandle, TRUE, ADDR windowRect
```

```
798
799 ; Wait for a key press
800 call ReadChar
801
802 ; Clear the screen buffer
803 call Clrscr
804
805 ; Wait for a second key press
806 call ReadChar
807
808 ; Exit the program
809 INVOKE ExitProcess, 0
810
811 main ENDP
812
813 END main
```

This code simulates scrolling the console window by writing lines of text to the screen buffer and then resizing and repositioning the console window using SetConsoleWindowInfo. After running this program, press a key to trigger the scroll, clear the screen, and exit the program.

Another example:

```
819 INCLUDE Irvine32.inc
820
821 .data
822 consoleInfo CONSOLE_CURSOR_INFO <25, 1> ; Default cursor info
823 outHandle HANDLE 0
824 coord COORD <10, 10> ; New cursor position
825
826 .code
827 main PROC
828     ; Get the standard output handle
829     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
830     mov outHandle, eax
831
832     ; Get the current cursor information
833     INVOKE GetConsoleCursorInfo, outHandle, ADDR consoleInfo
834
835     ; Display the current cursor size and visibility
836     mov eax, consoleInfo.dwSize
837     call WriteDec
838     call WriteString, ADDR " - Cursor Size, Visible: "
839     mov eax, consoleInfo.bVisible
840     call WriteDec
841     call Crlf
842
843     ; Set a new cursor size and visibility
844     mov consoleInfo.dwSize, 50
845     mov consoleInfo.bVisible, TRUE
846     INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo
847
```

```
848 ; Move the cursor to a new position
849 INVOKE SetConsoleCursorPosition, outHandle, ADDR coord
850
851 ; Display a message at the new cursor position
852 call WriteString, ADDR "New Cursor Position"
853
854 ; Wait for a key press
855 call ReadChar
856
857 ; Reset cursor info to the default values
858 mov consoleInfo.dwSize, 25
859 mov consoleInfo.bVisible, TRUE
860 INVOKE SetConsoleCursorInfo, outHandle, ADDR consoleInfo
861
862 ; Move the cursor back to the original position
863 mov coord.X, 0
864 mov coord.Y, 0
865 INVOKE SetConsoleCursorPosition, outHandle, ADDR coord
866
867 ; Display a message at the original cursor position
868 call WriteString, ADDR "Original Cursor Position"
869
870 ; Wait for a key press to exit
871 call ReadChar
872
873 INVOKE ExitProcess, 0
874 main ENDP
875
876 END main
```

This program demonstrates the usage of cursor control functions. It first retrieves the current cursor info, changes the cursor size and visibility, and moves the cursor to a new position. After displaying a message, it resets the cursor to its original state and waits for a key press before exiting.

## ***Setting Text Color***

### **Controlling Text Color in a Console Window**

In a console window, you can control text color using two main methods:

**SetConsoleTextAttribute Function:** This function allows you to set the foreground and background colors for all subsequent text output in the console window. It takes the console output handle and a color attribute as parameters. The color attribute specifies both foreground and background colors and is stored in the low-order byte of the wAttributes parameter.

**WriteConsoleOutputAttribute Function:** This function enables you to set the attributes (including text color) for specific cells in the console screen buffer. You provide an array of attributes, a length, starting coordinates, and a count of the number of cells affected.

**Example Program:** Let's create a simple program that demonstrates how to use these functions to set text colors. In this example, we'll display characters with different colors in a console window:

```
883 ; SetTextColors.asm - Demonstrates setting text colors in a console window
884 INCLUDE Irvine32.inc
885
886 .data
887 outHandle HANDLE ?
888 cellsWritten DWORD ?
889 xyPos COORD <10, 2>
890
891 ; Array of character codes
892 buffer BYTE 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
893 BYTE 16,17,18,19,20
894 BufSize DWORD ($-buffer)
895
896 ; Array of attributes (text colors)
897 attributes WORD 0Fh,0Eh,0Dh,0Ch,0Bh,0Ah,9,8,7,6
898 WORD 5,4,3,2,1,0F0h,0E0h,0D0h,0C0h,0B0h
899
900 .code
901 main PROC
902     ; Get the Console standard output handle
903     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
904     mov outHandle, eax
905
906     ; Set the colors of adjacent cells
907     INVOKE WriteConsoleOutputAttribute, outHandle, ADDR attributes, BufSize, xyPos, ADDR cellsWritten
908
909     ; Write character codes 1 through 20
910     INVOKE WriteConsoleOutputCharacter, outHandle, ADDR buffer, BufSize, xyPos, ADDR cellsWritten
911     INVOKE ExitProcess, 0
912 main ENDP
913 END main
```

This program sets different text colors for characters 1 to 20 and displays them in a console window. The text colors are specified in the attributes array. The characters and their associated colors are written to the console screen buffer, resulting in colorful text output.

This program is a simple example to get you started with text color manipulation in a console window. You can modify the attributes array to set different colors for your text as needed.

## ***Time and WinAPI***

Here's a brief description of each of the time and date-related functions in the Win32 API:

### **CompareFileTime:**

Compares two 64-bit file times to determine their order.

### **DosDateTimeToFileTime:**

Converts MS-DOS date and time values to a 64-bit file time, allowing easy compatibility with older date and time representations.

### **FileTimeToDosDateTime:**

Performs the reverse operation, converting a 64-bit file time to MS-DOS date and time values.

### **FileTimeToLocalFileTime:**

Converts a UTC (universal coordinated time) file time to a local file time, making it suitable for use in the local time zone.

### **FileTimeToSystemTime:**

Converts a 64-bit file time to a SYSTEMTIME structure, providing detailed date and time information.

**GetFileTime:**

Retrieves the date and time when a file was created, last accessed, and last modified.

**GetLocalTime:**

Retrieves the current local date and time, useful for obtaining the current local system time.

**GetSystemTime:**

Retrieves the current system date and time in UTC format, allowing for consistent time information across different time zones.

**GetSystemTimeAdjustment:**

Determines whether the system is applying periodic time adjustments to its time-of-day clock, important for handling time adjustments like daylight saving time.

**GetSystemTimeAsFileTime:**

Retrieves the current system date and time in UTC format, providing a 64-bit file time.

**GetTickCount:**

Retrieves the number of milliseconds that have elapsed since the system was started, useful for measuring time intervals or system uptime.

**GetTimeZoneInformation:**

Retrieves the current time-zone parameters, allowing you to obtain information about the system's time zone.

**LocalFileTimeToFileTime:**

Converts a local file time to a file time based on UTC, enabling the conversion of local time to a more universal format.

**SetFileTime:**

Sets the date and time that a file was created, last accessed, or last modified, allowing for the modification of file timestamps.

**SetLocalTime:**

Sets the current local time and date on the system, making it useful for adjusting the system's local time settings.

**SetSystemTime:**

Sets the current system time and date, enabling adjustments to the system's time settings.

**SetSystemTimeAdjustment:**

Allows you to enable or disable periodic time adjustments to the system's time-of-day clock.

**SetTimeZoneInformation:**

Sets the current time-zone parameters, providing control over the system's time zone settings.

**SystemTimeToFileTime:**

Converts a SYSTEMTIME structure to a 64-bit file time, allowing for the transformation of detailed date and time information into a universal format.

**SystemTimeToTzSpecificLocalTime:**

Converts a UTC time to a specified time zone's corresponding local time, useful when you need to adjust time information to a specific time zone.

*Here's the combined MASM program that includes the SYSTEMTIME structure, GetLocalTime, SetLocalTime, and GetTickCount functions, as well as a stopwatch timer:*

```
0917 INCLUDE Irvine32.inc
0918 INCLUDE macros.inc
0919
0920 .data
0921 sysTime SYSTEMTIME <> ; SYSTEMTIME structure
0922 startTime DWORD ? ; Start time for the stopwatch timer
0923
0924 .code
0925 main PROC
0926     ; Get the current local time.
0927     INVOKE GetLocalTime, ADDR sysTime
0928     ; Display the current local time.
0929     call DisplayTime
0930     ; Set the local time to a specific value.
0931     ; For example, you can set it to January 1, 2023, 12:00:00.
0932     mov sysTime.wYear, 2023
0933     mov sysTime.wMonth, 1
0934     mov sysTime.wDay, 1
0935     mov sysTime.wHour, 12
0936     mov sysTime.wMinute, 0
0937     mov sysTime.wSecond, 0
0938     INVOKE SetLocalTime, ADDR sysTime
0939     ; Get the current local time again after setting it.
0940     INVOKE GetLocalTime, ADDR sysTime
0941     ; Display the updated local time.
0942     call DisplayTime
0943     ; Start a stopwatch timer.
0944     INVOKE GetTickCount
0945     mov startTime, eax
0946     ; Perform some calculations to simulate a time-consuming operation.
0947     mov ecx, 10000100h
```

```
0948 L1:  
0949     imul ebx  
0950     imul ebx  
0951     imul ebx  
0952     loop L1  
0953  
0954 ; Get the current tick count and calculate elapsed time.  
0955 INVOKE GetTickCount  
0956 sub eax, startTime  
0957  
0958 ; Display the elapsed time.  
0959 call WriteDec  
0960 mWrite <" milliseconds have elapsed", 0dh, 0ah>  
0961  
0962 exit  
0963 main ENDP  
0964
```

```
0965 DisplayTime PROC
0966     ; Display the current local time.
0967     mWrite "Current Local Time: "
0968     call WriteDec, sysTime.wMonth
0969     mWrite <"/", 0>
0970     call WriteDec, sysTime.wDay
0971     mWrite <"/", 0>
0972     call WriteDec, sysTime.wYear
0973     mWrite <" ", 0>
0974     call WriteDec, sysTime.wHour
0975     mWrite <":", 0>
0976     call WriteDec, sysTime.wMinute
0977     mWrite <":", 0>
0978     call WriteDec, sysTime.wSecond
0979     mWrite <" (Day of Week: ", 0>
0980     call WriteDec, sysTime.wDayOfWeek
0981     mWrite <")", 0dh, 0ah>
0982     ret
0983 DisplayTime ENDP
0984 END main
0985
```

*Here's an explanation of the program in paragraph format:*

The program starts by defining a structure called SYSTEMTIME, which is used to hold information

about date and time. It includes fields like year, month, day of the week, day of the month, hours, minutes, seconds, and milliseconds. This structure is essential for working with date and time-related functions in the Windows API.

The program utilizes the `GetLocalTime` function, a Windows API function that retrieves the current local date and time according to the system's clock. It takes a single parameter, a pointer to a `SYSTEMTIME` structure, where it stores the current date and time values. This function is essential for obtaining the current time for further processing.

On the other hand, the `SetLocalTime` function is another Windows API function used to set the system's local date and time. It also takes a `SYSTEMTIME` structure as a parameter, but this time, it contains the desired date and time values. By calling this function, you can modify the system's date and time settings programmatically.

The program also incorporates the `GetTickCount` function, which is used to measure the number of milliseconds that have passed since the system started. This function doesn't require any parameters and returns the elapsed time in the `EAX` register. It's particularly useful for timing operations and determining the duration of processes.

There's a custom procedure in the program called `DisplayTime`, which serves to display the various components of a `SYSTEMTIME` structure, such as the year, month, day, hour, minute, second, and day of the week. This procedure uses different write functions to display these components on the console.

The program's main procedure is the entry point. It first calls `GetLocalTime` to retrieve and display the current local time. After that, it sets the local time to a specific value, allowing you to modify the date and time as needed. The program then calls `GetLocalTime` again to retrieve and display the updated local time. To simulate a time-consuming operation, the program performs calculations in a loop. Before and after this loop, it uses `GetTickCount` to measure the elapsed time and displays it.

In summary, this program showcases how to work with date and time in a Windows environment using the SYSTEMTIME structure and relevant Windows API functions. It demonstrates retrieving and displaying the current local time, setting the local time, and measuring elapsed time using GetTickCount. You can adjust the date and time values as necessary for your specific requirements.

=====

The **Sleep function** is a part of the Win32 API that allows programs to introduce pauses or delays. This can be useful for controlling the timing of various operations in a program.

The function takes a parameter that specifies the length of time to sleep, and then it puts the processor into a low-power state until the specified time has elapsed.

The **GetDateTime procedure** is a convenient utility to retrieve date and time information. It returns the number of 100-nanosecond intervals that have elapsed since January 1, 1601.

The procedure generally follows these steps:

It calls a function like GetLocalTime, which populates a SYSTEMTIME structure with the current date and time information.

It converts this SYSTEMTIME structure to a FILETIME structure using the SystemTimeToFileTime function. Then, it copies the resulting FILETIME structure to a 64-bit quadword.

The FILETIME structure is used to divide a 64-bit quadword into two doublewords.

The GetDateTime procedure, which receives a pointer to a 64-bit quadword variable as an argument, is responsible for storing the current date and time in the specified variable in the FILETIME format used by Win32.

In simpler terms, the **Sleep function** allows programs to pause for a specified period of time, while the **GetDateTime procedure** allows programs to retrieve the current date and time.

Both functions are useful for controlling the timing of various operations in Win32 applications.

```
0988 ; Sleep Function
0989 Sleep PROTO,
0990 dwMilliseconds:DWORD
0991
0992 ; GetDateTime Procedure
0993 GetDateTime PROC,
0994 pStartTime:PTR QWORD
0995 LOCAL sysTime:SYSTEMTIME, flTime:FILETIME
0996
0997 ; Get the system local time
0998 INVOKE GetLocalTime,
0999 ADDR sysTime
1000
1001 ; Convert the SYSTEMTIME to FILETIME
1002 INVOKE SystemTimeToFileTime,
1003 ADDR sysTime,
1004 ADDR flTime
1005
1006 ; Copy the FILETIME to a 64-bit integer
1007 mov esi, pStartTime
1008 mov eax, flTime.loDateTime
1009 mov DWORD PTR [esi], eax
1010 mov eax, flTime.hiDateTime
1011 mov DWORD PTR [esi+4], eax
1012 ret
1013 GetDateTime ENDP
```

The Sleep function allows you to introduce time delays, and the GetDateTime procedure retrieves the current date and time and stores it in a 64-bit quadword. This code can be integrated into your assembly programs as needed.

```
1019 ; Sleep for 1 second
1020 mov eax, 1000 ; 1000 milliseconds = 1 second
1021 call sleep
1022
1023 ; Continue execution
```

The eax register is used to specify the length of time to sleep. The call sleep instruction then calls the sleep function. Once the sleep function has returned, the program will continue execution.

It is important to note that the sleep function can be interrupted by certain events, such as a timer interrupt. If this happens, the program will resume execution immediately, even if the specified sleep time has not yet elapsed.

Here are some additional things to keep in mind when using the sleep function in MASM:  
The sleep function is typically implemented as a system call.

This means that it must be executed in a privileged mode, such as kernel mode. The sleep function can be blocked by other processes.

This means that if another process is holding the kernel lock, the sleep function will not be able to execute until the other process releases the lock.

The sleep function can cause the processor to enter a low-power state. This can save power, but it can also delay the execution of other programs.

Overall, the sleep function is a powerful tool that can be used to control the execution of a program. However, it is important to be aware of the limitations of the function and to use it carefully.

## ***Calling 64-bit WinAPI function in Masm***

To call a 64-bit Windows API function in MASM, you must follow these steps:

Reserve at least 32 bytes of shadow space by subtracting 32 from the stack pointer (RSP) register.

Make sure RSP is aligned on a 16-byte address boundary.

Place the first four arguments in the following registers, from left to right:

RCX, RDX, R8, and R9.

Push additional arguments on the runtime stack.

Call the function using the call instruction.

Restore RSP to its original value by adding the same value to it that was subtracted before the function call.

The system function will return a 64-bit integer value in RAX. Here is an example of how to call the 64-bit WriteConsoleA function:

```
1027 .data
1028     STD_OUTPUT_HANDLE EQU -11
1029     consoleOutHandle QWORD ?
1030
1031 .code
1032     sub rsp, 40 ; reserve shadow space & align RSP
1033     mov rcx, STD_OUTPUT_HANDLE
1034     mov rdx, message ; pointer to the string
1035     mov r8, message_length ; length of the string
1036     lea r9, bytesWritten
1037     mov qword ptr [rsp + 4 * SIZEOF QWORD], 0 ; (always zero)
1038     call WriteConsoleA
1039     add rsp, 40 ; restore RSP
```

The WriteConsoleA function takes five arguments:

- The console handle.
- A pointer to the string to write.
- The length of the string to write.
- A pointer to the variable that will store the number of bytes written.
- A dummy zero parameter.
- The bytesWritten variable is used to store the number of bytes that were actually written.

Once you have called the WriteConsoleA function, you can check the value of the bytesWritten variable to see how many bytes were written.

**To write a graphical Windows application, you need to:**

**Include the necessary libraries and header files.** This includes the kernel32.lib and user32.lib libraries, as well as a header file that contains structures, constants, and function prototypes used by the program.

**Create a main window.** This is done using the CreateWindowEx() function. Display the main window. This is done using the ShowWindow() function. Respond to mouse events. This is done by handling the WM\_MOUSEMOVE and WM\_LBUTTONDOWN messages. Display message boxes. This is done using the MessageBox() function.

Here is a simple example of a graphical Windows application in assembly language:



```
include "graphwin.inc"

.data
className db "WinApp", 0
instance HANDLE
window HANDLE

.code
start:
; Register the window class
invoke RegClassEx, addr className

; Create the main window
invoke CreateWindowEx, 0, addr className, addr className, WS_OVERLAPPEDWINDOW, 0, 0,
CW_USEDEFAULT, CW_USEDEFAULT, HWND_DESKTOP, 0, instance, 0
```

```
mov window, eax

; Show the main window
invoke ShowWindow, window, SW_SHOW

; Message loop
messageLoop:
    invoke GetMessage, addr msg, 0, 0, 0
    cmp eax, -1
    je end

; Translate and dispatch the message
    invoke TranslateMessage, addr msg
    invoke DispatchMessage, addr msg

    jmp messageLoop

end:
    invoke ExitProcess, 0
```

This program creates a simple window with the title "WinApp". The window fills the screen and is centered on the desktop.

The program also handles mouse events and displays a message box when the user clicks the left mouse button.

To build and run the program, you can use the following steps:

Create a new assembly language project in Visual Studio. Add the following files to the project: WinApp.asm GraphWin.inc Add the kernel32.lib and user32.lib libraries to the project. Set the subsystem to Windows (/SUBSYSTEM:WINDOWS).

Build and run the program. When you run the program, you will see a simple window with the title "WinApp". If you click the left mouse button, the program will display a message box.

Ignore this program, it's just a trial program:

```
1085 RECT STRUCT
1086     left DWORD ?
1087     top DWORD ?
1088     right DWORD ?
1089     bottom DWORD ?
1090 RECT ENDS
1091 .data
1092     rect1 RECT <10, 20, 100, 150> ; Define a RECT structure with specific coordinates
1093 .code
1094 main PROC
1095     mov eax, rect1.left      ; Access the left coordinate
1096     mov ebx, rect1.top       ; Access the top coordinate
1097     mov ecx, rect1.right     ; Access the right coordinate
1098     mov edx, rect1.bottom    ; Access the bottom coordinate
1099 ; Now you can use these values for various tasks
1100 ; For example, you can calculate the width and height of the rectangle
1101     sub ecx, eax            ; Width = right - left
1102     sub edx, ebx            ; Height = bottom - top
1103 ; Display the width and height
1104     call DisplayWidthAndHeight
1105 ; You can also modify the coordinates or dimensions as needed
1106     add rect1.left, 5         ; Move the left side 5 units to the right
1107     sub rect1.right, 10        ; Shrink the width by 10 units
1108 ; Now the rect1 structure has been updated
1109     exit
1110 main ENDP
1111 DisplayWidthAndHeight PROC
1112     ; Display the width and height
1113     ; You can implement this function as needed
1114     ret
1115 DisplayWidthAndHeight ENDP
```

**Rectangle struct:** The RECT structure is used to define the boundaries of a rectangle. It includes four members that determine the position and size of the rectangle. The "left" member holds the X-coordinate of the left side of the rectangle, while the "top" member stores the Y-coordinate of the top side.

Similarly, the "right" and "bottom" members hold values for the right and bottom sides of the rectangle, respectively. Together, these members specify the dimensions and position of the rectangle on the screen.

```
1133 RECT STRUCT
1134     left DWORD ?
1135     top DWORD ?
1136     right DWORD ?
1137     bottom DWORD ?
1138 RECT ENDS
```

The **MSGStruct structure** defines the data needed for an MS-Windows message:

```
1122 MSGStruct STRUCT  
1123     msgWnd      DWORD ?  
1124     msgMessage   DWORD ?  
1125     msgWparam    DWORD ?  
1126     msgLparam    DWORD ?  
1127     msgTime      DWORD ?  
1128     msgPt        POINT <>  
1129 MSGStruct ENDS
```

The **WNDCLASS** structure is used to define a window class in a Windows application. Every window within a program is associated with a specific class, and the program must register this class with the operating system before the main window can be displayed. Here is the WNDCLASS structure:

```
1144 WNDCLASS STRUC  
1145     style      DWORD ?      ; Window style options  
1146     lpfnWndProc DWORD ?      ; Pointer to the Window Procedure function  
1147     cbClsExtra   DWORD ?      ; Extra shared memory  
1148     cbWndExtra   DWORD ?      ; Number of extra bytes  
1149     hInstance    DWORD ?      ; Handle to the current program  
1150     hIcon       DWORD ?      ; Handle to the icon  
1151     hCursor      DWORD ?      ; Handle to the cursor  
1152     hbrBackground DWORD ?      ; Handle to the background brush  
1153     lpszMenuName DWORD ?      ; Pointer to the menu name  
1154     lpszClassName DWORD ?      ; Pointer to the window class name  
1155 WNDCLASS ENDS
```

This structure holds various parameters and settings for a window class, including its appearance, behavior, and how it interacts with the operating system. Registering a window class allows the program to create and manage windows of that class.

*Here's a concise summary of the parameters within the WNDCLASS structure:*

**style:** A combination of style options, such as WS\_CAPTION and WS\_BORDER, that determine the window's appearance and behavior.

**lpfnWndProc:** A function pointer that specifies the program's function for processing event messages triggered by the user.

**cbClsExtra:** Refers to shared memory used by all windows belonging to the class, and it can be set to null if not needed.

**cbWndExtra:** Specifies the number of extra bytes to allocate following the window instance.

**hInstance:** Holds a handle to the current program instance, allowing the class to be associated with this instance of the program.

**hIcon and hCursor:** Hold handles to icon and cursor resources for the current program, influencing the visual elements used in the window.

**hbrBackground:** Holds a handle to a background brush, which determines the window's background color.

**lpszMenuName:** Points to a menu name, defining the menu associated with the window.

**lpszClassName:** Points to a null-terminated string containing the window's class name, allowing the

program to identify and manage windows of this class effectively.

## The MessageBox Function

The MessageBox function is the easiest way to display text in a Windows application. It displays a simple message box with a text message, a caption, and one or more buttons. The buttons can be used to get the user's response to the message.

## *The WinMain Procedure*

The WinMain procedure is the startup procedure for every Windows application. It is responsible for the following tasks:

- Getting a handle to the current program.
- Loading the program's icon and mouse cursor.
- Registering the program's main window class and identifying the procedure that will process event messages for the window.
- Creating the main window.
- Showing and updating the main window.
- Beginning a loop that receives and dispatches messages.
- The loop continues until the user closes the application window.

## *The WinProc Procedure*

The WinProc procedure receives and processes all event messages relating to a window.

Most events are initiated by the user by clicking and dragging the mouse, pressing keyboard keys, and so on.

and so on.

The WinProc procedure's job is to decode each message, and if the message is recognized, to carry out application-oriented tasks relating to the message.

The following example code shows a simple Windows application that uses the MessageBox function to display a message to the user when the user clicks the left mouse button.

```
#include <windows.h>

LRESULT CALLBACK WinProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg) {
        case WM_LBUTTONDOWN:
            MessageBox(hWnd, "You clicked the left mouse button!", "Message Box Example", MB_OK);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hWnd;

    // Register the window class.
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
```

```
wc.lpfnWndProc = WinProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wc.lpszMenuName = NULL;
wc.lpszClassName = "MyWindowClass";

if (!RegisterClassEx(&wc)) {
    return 0;
}

// Create a window.
hWnd = CreateWindowEx(0, "MyWindowClass", "Message Box Example", WS_OVERLAPPEDWINDOW, 100, 100,
300, 200, NULL, NULL, hInstance, NULL);
if (!hWnd) {
    return 0;
}

// Show the window.
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

// Wait for a key press.
MSG msg;
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

getchar(); // Wait for a key press
return msg.wParam;
}
```

If you compile and run this code, you should see a window with the title "Message Box Example". Click the left mouse button in the window, and a message box should appear with the text "You clicked the left mouse button!".

It seems like you've provided a portion of code and information related to writing a graphical Windows application. This code appears to be written in assembly language, specifically designed for Windows programming. Here's a breakdown of the code and related information:

**ErrorHandler Procedure:** This procedure is called when an error occurs during the registration and creation of the program's main window. It performs several tasks, including retrieving the system error number, formatting the system error message, displaying it in a popup message box, and freeing the memory used by the error message string.

**Program Listing:** This part of the code defines various data structures and constants for the Windows application, such as window titles, messages, and class names. These are used throughout the application for display and interaction.

**MainWin WNDCLASS Structure:** It defines the window class structure for the application. It includes settings like window procedure, icon, cursor, and other attributes.

**WinMain Procedure:** This is the entry point of the application. It initializes various components, including registering the window class, creating the main window, displaying messages, and entering a message-handling loop.

**Message Handling Loop:** The code enters a continuous message-handling loop using GetMessage, processes messages with DispatchMessage, and continues until there are no more messages. When there are no more messages, it exits the program using ExitProcess.

This code is a part of a Windows application written in assembly language, which creates a main

window, displays messages, and handles messages in a loop. If you have any specific questions or need further details about this code, please let me know, and I'll address them accordingly.

```
; Windows Application (WinApp.asm)
; This program displays a resizable application window and
; several popup message boxes. Special thanks to Tom Joyce
; for the first version of this program.
.386
.model flat,STDCALL
INCLUDE GraphWin.inc

; ====== DATA ======
.data
AppLoadMsgTitle BYTE "Application Loaded",0
AppLoadMsgText BYTE "This window displays when the WM_CREATE "
                 BYTE "message is received",0
PopupTitle BYTE "Popup Window",0
PopupText BYTE "This window was activated by a "
            BYTE "WM_LBUTTONDOWN message",0
GreetTitle BYTE "Main Window Active",0
GreetText BYTE "This window is shown immediately after "
            BYTE "CreateWindow and UpdateWindow are called.",0
CloseMsg BYTE "WM_CLOSE message received",0
ErrorTitle BYTE "Error",0
WindowName BYTE "ASM Windows App",0
className BYTE "ASMWin",0

; Define the Application's Window class structure.
MainWin WNDCLASS <NULL,WinProc,NULL,NULL,NULL,NULL,NULL, \
           COLOR_WINDOW,NULL,className>

msg MSGStruct <>
winRect RECT <>
hMainWnd DWORD ?
hInstance DWORD ?
```

```
; ====== CODE ======
.code
WinMain PROC
    ; Get a handle to the current process.
    INVOKE GetModuleHandle, NULL
    mov hInstance, eax
    mov MainWin.hInstance, eax

    ; Load the program's icon and cursor.
    INVOKE LoadIcon, NULL, IDI_APPLICATION
    mov MainWin.hIcon, eax
    INVOKE LoadCursor, NULL, IDC_ARROW
    mov MainWin.hCursor, eax

    ; Register the window class.
    INVOKE RegisterClass, ADDR MainWin
    .IF eax == 0
        call ErrorHandler
        jmp Exit_Program
    .ENDIF

    ; Create the application's main window.
    INVOKE CreateWindowEx, 0, ADDR className,
        ADDR WindowName, MAIN_WINDOW_STYLE,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        CW_USEDEFAULT, NULL, NULL, hInstance, NULL
    ; If CreateWindowEx failed, display a message and exit.
    .IF eax == 0
        call ErrorHandler
        jmp Exit_Program
    .ENDIF

    ; Save the window handle, show and draw the window.
    mov hMainWnd, eax
    INVOKE ShowWindow, hMainWnd, SW_SHOW
    INVOKE UpdateWindow, hMainWnd
```

```
; Display a greeting message.  
INVOKE MessageBox, hMainWnd, ADDR GreetText,  
           ADDR GreetTitle, MB_OK  
  
; Begin the program's continuous message-handling loop.  
Message_Loop:  
    ; Get next message from the queue.  
    INVOKE GetMessage, ADDR msg, NULL,NULL,NULL  
    ; Quit if no more messages.  
    .IF eax == 0  
        jmp Exit_Program  
.ENDIF  
    ; Relay the message to the program's WinProc.  
    INVOKE DispatchMessage, ADDR msg  
    jmp Message_Loop  
  
Exit_Program:  
    INVOKE ExitProcess,0  
  
WinMain ENDP  
  
; The ErrorHandler Procedure  
; This procedure handles errors during window registration and creation.  
ErrorHandler PROC  
    ; Call GetLastError to retrieve the system error number.  
    INVOKE GetLastError  
    ; Call FormatMessage to retrieve the appropriate system-formatted error message string.  
    INVOKE FormatMessage, FORMAT_MESSAGE_FROM_SYSTEM, NULL, eax, \  
          0, ADDR ErrorTitle, 256, 0  
    ; Call MessageBox to display a popup message box containing the error message string.  
    INVOKE MessageBox, NULL, eax, ADDR ErrorTitle, MB_OK  
    ; Call LocalFree to free the memory used by the error message string.  
    INVOKE LocalFree, eax  
    ret  
ErrorHandler ENDP
```

This combined code includes the ErrorHandler procedure and the WinMain procedure along with the relevant data and constants. It's ready to be used in a Windows application written in assembly language.

## *WinMain Procedure:*

WinMain is the entry point of the application, where the program execution begins.

It starts by getting a handle to the current process using GetModuleHandle and stores it in hInstance.

It loads the program's icon and cursor using LoadIcon and LoadCursor functions and assigns them to the MainWin structure, which defines the window class.

The window class is registered using RegisterClass.

If the registration fails (indicated by eax == 0), the ErrorHandler procedure is called, and the program exits.

If the registration is successful, the application's main window is created using CreateWindowEx.

If this fails, it also calls the ErrorHandler procedure and exits.

After creating the main window, it's displayed and updated with ShowWindow and UpdateWindow functions.

A greeting message is displayed in a message box.

The program enters a message-handling loop using GetMessage, processes the messages with DispatchMessage, and continues until there are no more messages.

## *Exit\_Program Label:*

The Exit\_Program label is used to handle the program's exit. It's reached when there are no more messages in the message loop, and it invokes ExitProcess to terminate the program.

This code sets up the application's main window, registers its class, and enters the message-handling loop.

It handles basic application initialization, including window creation and message processing.

The Exit\_Program label is used for a clean program exit when there are no more messages to process.

```
=====
;  
WinProc PROC,  
hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD  
;  
; The application's message handler, which handles  
; application-specific messages. All other messages  
; are forwarded to the default Windows message  
; handler.  
;  
    mov eax, localMsg  
    .IF eax == WM_LBUTTONDOWN  
        ; Mouse button?
```

```
    INVOKE MessageBox, hWnd, ADDR PopupText,
          ADDR PopupTitle, MB_OK
    jmp WinProcExit
.ELSEIF eax == WM_CREATE
    ; Create window?
    INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
          ADDR AppLoadMsgTitle, MB_OK
    jmp WinProcExit
.ELSEIF eax == WM_CLOSE
    ; Close window?
    INVOKE MessageBox, hWnd, ADDR CloseMsg,
          ADDR WindowName, MB_OK
    INVOKE PostQuitMessage, 0
    jmp WinProcExit
.ELSE
    ; Other message?
    INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
    jmp WinProcExit
.ENDIF
WinProcExit:
    ret
WinProc ENDP
```

```
;-----
ErrorHandler PROC
; Display the appropriate system error message.
;-----
.data
pErrorMsg DWORD ?
; Pointer to error message
messageID DWORD ?

.code
INVOKE GetLastError
; Returns message ID in EAX
mov messageID, eax
; Get the corresponding message string.
```

```
INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
    FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId, NULL,
    ADDR pErrorMsg, NULL, NULL

; Display the error message.
INVOKE MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,
    MB_ICONERROR + MB_OK

; Free the error message string.
INVOKE LocalFree, pErrorMsg
ret
ErrorHandler ENDP

END WinMain
```

This code combines the WinProc and ErrorHandler procedures with your existing code and includes appropriate comments for clarity. It's ready for use in a Windows application written in assembly language.

## *WinProc Procedure:*

WinProc is a procedure that serves as the message handler for the Windows application. It takes four parameters: hWnd (a handle to the window), localMsg (the message ID), wParam, and lParam (message-specific data).

The purpose of WinProc is to handle application-specific messages. It checks the localMsg parameter to determine the type of message received.

If localMsg is equal to WM\_LBUTTONDOWN, it displays a message box indicating that the left mouse button was clicked.

If localMsg is equal to WM\_CREATE, it displays a message box indicating that the window was created.

If localMsg is equal to WM\_CLOSE, it displays a message box indicating that the window is about to close and triggers the application to quit.

If the message is none of the above, it forwards the message to the default Windows message handler using DefWindowProc.

### *ErrorHandler Procedure:*

ErrorHandler is a procedure designed to handle errors during window registration and creation.

It first declares data and code sections for its implementation.

Inside, it uses the GetLastError function to retrieve the system error number and stores it in messageID.

It then calls FormatMessage to retrieve the corresponding system-formatted error message string, which is allocated dynamically and stored in pErrorMsg.

Next, it displays the error message in a message box with the title "Error."

Finally, it frees the memory used by the error message string using LocalFree.

The provided code integrates these two procedures with your existing code to handle messages and errors in your Windows application written in assembly language.

It adds comments to explain each part of the code for better understanding and maintainability.

This code is now ready to be used in your application.

## ***Dynamic Memory***

**Dynamic memory allocation** is the process of allocating memory during the execution of a program.

This is in contrast to **static memory allocation**, where memory is allocated at compile time.

There are two main ways to perform dynamic memory allocation in assembly language:

**Using system calls:** This involves making calls to the operating system to allocate and deallocate memory.

**Implementing a heap manager:** This involves implementing your own data structure and algorithms to manage memory allocation and deallocation.

The example program in the section you provided uses the first method. It makes system calls to the Windows operating system to allocate and deallocate memory.

Here is a summary of the steps involved in dynamic memory allocation using system calls:  
Make a system call to allocate memory.

This will return a pointer to the allocated memory block. Use the allocated memory block.

Make a system call to deallocate the memory block when you are finished using it. The following

table lists some of the Win32 API functions that can be used for dynamic memory allocation:

Function	Description
GetProcessHeap	Returns a 32-bit integer handle to the program's existing heap area in EAX. If the function succeeds, it returns a handle to the heap in EAX. If it fails, the return value in EAX is NULL.
HeapAlloc	Allocates a block of memory from a heap. If it succeeds, the return value in EAX contains the address of the memory block. If it fails, the returned value in EAX is NULL.
HeapCreate	Creates a new heap and makes it available to the calling program. If the function succeeds, it returns a handle to the newly created heap in EAX. If it fails, the return value in EAX is NULL.
HeapDestroy	Destroys the specified heap object and invalidates its handle. If the function succeeds, the return value in EAX is nonzero.
HeapFree	Frees a block of memory previously allocated from a heap, identified by its address and heap handle. If the block is freed successfully, the return value is nonzero.
HeapReAlloc	Reallocates and resizes a block of memory from a heap. If the function succeeds, the return value is a pointer to the reallocated memory block. If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.
HeapSize	Returns the size of a memory block previously allocated by a call to HeapAlloc or HeapReAlloc. If the function succeeds, EAX contains the size of the allocated memory block, in bytes. If the function fails, the return value is SIZE_T – 1. (SIZE_T equals the maximum number of bytes to which a pointer can point.)

Here is a summary of the heap functions you provided:

`GetProcessHeap()` returns a handle to the current process's default heap.

`HeapCreate()` creates a new private heap for the current process.

`HeapDestroy()` destroys an existing private heap.

`HeapAlloc()` allocates a block of memory from a heap.

`HeapFree()` frees a block of memory previously allocated from a heap.

### **When to use which function:**

Use `GetProcessHeap()` if you are content to use the default heap owned by the current program.

Use `HeapCreate()` to create a new private heap if you need more control over memory management.

Use `HeapDestroy()` to destroy a private heap when you are finished using it. Use `HeapAlloc()` to allocate memory from a heap.

Use `HeapFree()` to free memory that was allocated from a heap.

Here is an example of how to use the `HeapAlloc()` and `HeapFree()` functions to allocate and free a block of memory from a heap:

```
1225 ; Create a new private heap.  
1226 INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX  
1227  
1228 ; Allocate a block of memory from the heap.  
1229 INVOKE HeapAlloc, hHeap, 0, 1000  
1230  
1231 ; Use the allocated memory block.  
1232 ; ...  
1233  
1234 ; Free the allocated memory block.  
1235 INVOKE HeapFree, hHeap, 0, pArray  
1236  
1237 ; Destroy the private heap.  
1238 INVOKE HeapDestroy, hHeap
```

It is important to note that dynamic memory allocation should be used carefully to avoid memory leaks. A memory leak occurs when a program allocates memory but does not free it when it is finished using it. Memory leaks can lead to performance problems and eventually cause the program to crash.

Here's the complete program:

```
1245 ; Heap Test #1 (HeapTest1.asm)
1246 INCLUDE Irvine32.inc
1247 ; This program uses dynamic memory allocation to allocate and
1248 ; fill an array of bytes.
1249
1250 .data
1251 ARRAY_SIZE = 1000
1252 FILL_VAL EQU 0FFh
1253 hHeap HANDLE ?
1254 ; handle to the process heap
1255 pArray DWORD ?
1256 ; pointer to block of memory
1257
1258 .code
1259 main PROC
1260 INVOKE GetProcessHeap
1261 ; get handle to the program heap
1262 .IF eax == NULL
1263 ; if failed, display message
1264     call WriteWindowsMsg
1265     jmp quit
1266 .ELSE
1267     mov hHeap,eax
1268     ; success
1269 .ENDIF
1270
```

```
1271 call allocate_array
1272 jnc array0k
1273 ; failed (CF = 1)?
1274 call WriteWindowsMsg
1275 call Crlf
1276 jmp quit
1277
1278 array0k:
1279 ; ok to fill the array
1280 call fill_array
1281 call display_array
1282 call Crlf
1283 ; free the array
1284 INVOKE HeapFree, hHeap, 0, pArray
1285
1286 quit:
1287     exit
1288
1289 main ENDP
```

```
1291 ;-----  
1292 allocate_array PROC USES eax  
1293;  
1294 ; Dynamically allocates space for the array.  
1295 ; Receives: EAX = handle to the program heap  
1296 ; Returns: CF = 0 if the memory allocation succeeds.  
1297 ;-----  
1298 INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, ARRAY_SIZE  
1299 .IF eax == NULL  
1300     stc  
1301 ; return with CF = 1  
1302 .ELSE  
1303     mov pArray,eax  
1304 ; save the pointer  
1305     clc  
1306 ; return with CF = 0  
1307 .ENDIF  
1308     ret  
1309  
1310 allocate_array ENDP  
1311
```

```
1312 ;-----  
1313 fill_array PROC USES ecx edx esi  
1314;  
1315 ; Fills all array positions with a single character.  
1316 ; Receives: nothing  
1317 ; Returns: nothing  
1318 ;-----  
1319 mov ecx,ARRAY_SIZE  
1320 ; loop counter  
1321 mov esi,pArray  
1322 ; point to the array  
1323 L1:  
1324 mov BYTE PTR [esi],FILL_VAL  
1325 ; fill each byte  
1326 inc esi  
1327 ; next location  
1328 loop L1  
1329 ret  
1330  
1331 fill_array ENDP  
1332
```

```
1333 ;-----  
1334 display_array PROC USES eax ebx ecx esi  
1335 ;  
1336 ; Displays the array  
1337 ; Receives: nothing  
1338 ; Returns: nothing  
1339 ;-----  
1340 mov ecx,ARRAY_SIZE  
1341 ; loop counter  
1342 mov esi,pArray  
1343 ; point to the array  
1344 L1:  
1345 mov al,[esi]  
1346 ; get a byte  
1347 mov ebx,TYPE BYTE  
1348 call WriteHexB  
1349 ; display it  
1350 inc esi  
1351 ; next location  
1352 loop L1  
1353 ret  
1354  
1355 display_array ENDP  
1356  
1357 END main
```

The HeapTest1.asm program is an assembly language example that showcases dynamic memory allocation and manipulation in the Windows environment.

The code demonstrates how to allocate memory from the heap, fill that memory with specific values, and display the allocated memory's contents.

The program starts with the .data section, where constants and variables are defined. It specifies the size of the array to be allocated, which is set to 1000 bytes, and the value used to fill the array, which is 0FFh.

The .code section begins with the main procedure. In this procedure, the program performs the following tasks:

It calls the GetProcessHeap function to obtain a handle to the default heap owned by the current process.

This is where memory allocations will be made. If obtaining the heap handle fails (resulting in a NULL handle), the program calls the WriteWindowsMsg function to display an error message and then exits.

If the GetProcessHeap call is successful, the obtained heap handle is stored in the hHeap variable for later use.

The program then proceeds to allocate memory for an array by calling the allocate\_array procedure.

If memory allocation fails (indicated by the Carry Flag being set), it calls the WriteWindowsMsg function to display an error message and exits.

If allocation is successful, the pointer to the allocated memory is saved in the pArray variable.

After successful allocation, the program calls the fill\_array procedure, which fills the allocated memory with a specified value (0FFh in this case).

Following the memory filling, the program calls the display\_array procedure to display the contents of the allocated memory in hexadecimal format.

After displaying the memory contents, the program frees the allocated memory by invoking the HeapFree function.

The program then proceeds to the quit label, where it invokes the Exit system call to terminate the program.

In summary, HeapTest1.asm demonstrates the process of dynamic memory allocation in assembly language within the Windows environment.

It allocates memory from the default process heap, fills that memory with specific values, displays the memory's contents, and finally releases the allocated memory.

The program uses the GetProcessHeap function to obtain the default heap handle and the HeapAlloc and HeapFree functions for memory allocation and deallocation, respectively.

Let's move on to heaptest2.asm:

```
1360 ; Heap Test #2 (Heaptst2.asm)
1361 INCLUDE Irvine32.inc
1362
1363 .data
1364 HEAP_START = 2000000      ; 2 MByte
1365 HEAP_MAX = 400000000     ; 400 MByte
1366 BLOCK_SIZE = 500000      ; 0.5 MByte
1367 hHeap HANDLE ?
1368 pData DWORD ?
1369 str1 BYTE 0dh, 0ah, "Memory allocation failed", 0dh, 0ah, 0
1370
1371 .code
1372 main PROC
1373     ; Create a new heap with specified size limits
1374     INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
1375     .IF eax == NULL
1376         ; Failed to create heap
1377         call WriteWindowsMsg
1378         call Crlf
1379         jmp quit
1380     .ELSE
1381         mov hHeap, eax
1382         ; Success: store the heap handle
1383     .ENDIF
1384
1385     mov ecx, 2000      ; Loop counter
1386
```

```
1387 L1:  
1388     call allocate_block  
1389     ; Allocate a block  
1390  
1391     .IF Carry?  
1392         ; Allocation failed  
1393         mov edx, OFFSET str1  
1394         ; Display error message  
1395         call WriteString  
1396         jmp quit  
1397     .ELSE  
1398         ; Allocation successful  
1399         mov al, '.'  
1400         ; Show progress with a dot  
1401         call WriteChar  
1402     .ENDIF  
1403  
1404     loop L1  
1405  
1406 quit:  
1407     ; Destroy the heap  
1408     INVOKE HeapDestroy, hHeap  
1409     .IF eax == NULL  
1410         ; Failed to destroy heap  
1411         call WriteWindowsMsg  
1412         call Crlf  
1413     .ENDIF  
1414  
1415     exit  
1416 main ENDP  
1417
```

```
1417  
1418 allocate_block PROC USES ecx  
1419     ; Allocate a block and fill it with all zeros  
1420     INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, BLOCK_SIZE  
1421     .IF eax == NULL  
1422         stc  
1423         ; Return with CF = 1 (allocation failed)  
1424     .ELSE  
1425         mov pData, eax  
1426         ; Save the pointer to the allocated memory  
1427         clc  
1428         ; Return with CF = 0 (allocation succeeded)  
1429     .ENDIF  
1430     ret  
1431 allocate_block ENDP  
1432  
1433 free_block PROC USES ecx  
1434     ; Free a previously allocated block  
1435     INVOKE HeapFree, hHeap, 0, pData  
1436     ret  
1437 free_block ENDP  
1438  
1439 END main
```

heap management.

It aims to allocate large blocks of memory repeatedly until the specified heap size limit is reached. The code is divided into sections for clarity.

The data section, defined using the .data directive, starts by declaring constants and variables.

**HEAP\_START** is set to 2 megabytes (2MB), representing the initial heap size.

**HEAP\_MAX** is set to 400 megabytes (400MB), indicating the maximum heap size.

**BLOCK\_SIZE** is set to 0.5 megabytes (0.5MB), representing the size of memory blocks to be allocated.

The program uses **hHeap** to store the handle to the custom heap and **pData** to hold the pointer to the allocated memory. **str1** is a string that will be used to display an error message in case of allocation failure.

The .code section contains the main procedure, labeled **main PROC**. It begins by invoking the **HeapCreate** function to create a new heap with specified initial and maximum sizes.

If the creation of the heap fails (resulting in a NULL heap handle), the program calls the **WriteWindowsMsg** function to display an error message and then jumps to the **quit** label to exit.

In case of a successful heap creation, the handle to the custom heap is stored in the **hHeap** variable for later use.

A loop is initiated using **ecx** as a loop counter, set to 2000 iterations. The purpose of this loop is to repeatedly allocate memory blocks.

Within the loop, the program calls the `allocate_block` procedure. This procedure uses the `HeapAlloc` function to allocate memory from the custom heap.

If memory allocation fails (indicated by the Carry Flag being set), the program displays an error message using `str1`, calls `WriteString` to print the message, and jumps to the `quit` label to exit.

If memory allocation is successful, a dot ('.') is displayed on the screen as a progress indicator, indicating a successful memory allocation.

The program continues the loop until all 2000 iterations are completed, each time allocating a memory block.

After the loop finishes, the program reaches the `quit` label, where it invokes `HeapDestroy` to destroy the custom heap.

If `HeapDestroy` fails (returns `NULL`), an error message is displayed using `WriteWindowsMsg`, and the program exits using the `exit` system call.

In summary, `HeapTest2.asm` showcases dynamic memory allocation using custom heap management. It repeatedly allocates memory blocks until a specified heap size limit is reached.

The program uses functions like `HeapCreate`, `HeapAlloc`, and `HeapDestroy` to manage custom heaps and memory allocation.

Progress is indicated by displaying dots for successful allocations, and any errors are communicated using appropriate error messages.

The program demonstrates the flexibility of heap management in assembly language within the Windows environment.

environment.

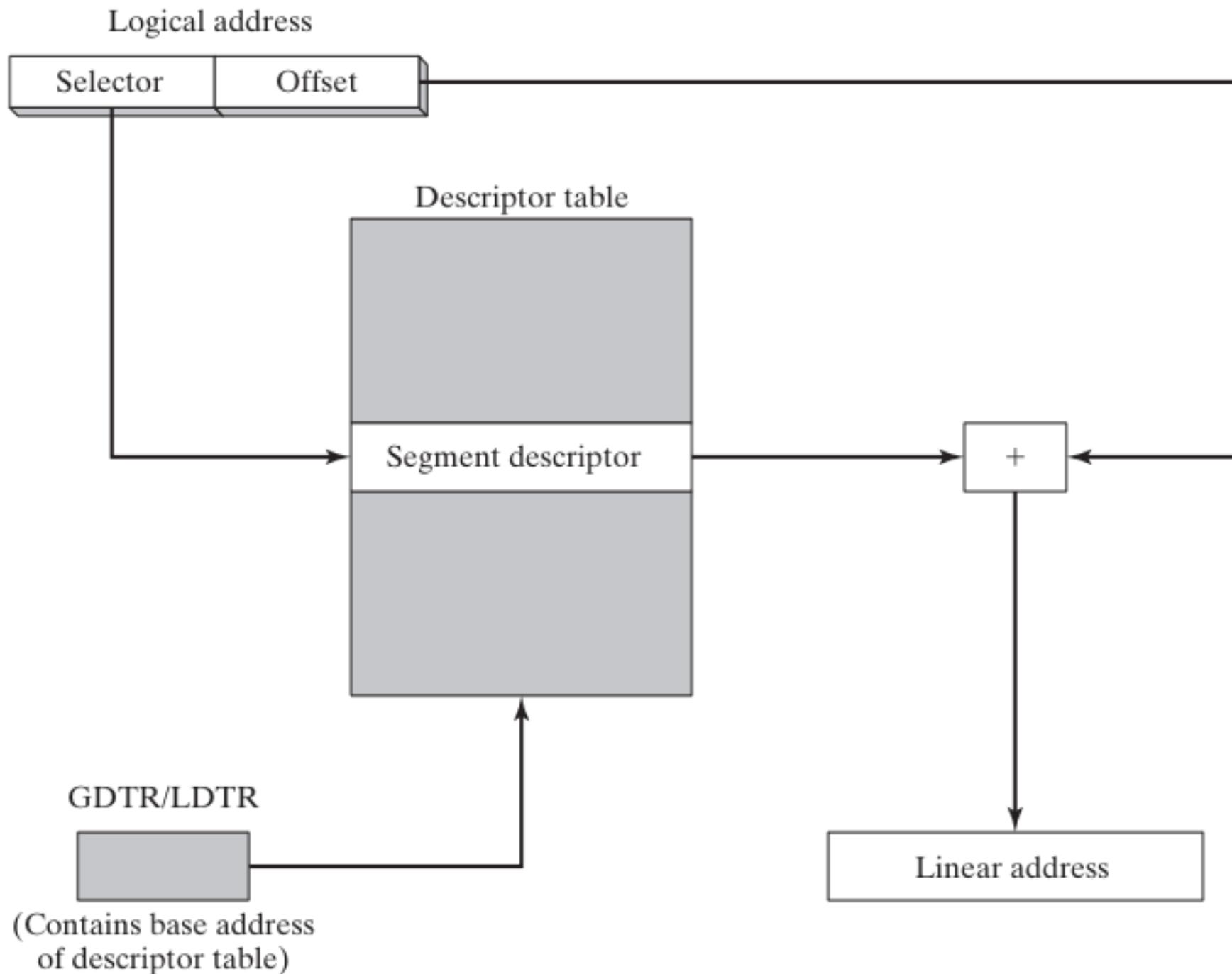
## **x86 Memory Management**

Logical addresses and linear addresses are two different ways of addressing memory in an x86 processor.

A **logical address** is a combination of a segment selector and a 32-bit offset. The segment selector is a 16-bit value that identifies a segment descriptor, which in turn contains information about a memory segment. The offset is a 32-bit value that identifies a location within the segment.

A **linear address** is a 32-bit value that uniquely identifies a location in memory. It is calculated by adding the segment base address to the offset.

The x86 processor uses a two-step process to translate logical addresses to linear addresses:



The **segment selector** is used to index the segment descriptor table (GDT or LDT) to obtain the segment descriptor.

The **segment base address** is added to the offset to produce the linear address. The following diagram shows the process of translating logical addresses to linear addresses:

```
1442 Logical address = segment selector + offset  
1443 Segment base address = segment descriptor table[segment selector]  
1444 Linear address = segment base address + offset
```

Once the linear address has been calculated, the processor can use it to access memory directly.

### *Example*

Suppose we have a program that has a variable at offset 200h in a segment with the segment selector value 0x1000. The segment descriptor table contains a segment descriptor for this segment with a base address of 0x100000.

To access this variable, the processor would first calculate the linear address:

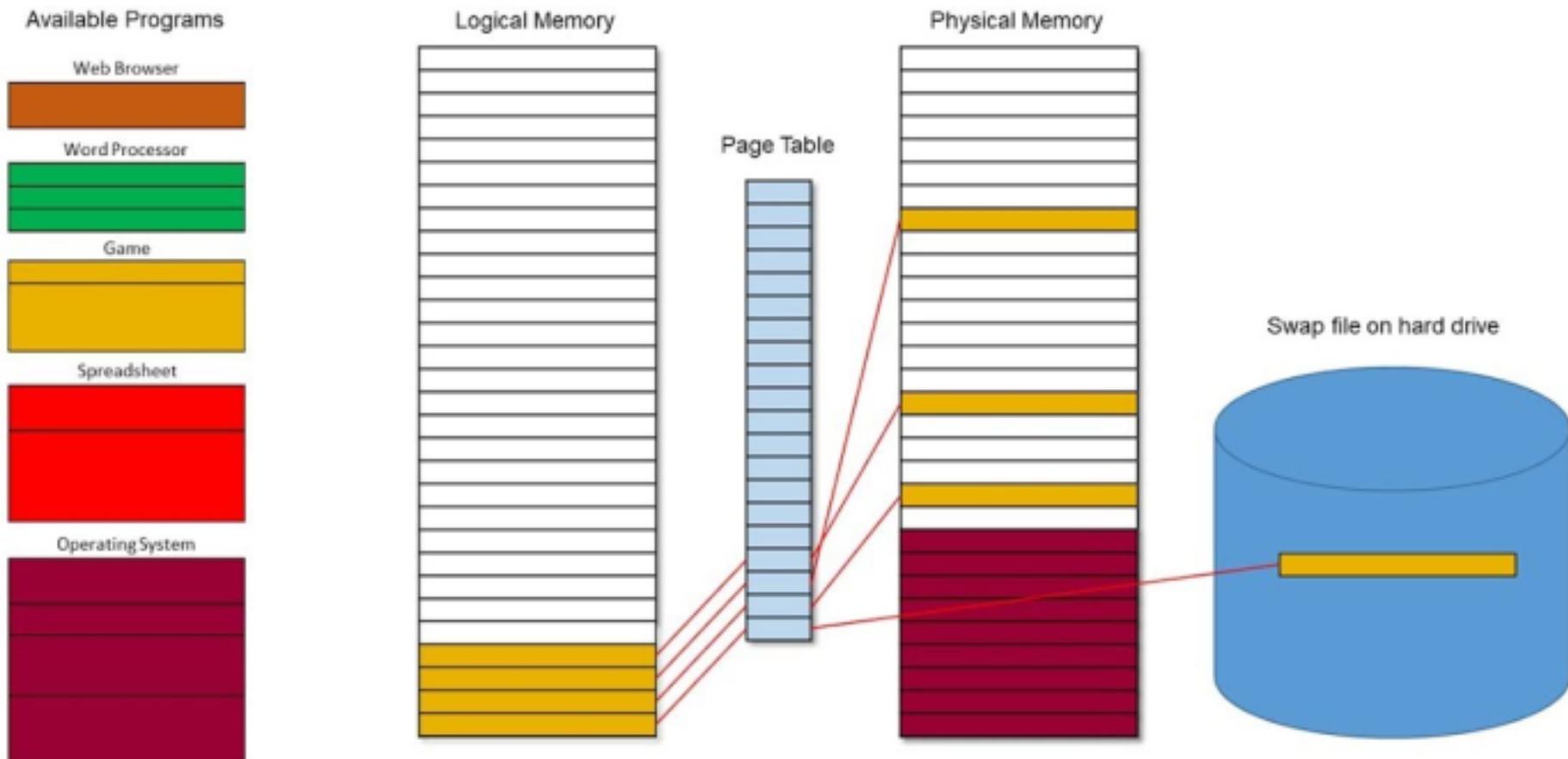
```
1448 Linear address = segment base address + offset  
1449 Linear address = 0x100000 + 200h  
1450 Linear address = 0x100200
```

Once the linear address has been calculated, the processor can use it to access the variable at memory location 0x100200.

## *Paging*

**Paging** is a memory management technique that allows the operating system to divide physical memory into pages.

# Paged Memory



**Pages** are typically 4KB in size, but can also be 2MB or larger.

When a program needs to access memory, the operating system converts the program's linear address to a physical address using a page table.

a physical address using a page table.

The **page table** is a data structure that maps linear addresses to physical addresses.

The following diagram shows the process of translating linear addresses to physical addresses using a page table:

1454 Linear address = page table index + page offset

1455 Physical address = page table[page table index] + page offset

The page table index is the upper 20 bits of the linear address. The page offset is the lower 12 bits of the linear address.

The operating system can use paging to implement a number of features, such as virtual memory and memory protection.

## Conclusion

Logical addresses and linear addresses are two different ways of addressing memory in an x86 processor. Logical addresses are used by programs, while linear addresses are used by the processor.

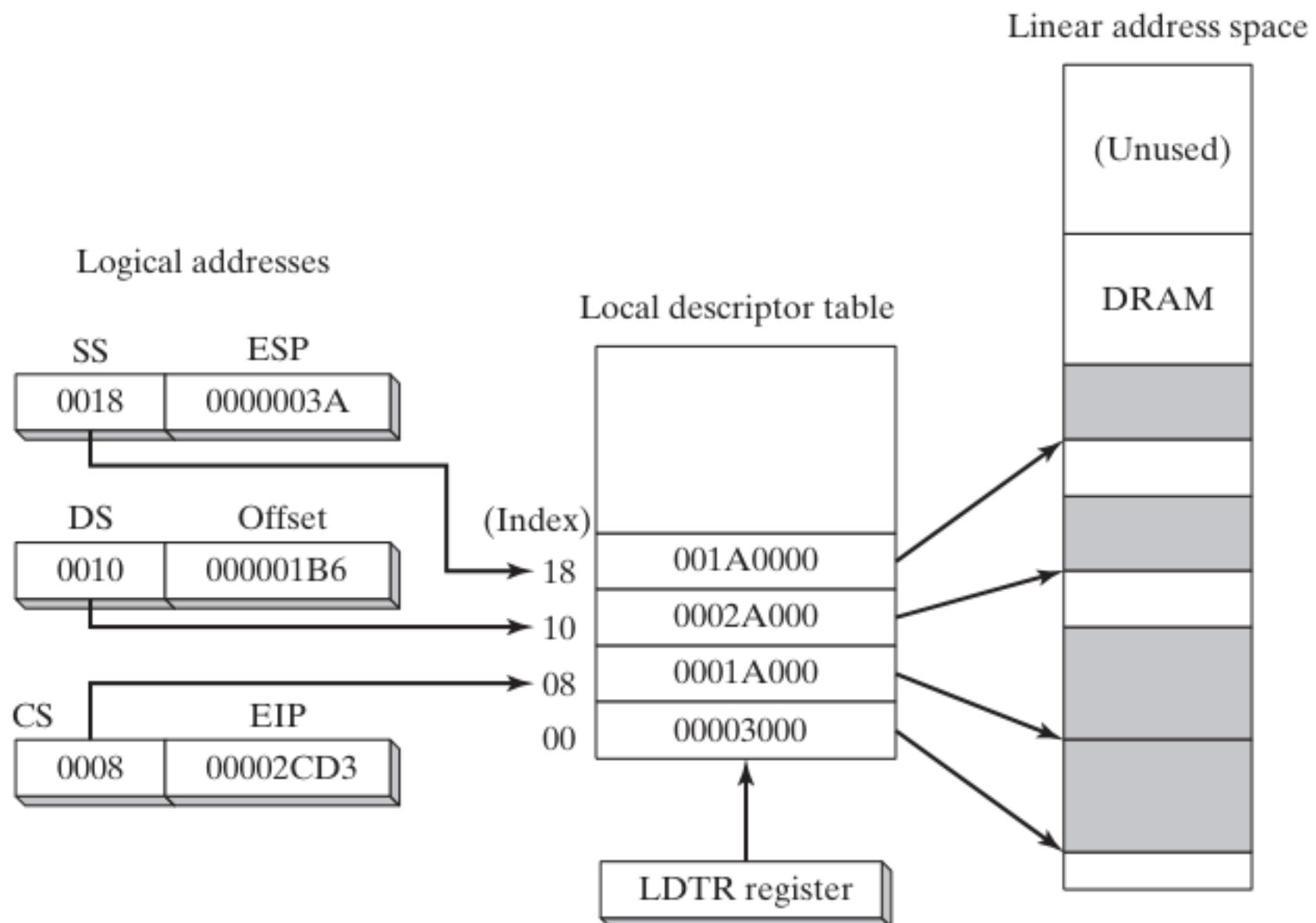
The processor translates logical addresses to linear addresses using segment descriptors. Linear addresses can then be translated to physical addresses using a page table.

---

## Descriptor tables

=====

## Indexing into a local descriptor table.



Descriptor tables are data structures that contain information about memory segments. A segment is a

variable-sized area of memory that is used by a program to store code or data. There are two types of descriptor tables:

**Global Descriptor Table (GDT):** The GDT contains segment descriptors for all of the segments that are used by the system.

**Local Descriptor Table (LDT):** Each task or process has its own LDT, which contains segment descriptors for the segments that are used by that task or process.

Segment descriptors contain information about a segment, such as its base address, size, and access rights. The processor uses this information to translate logical addresses to linear addresses.

A logical address is a combination of a segment selector and a 32-bit offset. The segment selector is a 16-bit value that identifies a segment descriptor in the GDT or LDT. The offset is a 32-bit value that identifies a location within the segment.

The processor calculates the linear address by adding the segment base address to the offset. The linear address is then used to access memory directly.

Suppose we have a program that has a variable at offset 200h in a segment with the segment selector value 0x1000.

The GDT contains a segment descriptor for this segment with a base address of 0x100000.

To access this variable, the processor would first calculate the linear address:

```
1459 Linear address = segment base address + offset  
1460 Linear address = 0x100000 + 200h  
1461 Linear address = 0x100200
```

Once the linear address has been calculated, the processor can use it to access the variable at memory location 0x100200.

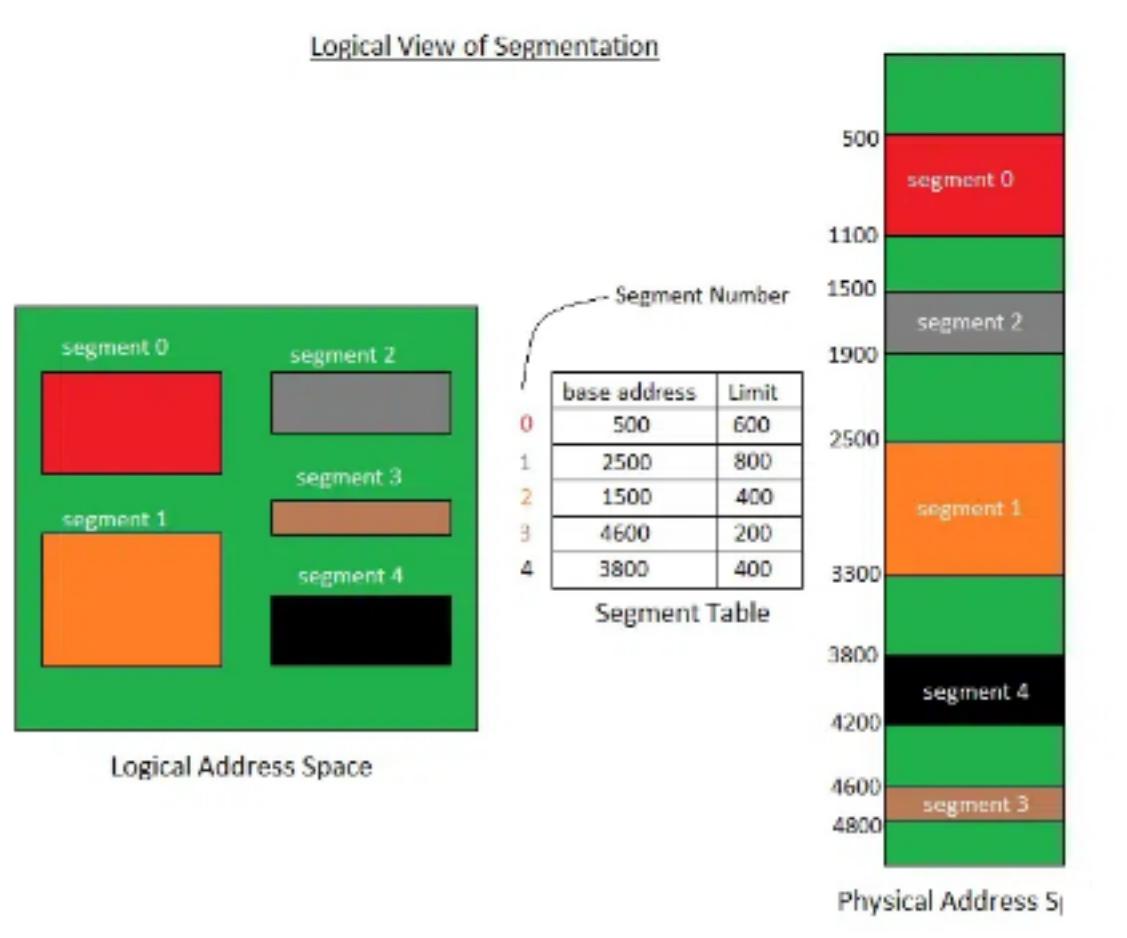
## *Segment descriptor details*

In addition to the segment's base address, the segment descriptor contains the following information:

**Segment limit:** The segment limit specifies the maximum size of the segment. If a program tries to access a memory location outside of the segment limit, a processor fault is generated.



**Segment type:** The segment type specifies the type of data that is stored in the segment. For example, a code segment contains code, and a data segment contains data.



**Access rights:** The access rights specify which operations are allowed on the segment. For example, a read-only segment can only be read, and a write-only segment can only be written to. The processor uses this information to ensure that programs do not access memory in an unauthorized way.



*Segment descriptors in x86 processors contain a number of fields that control how the segment is used, including:*

**Base address:** The starting address of the segment in the linear address space.

**Privilege level:** The privilege level required to access the segment.

**Segment type:** The type of segment, such as code, data, or stack.

**Segment present flag:** Indicates whether the segment is present in memory.

**Granularity flag:** Determines whether the segment limit is interpreted in bytes or 4096-byte units.

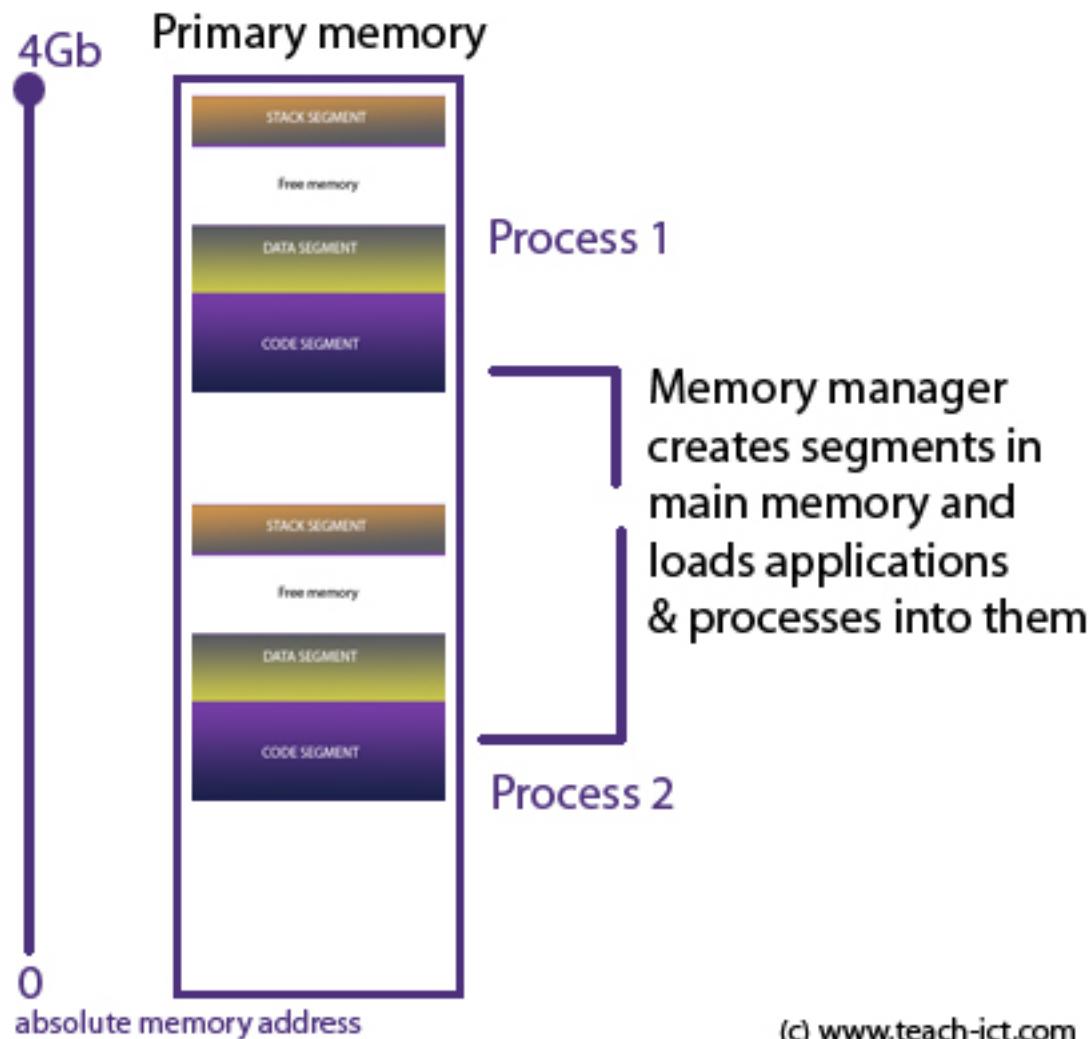
**Segment limit:** The maximum size of the segment.

The **protection level field** is used to protect operating system data from access by application programs.

Each segment can be assigned a **privilege level between 0 and 3**, where 0 is the most privileged and 3 is the least privileged.

If a program with a **higher privilege level** tries to access a segment with a lower privilege level, a processor fault is generated.

This prevents application programs from **accidentally or maliciously** modifying operating system data.



Memory manager creates segments in main memory and loads applications & processes into them

(c) www.teach-ict.com

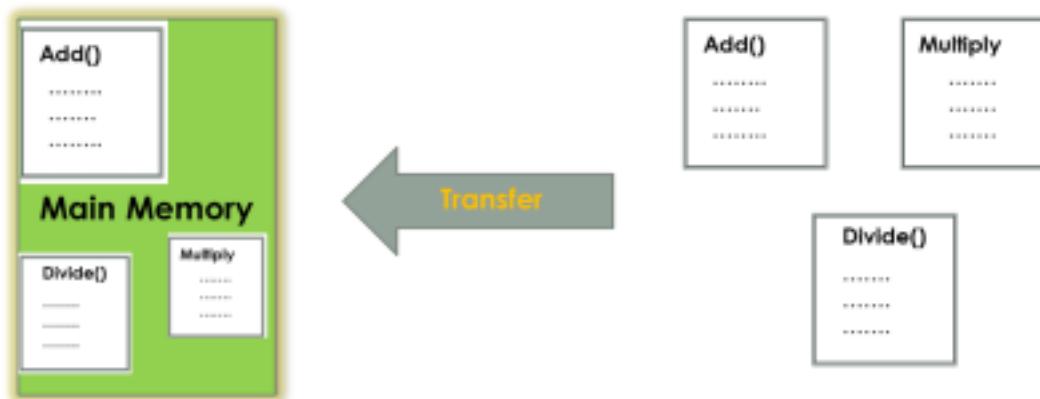
So,

The **segment type field** is used to specify the type of data that is stored in the segment and the type of access that is allowed. For example, a code segment can only be executed, and a data segment can only be read or written to.

The **segment present flag** is used to indicate whether the segment is currently present in memory. If the flag is set, the segment is present in memory and can be accessed. If the flag is not set, the segment is not present in memory and cannot be accessed.

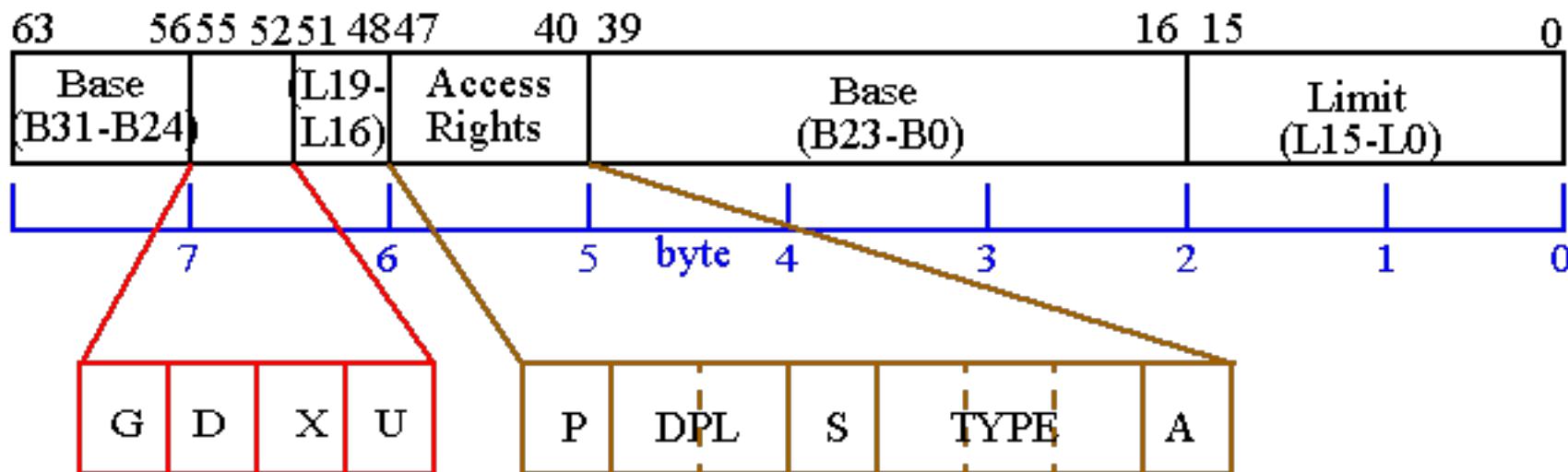
The **granularity flag** is used to determine how the segment limit field is interpreted. If the flag is set, the segment limit is interpreted in 4096-byte units. If the flag is not set, the segment limit is interpreted in bytes.

## SEGMENTATION CONCEPT



The **segment limit field** specifies the maximum size of the segment. If a program tries to access a memory location outside of the segment limit, a processor fault is generated.

**Segment descriptors** are an important part of memory management in x86 processors. They allow the processor to translate logical addresses to linear addresses and to ensure that programs do not access memory in an unauthorized way.



Described segment descriptor:

Field	Size (bits)	Description
Base address	32	The starting address of the segment in the linear address space.
Privilege level	2	The privilege level required to access the segment.
Segment type	2	The type of segment, such as code, data, or stack.
Segment present flag	1	Indicates whether the segment is present in memory.
Granularity flag	1	Determines whether the segment limit is interpreted in bytes or 4096-byte units.
Segment limit	20	The maximum size of the segment.

The segment descriptor image also shows the following:

The segment descriptor table (GDT) is located at address 0x00000000. The segment selector is a 16-bit value that identifies a segment descriptor in the GDT.

The linear address is a 32-bit value that identifies a memory location in the linear address space.

The processor uses the segment selector to index the GDT to obtain the segment descriptor.

The segment descriptor is then used to calculate the linear address of the memory location.

=====

## ***Page translation:***

=====

Page translation is the process of converting a linear address to a physical address in an x86 processor when paging is enabled.

A linear address is a 32-bit value that uniquely identifies a location in memory. A physical address is also a 32-bit value, but it identifies a location in physical memory.

Paging allows the operating system to divide physical memory into pages, which are typically 4KB in size. The operating system then uses a page table to map virtual addresses to physical addresses.

The page table is a data structure that contains one entry for each page in the virtual address space.

Each entry contains the physical address of the page and its access rights.

When the processor needs to access a memory location, it first translates the linear address to a physical address using the page table.

The processor does this by looking up the page table entry for the linear address. The page table entry contains the physical address of the page and its access rights.

If the page table entry is valid, the processor uses the physical address to access the memory location.

If the page table entry is not valid, the processor generates a page fault.

### ***Steps in page translation***

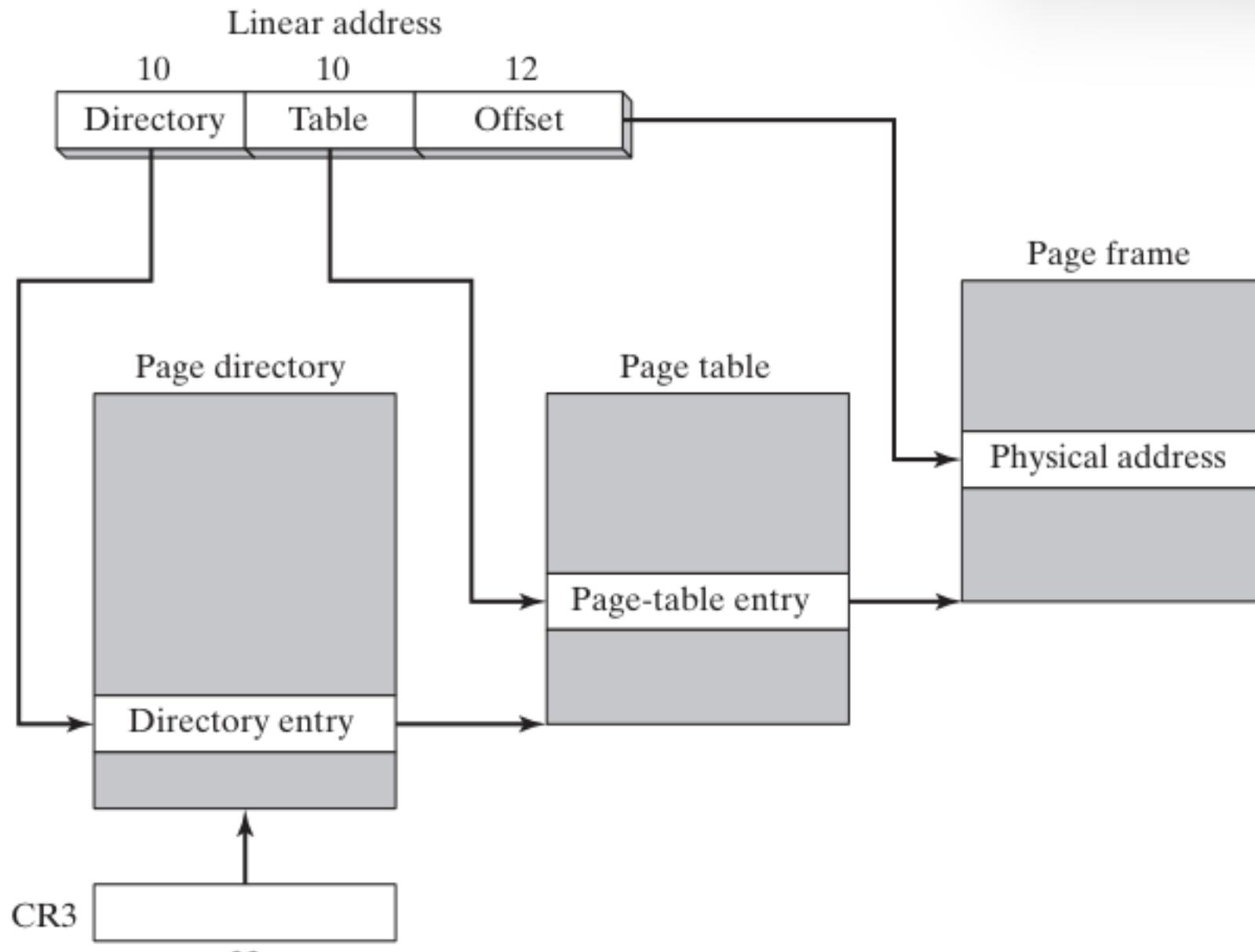
Let's describe the page table image first:

The linear address references a location in the linear address space. The 10-bit directory field in the linear address is an index to a page-directory entry. The page-directory entry contains the base address of a page table.

The 10-bit table field in the linear address is an index into the page table identified by the page-directory entry. The page-table entry at that position contains the base location of a page in physical memory.

The 12-bit offset field in the linear address is added to the base address of the page, generating the exact physical address of the operand.

## Translating linear address to physical address.



The following steps are carried out by the processor when translating a linear address to a physical address:

The processor splits the linear address into three fields:

**Directory field:** The directory field is the upper 10 bits of the linear address.

**Table field:** The table field is the middle 10 bits of the linear address. **Offset field:** The offset field is the lower 12 bits of the linear address.

The processor uses the directory field to index the page directory. The page directory is a table of 1024 4-byte entries.

Each entry in the page directory points to a page table. The processor uses the table field to index the page table pointed to by the directory entry.

The page table is also a table of 1024 4-byte entries. Each entry in the page table points to a physical page frame.

The processor adds the offset field to the physical address of the page frame pointed to by the page table entry. This results in the physical address of the memory location. Example:

Suppose we have a linear address of 0x12345678. The **directory field** would be 0x1234, the **table field** would be 0x5678, and the **offset field** would be 0x123456.

The processor would first use the directory field to index the page directory. The page directory entry at index 0x1234 would contain the address of the page table.

The processor would then use the table field to index the page table. The page table entry at index 0x5678 would contain the physical address of the page frame.

Finally, the processor would add the offset field to the physical address of the page frame. This results in the physical address of the memory location, which is 0x12345678.

## *Conclusion*

The operating system has the option of using a single page directory for all running programs and tasks, or one page directory per task, or a combination of the two.

Page translation is an important part of memory management in x86 processors. It allows the operating system to divide physical memory into pages and to map virtual addresses to physical addresses.

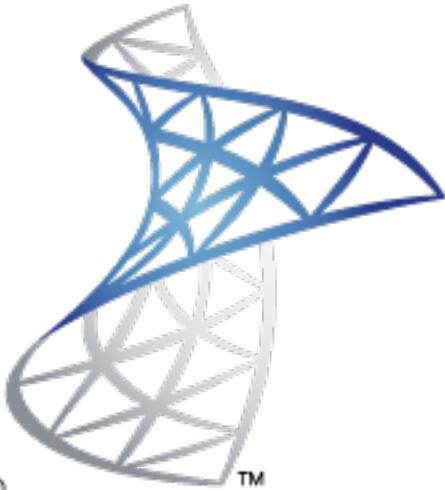
This allows the operating system to implement virtual memory and to protect memory from unauthorized access.

=====

## *Windows Virtual Machine Manager*

=====

**Windows Virtual Machine Manager (VMM)** is the 32-bit protected mode operating system at the core of Windows. It is responsible for creating, running, monitoring, and terminating virtual machines. It also manages memory, processes, interrupts, and exceptions.



Microsoft®

# System Center Virtual Machine Manager

VMM uses a single 32-bit flat model address space at privilege level 0. This means that all of the virtual machines, the VMM itself, and any virtual devices all share the same address space.

The VMM creates two global descriptor table (GDT) entries for each virtual machine, one for code and one for data. These segments are fixed at linear address 0.

VMM provides multithreaded, preemptive multitasking. This means that it can run multiple applications simultaneously by sharing CPU time between the virtual machines in which the applications run.

[How VMM handles memory management](#)

VMM uses a technique called paging to manage memory. Paging divides physical memory into pages, which are typically 4KB in size. VMM then uses a page table to map virtual addresses to physical addresses.

Each virtual machine has its own page table. The page table tells the processor which physical page contains a particular virtual address.

When a virtual machine tries to access a memory location, the processor first looks up the page table entry for that virtual address.

If the page table entry is valid, the processor uses it to access the physical memory location. If the page table entry is not valid, the processor generates a page fault.

Page faults are handled by the VMM. When a page fault occurs, the VMM checks to see if the virtual address is valid. If it is, the VMM loads the corresponding physical page into memory and updates the page table entry.

If the virtual address is not valid, the VMM generates an exception.

### Benefits of using VMM

VMM offers a number of benefits, including:

**Isolation:** VMM isolates virtual machines from each other, so that a failure in one virtual machine does not affect other virtual machines.



**Security:** VMM can be used to implement security features such as access control and encryption.

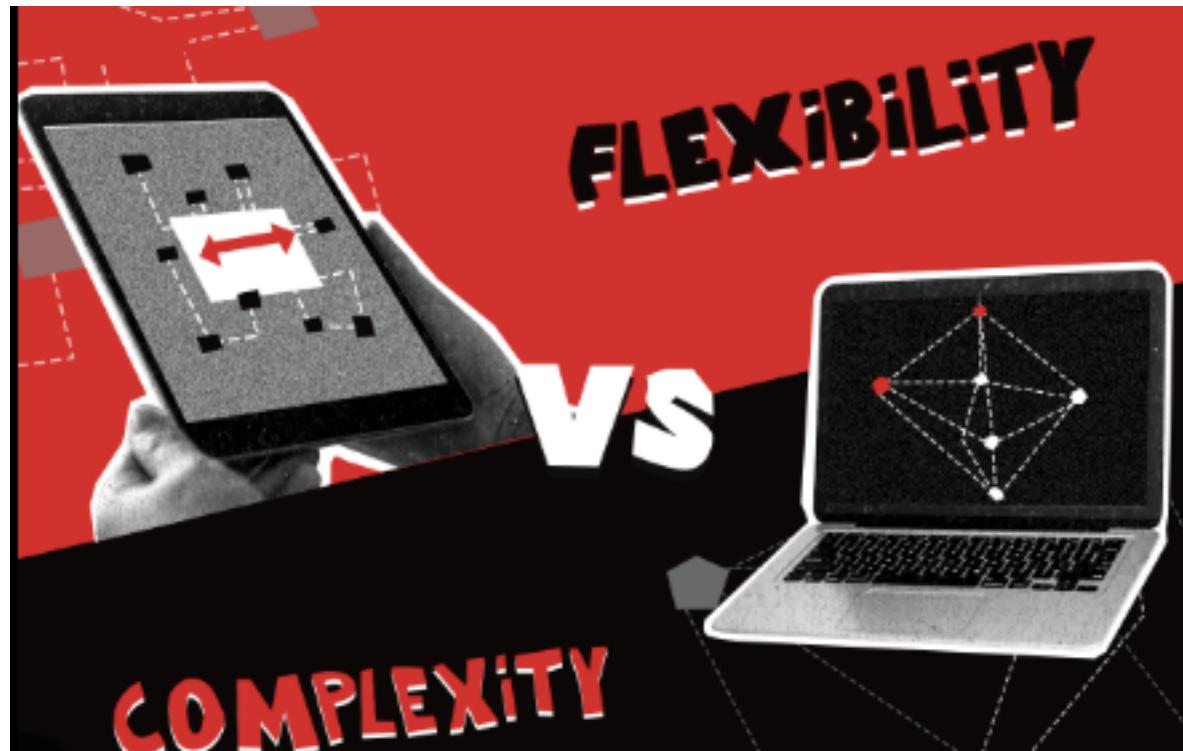


**Performance:** VMM can improve the performance of applications by running them in separate virtual

machines.



**Flexibility:** VMM can be used to create and manage different types of virtual machines, such as servers, desktops, and test environments.



Windows Virtual Machine Manager is a powerful tool that can be used to create, run, and manage virtual machines. VMM offers a number of benefits, including isolation, security, performance, and flexibility.

## *Floating Point Arithmetic in ASM*

Skip ...

# **Floating Point Binary Representation**

These data conversions were dealt with in the first chapters notes.

**Floating-point binary representation** is a method for encoding decimal numbers using three primary components: a **sign**, a **significand (or mantissa)**, and an **exponent**. These components work together to represent numbers like  $-1.23154 \times 10^5$  in a binary format.

- **Sign:** The sign indicates whether the number is positive or negative.

In the example  $-1.23154 \times 10^5$ , the sign is negative.

- **Significand (Mantissa):** The significand is the main part of the number and contains its value.

It's similar to the decimal part of a real number. In the example, the significand is 1.23154.

- **Exponent:** The exponent indicates the order of magnitude by which the significand should be multiplied.

It determines whether the number is very large or very small. In this case, the exponent is 5, which means the significant 1.23154 should be multiplied by  $10^5$  to get the final value.

This representation is commonly used in computer systems for storing and processing real numbers with both integer and fractional parts. It allows for a wide range of values, from very small to very large, and is essential for scientific and engineering calculations.

---

x86 processors employ the IEEE 754-1985 standard for Binary Floating-Point Arithmetic, which defines **three distinct binary floating-point storage formats**.

These formats are designed to represent real numbers with varying precision and characteristics. Here, we'll focus on the single-precision format, as it's the most common and widely used.

## Single-Precision Format (32 bits)

In the single-precision format, 32 bits are used to represent a floating-point number. These bits are organized with the most significant bit (MSB) on the left, and they are stored in memory in little-endian order, meaning that the least significant bit (LSB) is located at the starting memory address.

**The Sign Bit:** The leftmost bit, or the bit at the very start, is the sign bit. If this bit is set to 1, it indicates that the number is negative, whereas if it's set to 0, the number is positive. It's worth noting that zero is considered a positive number in this context.

**The Significand (Mantissa):** The significand, also known as the mantissa, constitutes the part of the number that holds the value. It contains both the integer and fractional components of the number. In this representation, the significand consists of 23 bits for the fractional part.

**The Exponent:** In the floating-point number expressed as  $m * b^e$ , 'm' represents the significand (or mantissa), 'b' stands for the base, and 'e' represents the exponent. The exponent part plays a vital role in determining the scale or magnitude of the number.

In essence, the significand (mantissa) represents the decimal digits on both sides of the decimal point, and this can be seen as an extension of the concept of weighted positional notation used in various numbering systems like binary, decimal, and hexadecimal.

Just like the integers to the left of the decimal point have positive powers of 10, the numbers to the right side of the decimal point have negative powers of 10.

To illustrate this with a decimal number, consider 123.154. It can be represented as a sum, with each digit's position contributing based on its exponent:

$$123.154 = (1 * 10^2) + (2 * 10^1) + (3 * 10^0) + (1 * 10^{-1}) + (5 * 10^{-2}) + (4 * 10^{-3})$$

This concept applies similarly to the binary floating-point format, where the significand holds the binary digits to the left and right of the binary point (analogous to the decimal point).

It allows representation of a wide range of real numbers, from very small to very large, making it an essential part of computer systems and scientific calculations.

In summary, the IEEE single-precision floating-point format, as used in x86 processors, provides a standardized way to represent real numbers using a combination of a sign bit, significand, and exponent.

This format allows for accurate and flexible representation of numbers across a wide range of magnitudes.

## **Double precision format and Double Extended**

**Double precision** is a floating-point number format that uses **64 bits** to represent a number. This allows for a wider range of values and greater precision than single precision, which uses 32 bits.

It's the most common floating-point format used in scientific computing and other applications where high precision is required.

**Double extended precision** is a floating-point format that uses **80 bits** to represent a number. This provides even greater range and precision than double precision.

provides even greater range and precision than double precision.

However, double extended precision is not as widely supported as double precision, so it is only used in applications where the extra precision is essential.

Here is a table comparing double precision and double extended precision:

Feature	Double precision	Double extended precision
Number of bits	64	80
Sign bit	1	1
Exponent bits	11	16
Significand bits	52	63
Approximate normalized range	$2^{-1022}$ to $2^{1023}$	$2^{-16382}$ to $2^{16383}$

The single-precision floating-point format is a 32-bit format that consists of three parts:

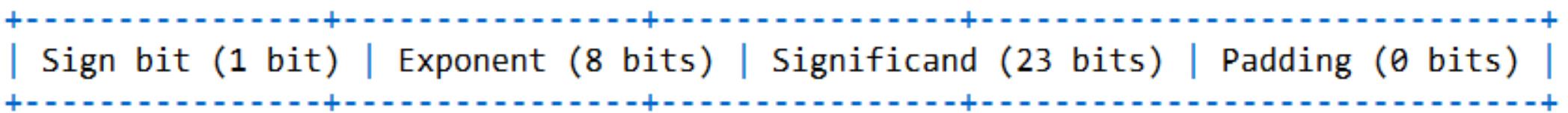
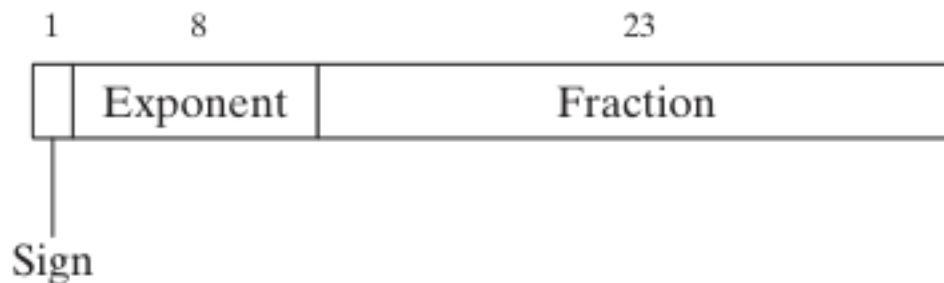
**Sign bit:** A single bit that indicates whether the number is positive or negative.

**Exponent:** An 8-bit field that represents the power of 2 by which the significand is multiplied.

**Significand:** A 23-bit field that represents the fractional part of the number.

The **sign bit** is the most significant bit of the floating-point number. The exponent is stored next, followed by the significand.

The following diagram shows the layout of the single-precision floating-point format:



The padding bits are zeros that are not used in the floating-point representation.

Value	Approximate range	Explanation
Normalized	$2^{-126}$ to $2^{127}$	Range for typical non-zero values.
Denormalized	$2^{-149}$ to $2^{-126}$	Range for very small values close to 0.
Special values	NaN, +/-Infinity	Represents Not-a-Number and Infinity.

**Normalized numbers** are the most common type of floating-point number. They have a significand that is between 1 and 2 (not including 2).

**Denormalized numbers** are numbers that have a significand that is less than 1. Denormalized values are used to represent very small numbers close to zero.

Denormalized	Normalized
1110.1	$1.1101 \times 2^3$
.000101	$1.01 \times 2^{-4}$
1010001.	$1.010001 \times 2^6$

**Special values** are used to represent NaN (Not a Number) and infinity. **NaN (Not-a-Number)** represents undefined or unrepresentable values. **+/-Infinity** represents positive and negative infinity, often used to represent overflow and underflow situations in calculations.

The single-precision floating-point format is a widely used format for representing floating-point numbers. It is used in many different types of applications, including scientific computing,

graphics, and gaming.

## Sign

The sign bit of a floating-point number indicates whether the number is positive or negative. A sign bit of 1 means the number is negative, and a sign bit of 0 means the number is positive. Zero is considered positive.

## Significand

The significand of a floating-point number is the fractional part of the number. It is represented by a sequence of digits to the left and right of the decimal point.

The digits to the left of the decimal point have positive exponents, and the digits to the right of the decimal point have negative exponents.

For example, the decimal floating-point number 123.154 can be represented as follows:

$$123.154 = (1 * 10^2) + (2 * 10^1) + (3 * 10^0) + (1 * 10^{-1}) + (5 * 10^{-2}) + (4 * 10^{-3})$$

The significand of this number is 123.154.

In binary floating-point numbers, the significand is represented by a sequence of 1s and 0s to the right of the binary point.

The first bit of the significand is always 1, and the remaining bits represent the fractional part of the number.

For example, the binary floating-point number 11.1011 can be represented as follows:

$$11.1011 = (1 * 2^1) + (1 * 2^0) + (1 * 2^{-1}) + (0 * 2^{-2}) + (1 * 2^{-3}) + (1 * 2^{-4})$$

The significand of this number is 1.1011.

Floating-point numbers can represent a very wide range of values, from very small numbers to very large numbers. This is because the significand and exponent can be adjusted to represent different values.

For example, the following floating-point numbers represent the same value:

24 1.23456789e+10

25 1.23456789e9

26 1.23456789e8

The significand is the same in all three numbers, but the exponent is different. The exponent indicates how many times the significand is multiplied by 2.

In the first number, the significand is multiplied by 2 ten times. In the second number, the significand is multiplied by 2 nine times.

And in the third number, the significand is multiplied by 2 eight times.

---

The precision of the significand is the number of bits that are used to represent the fractional part of the number.

In the IEEE double-precision format, the significand has 53 bits. This means that the significand can represent up to 53 bits of precision.

However, the entire continuum of real numbers cannot be represented in any floating-point format having a finite number of bits.

This is because there are an infinite number of real numbers, but only a finite number of bits can be used to represent a number.

For example, the binary value 1.11111 requires a 54-bit significand to be represented exactly. However, the IEEE double-precision format only has a 53-bit significand.

This means that the number 1.11111 cannot be represented exactly in the IEEE double-precision format.

## Exponents

The exponent in a floating-point number indicates how many times the significand is multiplied by 2. For example, the binary value  $1.1011 * 2^5$  means that the significand, 1.1011, is multiplied by 2 five times.

The biased exponent is always positive, between 1 and 254. The actual exponent range is from -126 to 127. The range was chosen so the smallest possible exponent's reciprocal cannot cause an overflow.

Exponent (E)	Biased (E + 127)	Binary
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

Single precision exponents are stored as 8-bit unsigned integers with a **bias of 127**. This means that the number's actual exponent must be added to 127 to get the biased exponent.

For example, the binary value  $1.1011 * 2^5$  has an actual exponent of 5. The biased exponent is therefore  $127 + 5 = 132$ .

## Translating Binary Floating-Point to Fractions

The table you sent shows examples of how to translate binary floating-point numbers to base-10 fractions.

The first column of the table shows the binary floating-point number. The second column shows the base-10 fraction. The third column shows the decimal value of the base-10 fraction.

Binary Floating-Point	Base-10 Fraction
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.00000000000000000000000001	1/8388608

Here is an explanation of each row in the table:

#### Row 1:

Binary floating-point number: 11.11

Base-10 fraction: 3 3/4

Decimal value: 3.75

#### Row 2:

Binary floating-point number: 101.0011

Base-10 fraction: 5 3/16

Decimal value: 5.1875

**Row 3:**

Binary floating-point number: 1101.100101

Base-10 fraction: 13 37/64

Decimal value: 13.58125

**Row 4:**

Binary floating-point number: 0.00101

Base-10 fraction: 5/32

Decimal value: 0.15625

**Row 5:**

Binary floating-point number: 1.011

Base-10 fraction: 1 3/8

Decimal value: 1.375

**Row 6:**

Binary floating-point number: 0.0000000000000000000000000001

Base-10 fraction: 1/8388608

Decimal value: 0.0000011920928955078125

*To translate a binary floating-point number to a base-10 fraction, we can use the following steps:*

- Convert the significand to a decimal number.
- Multiply the significand by the base 2 raised to the power of the exponent.

- If the sign bit is 1, the result is negative. Otherwise, the result is positive.

*For example, to translate the binary floating-point number 11.11 to a base-10 fraction, we would follow these steps:*

- Convert the significand 1.11 to a decimal number.  $1.11 = 1 + 1/2 + 1/4 = 7/4$
- Multiply the significand by the base 2 raised to the power of the exponent.  $7/4 * 2^1 = 7/2$
- If the sign bit is 1, the result is negative. Otherwise, the result is positive. The sign bit is 0, so the result is positive.
- Therefore, the base-10 fraction of the binary floating-point number 11.11 is 7/2.

To calculate the value of a floating-point number, we simply multiply the significand by the base 2 raised to the power of the exponent. For example, the following floating-point number:

$$1.1011 * 2^3$$

The exponent in the floating-point number  $1.1011 * 2^3$  is 3. This means that the significand, 1.1011, is multiplied by 2 three times.

The exponent in a floating-point number indicates how many times the significand is multiplied by the base of the number system. In this case, the base is 2, because the number is in binary.

The exponent in this floating-point number was chosen so that the value of the number would be between 1 and 2.

If the exponent had been smaller, the value of the number would have been less than 1. If the exponent had been larger, the value of the number would have been greater than 2.

Floating-point numbers are used to represent a wide range of values, from very small numbers to very large numbers. The exponent allows us to do this by scaling the significand up or down.

Let's discuss another table:

Binary	Decimal Fraction	Decimal Value
.1	$1/2$	.5
.01	$1/4$	.25
.001	$1/8$	.125
.0001	$1/16$	.0625
.00001	$1/32$	.03125

To convert from binary to decimal, we multiply each digit of the binary fraction by its corresponding power of two, and then add the results together.

For example, to convert the binary fraction 0.1011 to decimal, we would do the following:

$$0.1011 * 2^{-3} + 0.1011 * 2^{-2} + 0.1011 * 2^{-1} + 0.1011 * 2^0$$

= 0.125 + 0.25 + 0.5 + 1.1011

= 2.0761

To convert from decimal to binary, we repeatedly divide the decimal number by two, and write down the remainder at each step.

The remainders, in reverse order, give the binary representation of the number. For example, to convert the decimal number 2.0761 to binary, we would do the following:

29	2.0761 / 2 = 1.0381	(remainder of 1)
30	1.0381 / 2 = 0.51905	(remainder of 1)
31	0.51905 / 2 = 0.259525	(remainder of 1)
32	0.259525 / 2 = 0.1297625	(remainder of 1)
33	0.1297625 / 2 = 0.06488125	(remainder of 1)
34	0.06488125 / 2 = 0.032440625	(remainder of 0)

The binary representation of 2.0761 is therefore 1.0111.

Floating-point numbers can represent a much wider range of values by using more bits.

## ***Creating the IEEE Representation***

The passage below describes how to create the IEEE representation of a floating-point number.

The first step is to normalize the sign bit, exponent, and significand fields. This means that the significand must be adjusted so that the leading bit is 1.

For example, the significand 1.1011 is normalized by shifting the bits one position to the left, giving the normalized significand 11.011.

The next step is to encode the sign bit, exponent, and significand fields.

The **sign bit** is encoded as a single bit, with 0 representing a positive number and 1 representing a negative number.

The **exponent** is encoded as an 8-bit unsigned integer with a bias of 127. This means that the number's actual exponent must be added to 127 to get the biased exponent.

For example, the actual **exponent 5** is **encoded as the biased exponent 132**.

The **significand** is encoded as a 23-bit unsigned integer. The leading bit of the normalized significand is not explicitly encoded, since it is always 1.

This means that the significand 11.011 is encoded as the 23-bit integer 01000100000000000000000.

Once the sign bit, exponent, and significand fields have been encoded, they can be combined to form the complete binary IEEE short real.

The following table shows an example of how to create the IEEE representation of the floating-point number  $1.101 * 2^{20}$ :

Field	Value
Sign bit	0
Exponent	132
Significand	010001000000000000000000
IEEE representation	01111110100010000000000000000000

The IEEE representation of a floating-point number can be used to store and manipulate the number in a computer.

IEEE floating-point numbers are widely used in a variety of applications, including scientific computing, graphics, and gaming.

The table you sent shows the binary value, exponent, sign bit, and significand of several different IEEE floating-point numbers. Here is a concise explanation of each row:

Binary Value	Biased Exponent	Sign, Exponent, Fraction		
-1.11	127	1	01111111	11000000000000000000000000000000
+1101.101	130	0	10000010	10110100000000000000000000000000
-.00101	124	1	01111100	01000000000000000000000000000000
+100111.0	132	0	10000100	00111000000000000000000000000000
+.0000001101011	120	0	01111000	10101100000000000000000000000000

Binary value	Exponent	Sign bit	Significand
-1.11	127	1	1101.101
130	0	0	1.10111

The first row shows the IEEE representation of the floating-point number -1.11. The sign bit is 1, which indicates that the number is negative. The exponent is 127, which is the largest possible exponent for a normalized floating-point number. The significand is 1101.101, which is the binary representation of the fraction 3/4.

The second row shows the IEEE representation of the floating-point number 1.10111. The sign bit is 0, which indicates that the number is positive. The exponent is 0, which means that the significand

is not multiplied by 2. The significand is 1.10111, which is the binary representation of the fraction 23/16.

The other rows in the table show the IEEE representations of other floating-point numbers. The last row in the table shows the smallest possible normalized floating-point number.

---

The IEEE specification includes several real-number and non-number encodings, including:

***Positive and negative zero:***

These are two different representations of the number zero.

***Denormalized finite numbers:***

These are numbers that are very close to zero. They are represented using a special format that allows them to be represented with a wider range of exponents.

***Normalized finite numbers:***

These are all the other nonzero finite numbers that can be represented in floating-point format. They are represented using a format that allows them to be represented with a high degree of accuracy.

***Positive and negative infinity:***

These are representations of the concepts of positive and negative infinity.

***Non-numeric values (NaN, known as Not a Number):***

These are used to represent results of invalid floating-point operations.

***Indefinite numbers:***

These are used by the floating-point unit (FPU) as responses to some invalid floating-point operations.

## Normalized and denormalized numbers

**Normalized finite numbers** are all the nonzero finite values that can be encoded in a normalized real number between zero and infinity. Normalized numbers have a leading bit of 1 in the significand. This allows for the greatest possible precision in representing the number.

**Denormalized finite numbers** are numbers that are very close to zero. They are represented using a special format that allows them to be represented with a wider range of exponents. Denormalized numbers have a leading bit of 0 in the significand. This means that some precision is lost in representing the number, but it allows for a wider range of numbers to be represented.

In the example you provided, the number:

**1.010111000000000001111 x 2<sup>-129</sup>**

Is too small to be represented as a normalized number.

The FPU therefore denormalizes the number by shifting the binary point left 1 bit at a time until the exponent reaches a valid range.

**1.010111000000000001111 x 2<sup>-129</sup> -> 0.1010111000000000000111 x 2<sup>-128</sup>**

**0.1010111000000000000111 x 2<sup>-128</sup> -> 0.0101011100000000000011 x 2<sup>-127</sup>**

**0.010101110000000000011 x 2<sup>-127</sup> -> 0.001010111000000000001 x 2<sup>-126</sup>**

After three denormalization steps, the exponent is now within the valid range. The denormalized number is now represented as **0.001010111000000000001 x 2<sup>-126</sup>**.

This results in some loss of precision in the significand, but it allows the number to be represented.

**Denormalized numbers** are important because they allow the FPU to represent a wider range of numbers than it would otherwise be able to. This is important for many applications, such as scientific computing and graphics.

So, to finish off:

### Converting IEEE single-precision values to decimal

The following are the steps to convert an IEEE single-precision value to decimal:

- Check the most significant bit (MSB) to determine the sign of the number. If the MSB is 1, the number is negative. If the MSB is 0, the number is positive.
- Extract the next 8 bits to get the exponent. Subtract 127 from the exponent to get the unbiased exponent. Convert the unbiased exponent to decimal.
- Extract the next 23 bits to get the significand. Add a leading 1 to the significand bits. Ignore any trailing zeros.
- Create a floating-point binary number using the significand, the sign determined in step 1, and the exponent calculated in step 2.
- Denormalize the binary number produced in step 3. To do this, shift the binary point the number of places equal to the value of the exponent. If the exponent is positive, shift the binary point to the right. If the exponent is negative, shift the binary point to the left.
- Convert the denormalized binary number to decimal using weighted positional notation.

Example:

Convert the following IEEE single-precision value to decimal:

**0 10000010 010110000000000000000000**

- The MSB is 0, so the number is positive.
- The unbiased exponent is 00000011, which is equal to decimal 3.
- The significand is 010110000000000000000000, which is equal to 1.01011 binary.
- The floating-point binary number is 1.01011 \* 2^3.
- The denormalized binary number is 1010.11.
- The decimal value is 10 3/4, or 10.75.

## ***Positive and Negative Infinity***

Positive and negative infinity are special floating-point values that represent the maximum positive and negative real numbers, respectively. They are used to represent results of operations that would otherwise overflow, such as dividing by zero or adding a very large number to a very small number.

Nans are bit patterns that do not represent any valid real number. They are used to represent results of invalid floating-point operations, such as dividing zero by zero or taking the square root of a negative number.

The following table shows the specific encodings for positive infinity, negative infinity, quiet NaNs, and signaling NaNs in the IEEE floating-point standard:

Type	Sign bit	Exponent	Significand
Positive infinity	0	11111111	00000000
Negative infinity	1	11111111	00000000
Quiet NaN	x	11111111	xxxxxx...
Signaling NaN	x	11111110	xxxxxx...

Note that the significand field for NaNs can be any value.

Floating-point units (FPUs) handle positive infinity, negative infinity, and NaNs in special ways.

For example, if the FPU attempts to perform an operation that would result in positive infinity or negative infinity, it will instead return the corresponding infinity value.

If the FPU attempts to perform an operation that would result in a NaN, it will instead return a NaN value and may also generate an exception.

The specific behavior of the FPU when handling positive infinity, negative infinity, and NaNs is defined by the IEEE floating-point standard.

This ensures that all floating-point units behave in a consistent manner, regardless of the manufacturer or architecture.

Table 12-6 Specific Single-Precision Encodings.

Value	Sign, Exponent, Significand		
Positive zero	0	00000000	00000000000000000000000000000000
Negative zero	1	00000000	00000000000000000000000000000000
Positive infinity	0	11111111	00000000000000000000000000000000
Negative infinity	1	11111111	00000000000000000000000000000000
QNaN	x	11111111	1xxxxxxxxxxxxxxxxxxxxxx
SNaN	x	11111111	0xxxxxxxxxxxxxxxxxxxxxx <sup>a</sup>

<sup>a</sup> SNaN significand field begins with 0, but at least one of the remaining bits must be 1.

## *Converting Decimal Fractions to Binary Reals*

### Method 1

- Write the decimal fraction as a sum of fractions in the form  $(1/2 + 1/4 + 1/8 + \dots)$ .
- Translate each fraction in the sum to binary.
- Add the binary fractions together to get the final result.

## Method 2

- Convert the numerator and denominator of the decimal fraction to binary.
- Perform long division on the binary numbers.
- The quotient of the division is the binary representation of the decimal fraction.

### Example

Let's use the second method to convert the decimal fraction 0.5 to binary.

- Convert the numerator and denominator of the decimal fraction to binary.
- 5 in decimal is 101 in binary.
- 10 in decimal is 1010 in binary.
- Perform long division on the binary numbers:

$$\begin{array}{r} .1 \\ 1010 \overline{)0101.0} \\ -1010 \\ \hline 0 \end{array}$$

When 1010 binary is subtracted from the dividend the remainder is zero, and the division stops.

Therefore, the decimal fraction 5/10 equals 0.1 binary. We will call this approach the binary long division method.

## Representing 0.2 in Binary

Let's convert decimal 0.2 (2/10) to binary using the binary long division method. First, we divide binary 10 by binary 1010 (decimal 10).

The first quotient large enough to use is 10000. After dividing 1010 into 10000, the remainder is 110.

Appending another zero, the new dividend is 1100. After dividing 1010 into 1100, the remainder is 10. After appending three zeros, the new dividend is 10000.

This is the same dividend we started with. From this point on, the sequence of the bits in the quotient repeats (0011. . .), so we know that an exact quotient will not be found and 0.2 cannot be represented by a finite number of bits.

The single-precision encoded significand is **00110011001100110011001**.

$$\begin{array}{r} .00110011 \text{ (etc.)} \\ \hline 1010 \left| \begin{array}{r} 10.0000000 \\ 1010 \\ \hline 1100 \\ 1010 \\ \hline 1000 \\ 1010 \\ \hline 1100 \\ 1010 \\ \hline \text{etc.} \end{array} \right. \end{array}$$

=====

Examples of decimals factored as binary reals:

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

The table above shows how to convert decimal fractions to binary reals using the first method I described above.

The first column of the table shows the decimal fraction. The second column shows the factored form of the decimal fraction.

The third column shows the binary representation of the decimal fraction.

For example, the first row of the table shows the decimal fraction 1/2. The factored form of 1/2 is 1/2. The binary representation of 1/2 is .1.

The second row of the table shows the decimal fraction  $1/4$ . The factored form of  $1/4$  is  $1/4$ . The binary representation of  $1/4$  is  $.01$ .

The third row of the table shows the decimal fraction  $3/4$ . The factored form of  $3/4$  is  $1/2 + 1/4$ . The binary representation of  $3/4$  is  $.11$ .

The rest of the rows in the table follow the same pattern.

To use the table to convert a decimal fraction to binary, simply find the decimal fraction in the first column of the table. The binary representation of the decimal fraction will be in the third column.

For example, to convert the decimal fraction  $5/8$  to binary, find the decimal fraction  $5/8$  in the first column of the table. The binary representation of  $5/8$  is  $.101$ .

It is important to note that not all decimal fractions can be represented exactly in binary. This is because there is an infinite number of real numbers, but only a finite number of binary digits. As a result, some decimal fractions must be approximated when they are converted to binary.

=====

The **factored column** in the table shows how the decimal fraction can be expressed as a sum of fractions in the form  $(1/2 + 1/4 + 1/8 + \dots)$ . This is useful because each of the fractions in the sum can be easily converted to binary.

*To factor a decimal fraction, you can use the following steps:*

- Find the largest power of two that is less than or equal to the decimal fraction.

- Subtract that power of two from the decimal fraction.
- Repeat steps 1 and 2 until the decimal fraction is zero.
- The factors of the decimal fraction are the powers of two that were subtracted in each step.

*For example, to factor the decimal fraction 3/4, we would do the following:*

- The largest power of two that is less than or equal to 3/4 is 1/2.
- Subtracting 1/2 from 3/4 gives us 1/4.
- The largest power of two that is less than or equal to 1/4 is 1/4.
- Subtracting 1/4 from 1/4 gives us zero.
- Therefore, the factors of 3/4 are 1/2 and 1/4.

Once you have factored the decimal fraction, you can convert each of the factors to binary. To do this, **simply divide the numerator of the factor by the denominator. The quotient is the binary representation of the factor.**

For example, to convert the factor 1/2 to binary, we would divide the numerator 1 by the denominator 2. The quotient is 0.5, which is the binary representation of 1/2.

To convert the factor 1/4 to binary, we would divide the numerator 1 by the denominator 4. The quotient is 0.25, which is the binary representation of 1/4.

Once you have converted each of the factors to binary, you can add them together to get the binary representation of the decimal fraction.

For example, the binary representation of 3/4 is 0.11, which is the sum of the binary representations of 1/2 and 1/4.

## **Questions**

### **Why doesn't the single-precision real format permit an exponent of -127?**

The single-precision real format does not permit an exponent of -127 because it would generate a zero. This is because the unbiased exponent is calculated by subtracting 127 from the exponent. If the exponent is -127, then the unbiased exponent would be zero. However, the significand of a non-zero floating-point number cannot be zero. Therefore, an exponent of -127 is not permitted.

### **Why doesn't the single-precision real format permit an exponent of 128?**

The single-precision real format does not permit an exponent of 128 because it would generate an overflow. This is because the unbiased exponent is calculated by subtracting 127 from the exponent. If the exponent is 128, then the unbiased exponent would be 1. However, the unbiased exponent cannot be greater than 255. Therefore, an exponent of 128 is not permitted.

### **In the IEEE double-precision format, how many bits are reserved for the fractional part of the significand?**

The IEEE double-precision format reserves 52 bits for the fractional part of the significand. This gives double-precision floating-point numbers a much higher degree of precision than single-precision floating-point numbers.

### **In the IEEE single-precision format, how many bits are reserved for the exponent?**

The IEEE single-precision format reserves 8 bits for the exponent. This gives single-precision floating-point numbers a range of values from  $2^{-126}$  to  $2^{127}$ .

## **Floating-Point Unit**

The **FPU** is a specialized hardware unit that performs floating-point calculations. It has its own set of registers called a **register stack**.

The FPU loads values from memory into the register stack, performs calculations, and stores stack values into memory.

The FPU evaluates mathematical expressions in **postfix format**.

A **postfix expression** is an expression where the operands appear before the operators. For example, the postfix expression for the **infix expression**  $(5 * 6) - 4$  is  $5 \ 6 \ * \ 4 \ -$ .

Left to Right	Stack	Action		
5	<table border="1"><tr><td>5</td></tr></table>	5	ST (0) push 5	
5				
5 6	<table border="1"><tr><td>5</td></tr><tr><td>6</td></tr></table>	5	6	ST (1) push 6 ST (0)
5				
6				
5 6 *	<table border="1"><tr><td>30</td></tr></table>	30	Multiply ST(1) by ST(0) and pop ST(0) off the stack.	
30				
5 6 * 4	<table border="1"><tr><td>30</td></tr><tr><td>4</td></tr></table>	30	4	ST (1) push 4 ST (0)
30				
4				
5 6 * 4 -	<table border="1"><tr><td>26</td></tr></table>	26	Subtract ST(0) from ST(1) and pop ST(0) off the stack.	
26				

The FPU uses an expression stack to hold intermediate values during the evaluation of postfix expressions. The expression stack is a **last-in-first-out (LIFO) stack**.

This means that the last value pushed onto the stack is the first value popped off the stack.

To evaluate a postfix expression, the FPU pushes the operands onto the expression stack.

Then, it pops the top two operands off the stack and performs the operation specified by the next operator in the expression. The result of the operation is pushed onto the stack.

This process continues until all of the operators in the expression have been processed.

The following table shows some examples of equivalent infix and postfix expressions:

Infix expression	Postfix expression
$(5 * 6) - 4$	$5 6 * 4 -$
$(A + B) * C$	$A B + C *$
$(A - B) / C$	$A B - C /$
$A + B$	$A B +$
$(A - B) / D$	$A B - D /$
$(A + B) * (C + D)$	$A B + C D + *$
$((A + B) / C) * (E - F)$	$A B + C / E F - *$

Note that the parentheses in the infix expressions are not necessary in the postfix expressions, because the order of operations is implied by the order of the operands.

For example, in the expression  $(A + B) * (C + D)$ , the multiplication operation is performed before the addition operation, even though there are no parentheses. This is because multiplication has a higher precedence than addition.

Here is an example of how the FPU would evaluate the postfix expression  $A B +$  using its expression stack:

- Expression stack: empty
- Next operand: A
- Push A onto the expression stack
- Expression stack: A
- Next operand: B
- Push B onto the expression stack
- Expression stack: A, B
- Next operator: +
- Pop the top two operands off the expression stack and perform the addition operation
- Push the result of the addition operation onto the expression stack
- Expression stack: A + B

The FPU would then stop evaluating the expression, because there are no more operands or operators. The result of the expression is  $A + B$ .

The FPU evaluates all postfix expressions in the same way.

It pushes the operands onto the expression stack, pops the top two operands off the stack and performs the operation specified by the next operator, and pushes the result of the operation onto the stack.

This process continues until all of the operators in the expression have been processed. The final value on the expression stack is the result of the expression.

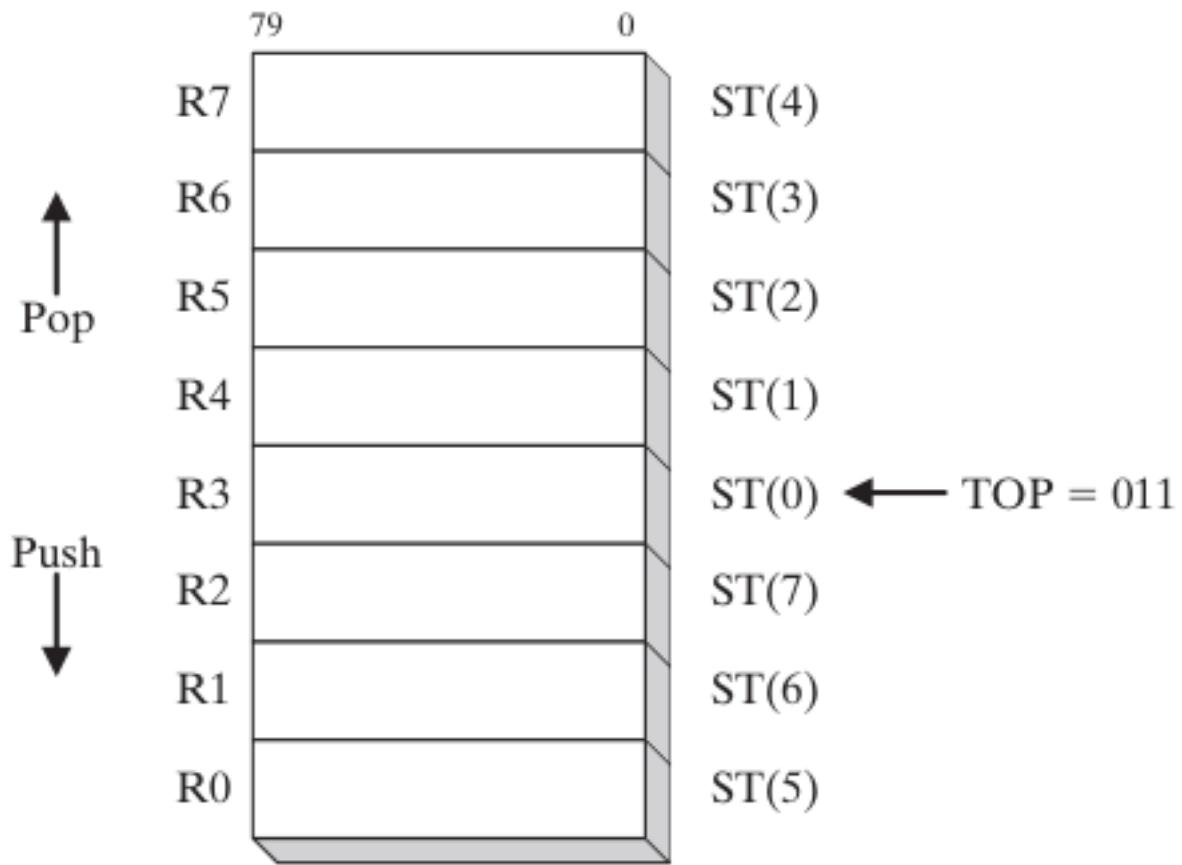
## FPU DATA REGISTERS

The FPU has eight individually addressable 80-bit data registers named R0 through R7. Together, they are called a register stack.

The FPU stack works in a **last-in-first-out (LIFO)** manner. This means that the last value pushed onto the stack is the first value popped off the stack.

The top of the FPU stack is indicated by a three-bit field named TOP in the FPU status word. The register at the top of the stack is also known as ST(0).

The rest of the registers are known as ST(1), ST(2), ..., ST(7), in order.



To push a value onto the FPU stack, the FPU decrements TOP by 1 and copies the value into the register identified as ST(0). If TOP equals 0 before a push, TOP wraps around to register R7.

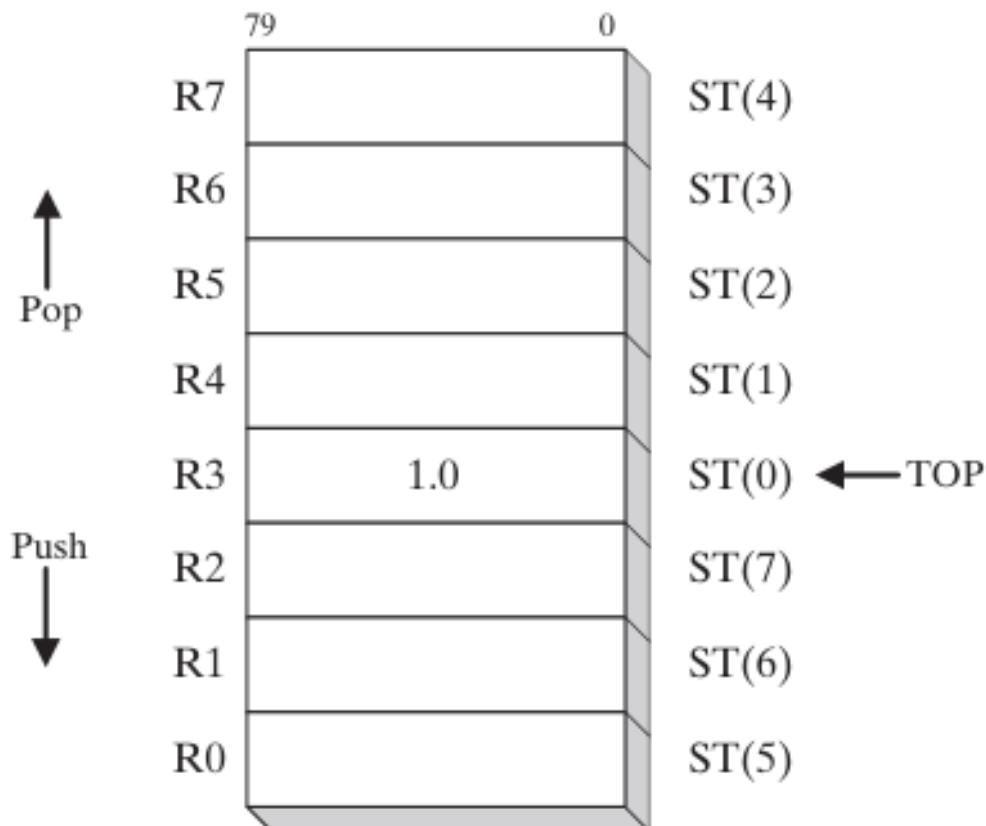
To pop a value off the FPU stack, the FPU copies the data at ST(0) into an operand, then adds 1 to TOP. If TOP equals 7 before the pop, it wraps around to register R0.

If loading a value into the stack would result in overwriting existing data in the register stack, a floating-point exception is generated.

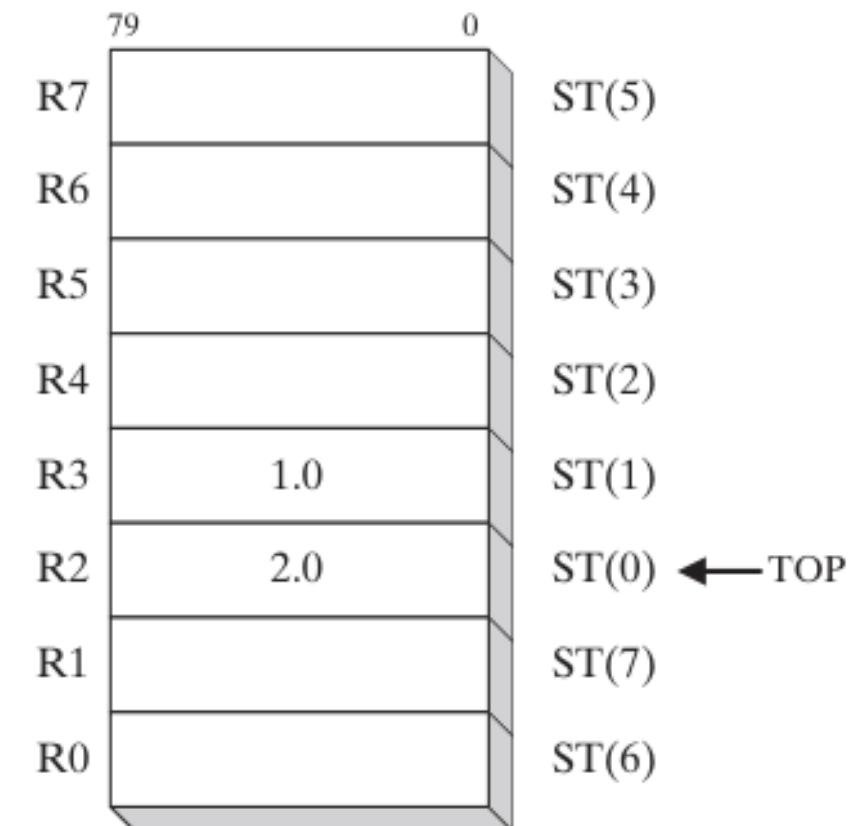
The following diagram shows the FPU stack after 1.0 and 2.0 have been pushed onto the stack:

FPU stack after pushing 1.0 and 2.0.

After pushing 1.0



After pushing 2.0



To perform a floating-point operation, the FPU pops the required operands off the stack, performs the operation, and pushes the result back onto the stack.

For example, to perform the addition operation  $1.0 + 2.0$ , the FPU would pop the operands 1.0 and 2.0

off the stack, add them together, and push the result, 3.0, back onto the stack.

The FPU stack is a powerful tool for performing floating-point calculations. It allows the FPU to efficiently perform complex operations without having to store intermediate results in memory.

The functionality of the Floating-Point Unit (FPU) in a processor, which is responsible for handling floating-point arithmetic operations. Let's break down some of the key points:

### ***ST(n) Notation:***

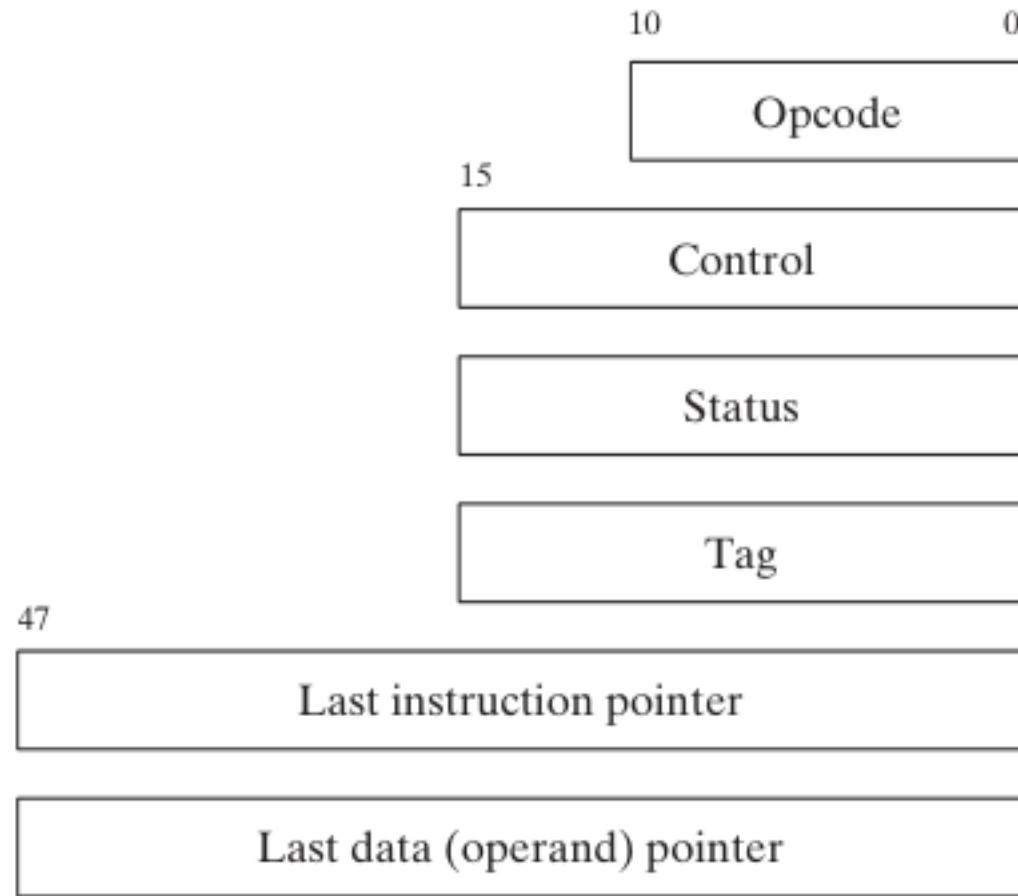
The FPU uses a notation like ST(0), ST(1), and so on to refer to stack registers. ST(0) always represents the top of the stack. This notation is used to address data within the FPU's stack-based architecture.

### ***Floating-Point Value Format:***

The floating-point values within FPU registers are stored in the IEEE 10-byte extended real format. This format is sometimes referred to as temporary real.

When the FPU stores the result of an operation in memory, it may translate it into various formats, including integers, long integers, single precision, double precision, or packed binary-coded decimal (BCD).

## FPU special-purpose registers.



**Special-Purpose Registers:** The FPU has several special-purpose registers:

**Opcode Register:** Stores the opcode of the last non-control instruction executed.

**Control Register:** Controls the precision and rounding method used by the FPU during calculations. It can also be used to mask individual floating-point exceptions.

**Status Register:** Contains the top-of-stack pointer, condition codes, and information about exceptions or warnings.

**Tag Register:** Indicates the content of each register in the FPU data-register stack. It uses two bits per register to specify whether the register holds a valid number, zero, or a special value (NaN, infinity, denormal, unsupported format) or is empty.

**Last Instruction Pointer Register:** Stores a pointer to the last non-control instruction executed.

**Last Data (Operand) Pointer Register:** Stores a pointer to a data operand, if any, used by the last executed instruction.

## ROUNDING IN FPU

The concept of rounding in floating-point calculations and how the Floating-Point Unit (FPU) handles it. I'll break down the key points for better understanding:

**Purpose of Special-Purpose Registers:** The special-purpose registers mentioned are used by operating systems to preserve the state of the FPU when switching between tasks. This state preservation is crucial for multitasking, where the CPU switches between different tasks.

**Rounding in Floating-Point Calculations:** The FPU aims to produce precise results from floating-point calculations, but sometimes the destination operand (the format in which the result is stored) can't represent the exact calculated result. Rounding is used to adjust the result to fit within the destination format.

**Example of Rounding:** Let's take an example where the precise result is 1.0111, but the destination format can only represent three fractional bits.

Rounding can occur in two ways:

- (a) Round up to the next higher value by adding 0.0001: 1.0111 → 1.100
- (b) Round down to the closest value by subtracting 0.0001: 1.0111 → 1.011

Similar rounding can be applied for negative values:

- (a) -1.0111 → -1.100
- (b) -1.0111 → -1.011

**Rounding Methods:** The FPU provides four rounding methods:

- **Round to Nearest Even:** The result is rounded to the nearest value. If two values are equally close, the result is an even value (least significant bit = 0).
- **Round Down Toward  $-\infty$ :** The result is rounded to a value less than or equal to the precise result.
- **Round Up Toward  $+\infty$ :** The result is rounded to a value greater than or equal to the precise result.
- **Round Toward Zero (Truncation):** The absolute value of the rounded result is less than or equal to the precise result.

**FPU Control Word:** The FPU control word contains the RC (Rounding Control) field, which specifies the rounding method to use. The values are binary-encoded as follows:

- **00 binary:** Round to nearest even (default).
- **01 binary:** Round down toward negative infinity.
- **10 binary:** Round up toward positive infinity.

- 11 **binary**: Round toward zero (truncate).

The default method is "**Round to Nearest Even**", which is generally considered the most accurate and appropriate for most application programs.

The tables below would show how these rounding methods would be applied to the binary values 1.0111 and -1.0111, respectively.

Example: Rounding +1.0111.

Method	Precise Result	Rounded
Round to nearest even	1.0111	1.100
Round down toward $-\infty$	1.0111	1.011
Round toward $+\infty$	1.0111	1.100
Round toward zero	1.0111	1.011

Example: Rounding -1.0111.

Method	Precise Result	Rounded
Round to nearest (even)	-1.0111	-1.100
Round toward $-\infty$	-1.0111	-1.100
Round toward $+\infty$	-1.0111	-1.011
Round toward zero	-1.0111	-1.011

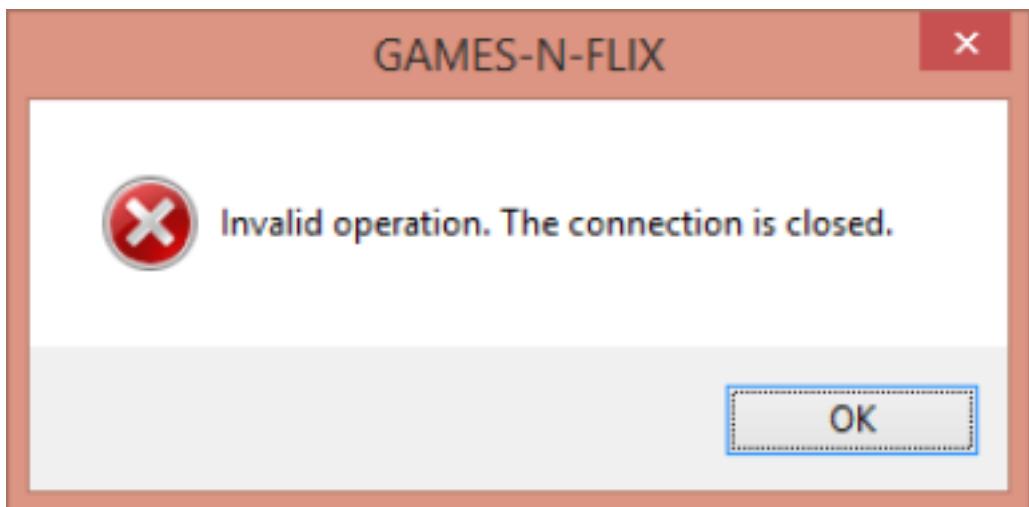
Rounding in floating-point arithmetic is essential to ensure that calculated results fit within the chosen format while minimizing error. The method chosen depends on the specific requirements of the application.

## **Floating-Point Exceptions**

The passage below, describes floating-point exceptions and how the FPU handles them.

Floating-point exceptions are errors that can occur during floating-point calculations. The FPU recognizes and detects six types of floating-point exceptions:

**Invalid operation (#I):** This exception is raised when an invalid operation is attempted, such as dividing zero by zero or taking the square root of a negative number.



**Divide by zero (#Z):** This exception is raised when an attempt is made to divide by zero.

Any number divided by **ZERO**



is **UNDEFINED**

**Denormalized operand (#D):** This exception is raised when an attempt is made to operate on a denormalized number. A denormalized number is a floating-point number that is very close to zero.

Denormalized

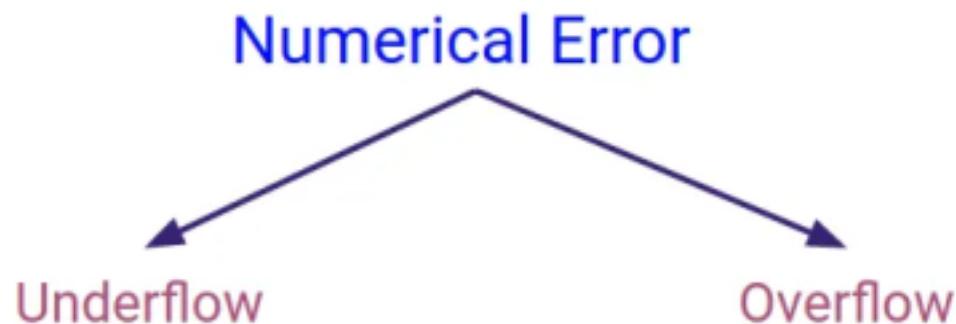
operand

**Numeric overflow (#O):** This exception is raised when the result of a floating-point operation is too large to be represented by the FPU.

## #2724 Unexpected error "arithmetic exception, numeric overflow, or...



**Numeric underflow (#U):** This exception is raised when the result of a floating-point operation is too small to be represented by the FPU.



**Inexact precision (#P):** This exception is raised when the result of a floating-point operation is not exact. This can happen because floating-point numbers are represented by a finite number of bits.



Each exception type has a corresponding flag bit and mask bit. When a floating-point exception is detected, the processor sets the matching flag bit.

The mask bit controls whether or not the processor automatically handles the exception. If the mask bit is set, the processor automatically handles the exception and lets the program continue. If the mask bit is clear, the processor invokes a software exception handler.

The processor's masked (automatic) responses are generally acceptable for most programs. However, custom exception handlers can be used in cases where specific responses are required by the application.

A single instruction can trigger multiple exceptions. The processor keeps an ongoing record of all exceptions occurring since the last time exceptions were cleared. After a sequence of calculations completes, you can check to see if any exceptions occurred.

*Here are some examples of how floating-point exceptions can occur:*

- Dividing a number by zero will raise a divide by zero exception.
- Taking the square root of a negative number will raise an invalid operation exception.
- Adding two very large numbers together may raise a numeric overflow exception.

- Subtracting two very small numbers from each other may raise a numeric underflow exception.
- Multiplying two numbers together may raise an inexact precision exception if the result is too large to be represented by the FPU.

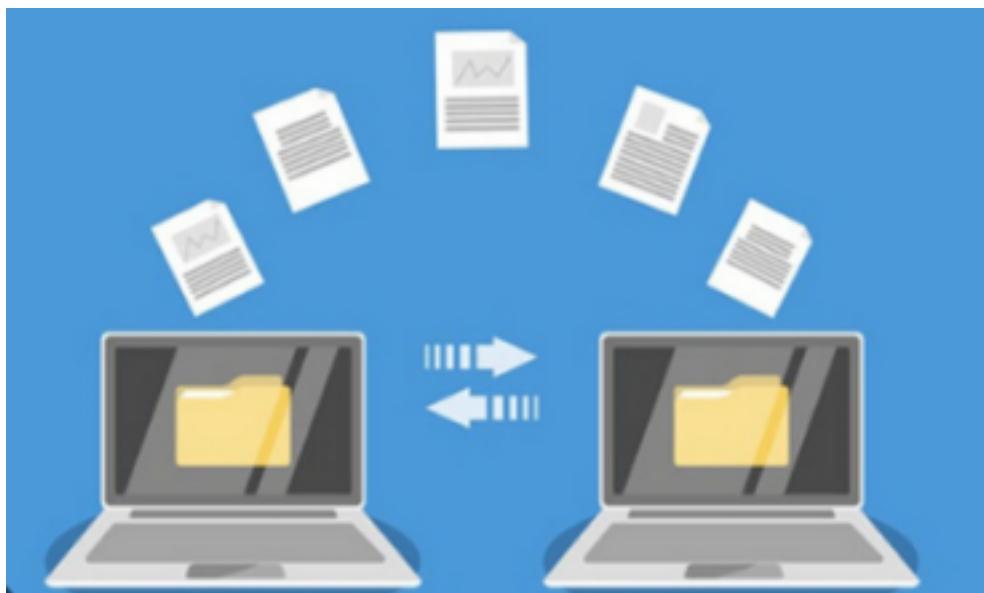
Floating-point exceptions can cause unexpected behavior in programs. It is important to be aware of the different types of floating-point exceptions and how to handle them.

You can use the FPU's mask bits to control how the processor handles floating-point exceptions. You can also write custom exception handlers to handle floating-point exceptions in a specific way.

## *Floating-Point Instruction Set*

The floating-point instruction set is a set of instructions that the FPU uses to perform floating-point calculations. The instruction set is divided into the following categories:

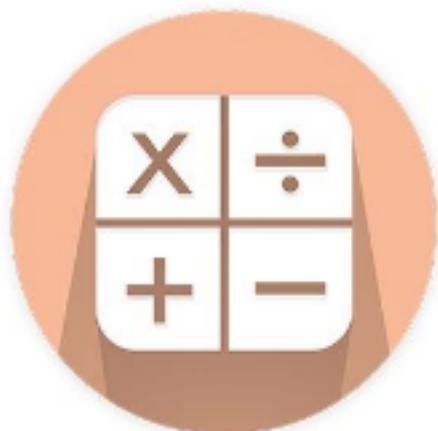
**Data transfer:** These instructions move data between the FPU registers and memory.



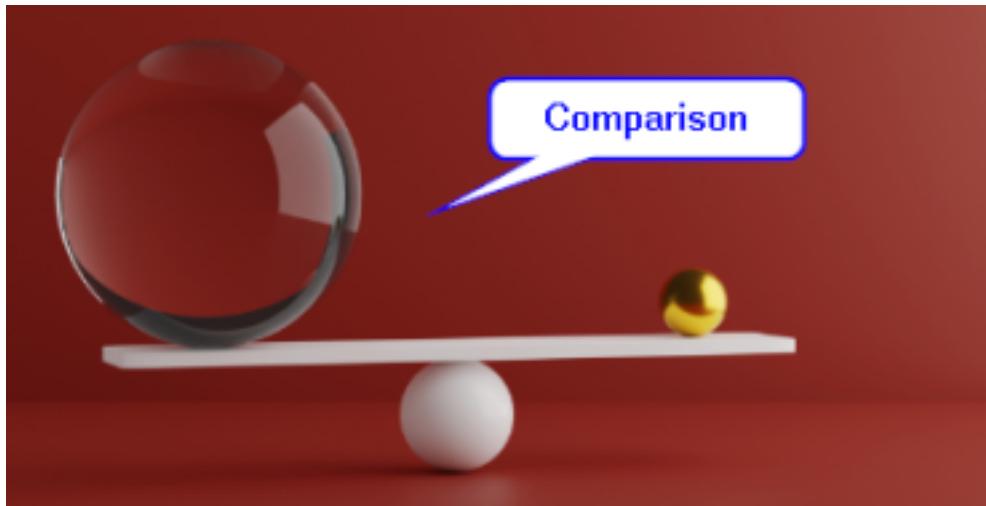
**Basic arithmetic:** These instructions perform basic floating-point operations such as addition, subtraction, multiplication, and division.

## Arithmetic

1 2 3  
4 5 6  
7 8 9  
0



**Comparison:** These instructions compare two floating-point numbers and return a result indicating whether one number is greater than, equal to, or less than the other number.



**Transcendental:** These instructions perform transcendental functions such as sine, cosine, and tangent.



**Load constants:** These instructions load predefined constants into the FPU registers.



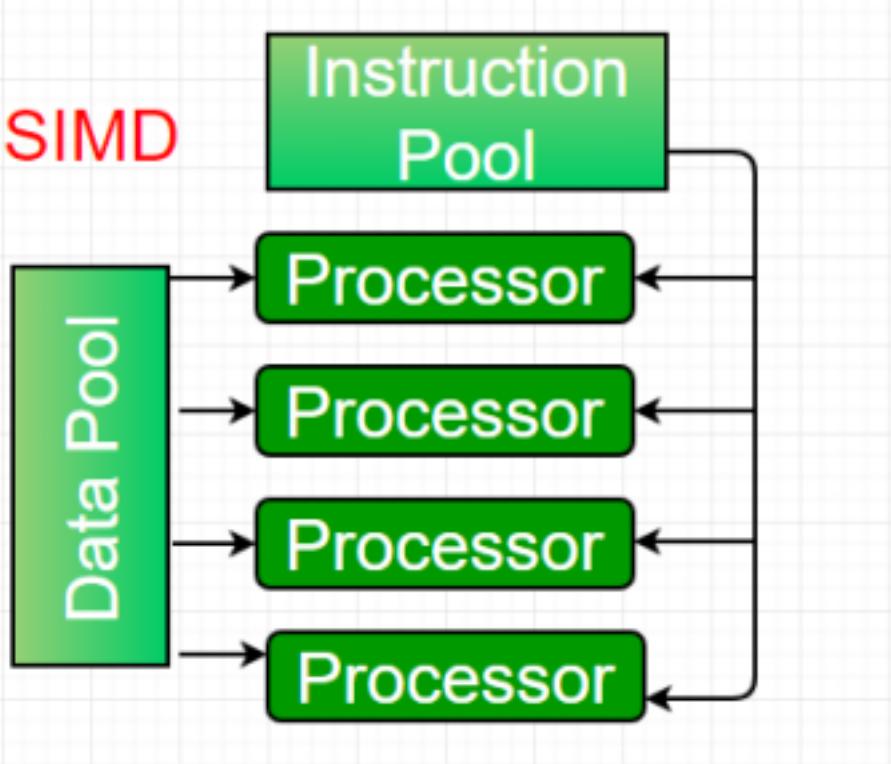
**x87 FPU control:** These instructions control the behavior of the FPU, such as the rounding mode and the calculation precision.

```
const int mask = 0x0c00

asm(
    "    fldcw          \n"
    "    fstcw          \n"
    "    finit          \n"
    "    fldl    %[mask] \n"
    "    //& together   \n"
    //set bits 10 & 11
);
```

**x87 FPU and SIMD state management:** These instructions manage the state of the FPU, such as saving and restoring the FPU register stack.

SIMD



Floating-point instruction names begin with the letter F.

The second letter of the instruction mnemonic indicates how a memory operand is to be interpreted: **B** indicates a **BCD operand**, and **I** indicates a **binary integer operand**. If neither is specified, the memory operand is assumed to be in real-number format.

Floating-point instructions can have **zero operands, one operand, or two operands**.

If there are two operands, one must be a floating-point register. There are no immediate operands, but certain predefined constants can be loaded into the stack.

**Integer operands** must be loaded into the FPU from memory before they can be used in floating-point

calculations. When storing floating-point values into integer memory operands, the values are automatically truncated or rounded into integers.

To initialize the FPU, you can use the **FINIT instruction**.

The FINIT instruction sets the FPU control word to **037Fh**, which masks all floating-point exceptions, sets rounding to nearest even, and sets the calculation precision to 64 bits.

It is recommended to **call FINIT at the beginning of your programs** so that you know the starting state of the FPU.

Here are some examples of floating-point instructions:

```
61 FLD [memory_address] ;Load a floating-point value from memory into the FPU stack.  
62 FADD ST(0), ST(1) ;Add the top two values on the FPU stack and store the result at the top of the stack.  
63 FSTP [memory_address] ;Store the top value on the FPU stack to memory.
```

## FLOATING POINT DATATYPES

MASM supports the following floating-point data types:

- **QWORD**: 64-bit integer
- **TBYTE**: 80-bit (10-byte) integer
- **REAL4**: 32-bit (4-byte) IEEE short real
- **REAL8**: 64-bit (8-byte) IEEE long real
- **REAL10**: 80-bit (10-byte) IEEE extended real

When defining memory operands for FPU instructions, you must use one of these data types.

For example, to load a floating-point variable into the FPU stack, the variable must be defined as REAL4, REAL8, or REAL10.

Here is an example of how to use the REAL8 data type to define a floating-point variable and load it into the FPU stack:

```
75 ; Define a double-precision floating-point variable named bigVal
76 .data
77 bigVal REAL8 1.212342342234234243E+864
78
79 ; Start of the code section
80 .code
81 ; Load the value of bigVal into the FPU stack
82 fld QWORD PTR [bigVal]
83
84 ; Here, QWORD PTR is used to indicate that we are loading a double-precision (64-bit) value from memory.
85 ; The brackets [] indicate that we are accessing the memory location pointed to by bigVal.
86
87 ; The value of bigVal is now loaded onto the FPU stack.
```

This code will load the value stored in bigVal onto the FPU stack, making it ready for further floating-point operations.

The fld instruction loads the value of the variable bigVal into the FPU stack. The bigVal variable is defined as REAL8, so the fld instruction knows how to interpret the data in memory.

The floating-point data types supported by MASM are used in a wide variety of applications,

including graphics, scientific computing, and financial modeling.

## **LOAD FLOATING POINT VALUES**

This passage describes the FLD and FILD instructions.

The **FLD (load floating-point value)** instruction copies a floating-point operand to the top of the FPU stack (known as ST(0)).

The operand can be a 32-bit, 64-bit, or 80-bit memory operand (REAL4, REAL8, REAL10) or another FPU register.

The **FILD (load integer)** instruction converts a 16-, 32-, or 64-bit signed integer source operand to double-precision floating point and loads it into ST(0). The source operand's sign is **preserved**.

The following instructions load specialized constants on the stack. They have no operands:

- **FLD1:** pushes 1.0 onto the register stack.
- **FLDL2T:** pushes  $\log_2 10$  onto the register stack.
- **FLDL2E:** pushes  $\log_2 e$  onto the register stack.
- **FLDPI:** pushes  $\pi$  onto the register stack.
- **FLDLG2:** pushes  $\log_{10} 2$  onto the register stack.
- **FLDLN2:** pushes  $\log_e 2$  onto the register stack.

- **FLDZ:** pushes 0.0 on the FPU stack.

*Here are some examples of how to use the FLD and FILD instructions:*

```
090 .data
091     dblOne    REAL8 234.56
092     dblTwo    REAL8 10.1
093     intVal    DWORD 100
094
095 .code
096     ; Load the value of dblOne onto the FPU stack.
097     fld      QWORD PTR [dblOne]
098
099     ; Load the value of dblTwo onto the FPU stack.
100     fld      QWORD PTR [dblTwo]
101
102     ; Load the integer value intVal onto the FPU stack, preserving its sign.
103     fild    DWORD PTR [intVal]
104
105     ; Load the mathematical constant pi onto the FPU stack.
106     fldpi
107
108     ; Load the constant 1.0 onto the FPU stack.
109     fld1
```

In this section, you're defining data variables and loading them onto the FPU (Floating-Point Unit) stack:

**dblOne** and **dblTwo** are double-precision floating-point variables with values 234.56 and 10.1,

respectively. These values are stored in 64 bits each.

**intval** is a 32-bit integer variable with a value of 100.

Now, let's explain the code part:

**fld QWORD PTR [dblOne]**: This instruction loads the value of dblOne (234.56) onto the FPU stack. QWORD PTR indicates that you're accessing a 64-bit value in memory. The fld instruction is used for loading floating-point values.

**fld QWORD PTR [dblTwo]**: This instruction loads the value of dblTwo (10.1) onto the FPU stack, similar to the previous instruction.

**fild DWORD PTR [intval]**: This instruction is different. It loads the 32-bit integer value intval (100) onto the FPU stack. The fild instruction is used to load integers and preserves the sign. It converts the integer into a floating-point format in the FPU.

**fldpi**: This instruction loads the mathematical constant  $\pi$  (pi) onto the FPU stack. It's a predefined constant. (we described it above)

**fld1**: This instruction loads the constant 1.0 onto the FPU stack. It's a predefined constant representing the floating-point value 1.0.

## FST AND FSTP INSTRUCTIONS

Here is a concise explanation of the FST and FSTP instructions:

**FST (Store Floating-Point Value)**:

- Copies a floating-point operand from the top of the FPU stack to memory OR
- Stores the value on top of the FPU stack (ST(0)) to memory.
- Supports memory operand types (REAL4, REAL8, REAL10) and can store to another FPU register.
- Does not pop the stack.

The FST instruction in x86 assembly language serves a dual purpose. It can be used to copy a floating-point operand from the top of the FPU stack (ST(0)) to memory. Additionally, it can store the value at the top of the FPU stack directly to memory.

FST supports various memory operand types such as REAL4, REAL8, and REAL10, and it can also be used to store values to another FPU register. Crucially, FST does not pop the stack, leaving the values on the stack intact for further use in calculations.

#### ***FSTP (Store Floating-Point Value and Pop):***

- Copies the value in ST(0) to memory and pops ST(0) off the stack.
- Stores the value on top of the FPU stack (ST(0)) to memory and pops the stack.
- Supports the same memory operand types as FST.
- Physically removes values from the stack, changing the TOP pointer.

In contrast, the FSTP instruction not only copies the value in ST(0) to memory but also pops (removes) it from the FPU stack. This dual functionality makes it particularly useful for efficient stack management. Similar to FST, FSTP supports the same memory operand types and can store to other FPU registers.

By physically removing values from the stack, FSTP changes the TOP pointer, effectively "popping" the value from the stack, making it a preferred choice when you no longer need the value in the stack for subsequent operations.

Here's a code example illustrating the use of these instructions:

```
115 .data
116     dblOne    REAL8 234.56
117     dblTwo    REAL8 10.1
118     dblThree  REAL8 0.0
119     dblFour   REAL8 0.0
120
121 .code
122     ; Load values onto the FPU stack
123     fld QWORD PTR [dblOne]    ; ST(0) = 234.56
124     fld QWORD PTR [dblTwo]    ; ST(0) = 10.1, ST(1) = 234.56
125
126     ; Store values from the stack to memory without popping
127     fst QWORD PTR [dblThree]  ; Stores 10.1 in dblThree
128     fst QWORD PTR [dblFour]   ; Stores 10.1 in dblFour
129
130     ; Reset the values
131     fld QWORD PTR [dblOne]    ; ST(0) = 234.56
132     fld QWORD PTR [dblTwo]    ; ST(0) = 10.1, ST(1) = 234.56
133
134     ; Store values from the stack to memory and pop
135     fstp QWORD PTR [dblThree] ; Stores 10.1 in dblThree and pops ST(0)
136     fstp QWORD PTR [dblFour]  ; Stores 234.56 in dblFour and pops ST(0)
```

In this assembly code, you're working with the FPU (Floating-Point Unit) to load values onto the FPU stack and then store them in memory using the FST and FSTP instructions. Let's delve into the details without code boxes:

You begin by defining four data variables:

- **dblOne** and **dblTwo** are double-precision floating-point variables with values 234.56 and 10.1, respectively.
- **dblThree** and **dblFour** are initialized as 0.0 but will be used to store values from the FPU stack.

The fld instruction is used to load values onto the FPU stack. You load the value of dblOne (234.56) into ST(0) and then dblTwo (10.1) into ST(0), effectively pushing the previous value down in the stack (ST(1) = 234.56).

You then use the fst instruction to store the value at the top of the FPU stack (ST(0)) into memory without popping the value from the stack. You store this value in dblThree and dblFour, so both variables receive the value 10.1.

To reset the values for demonstration, you reload dblOne and dblTwo onto the FPU stack.

The fstp instruction is employed to store the value from the top of the FPU stack (ST(0)) into memory and simultaneously pop it off the stack.

This effectively removes the value from ST(0) and stores it in dblThree and dblFour. The first fstp stores 10.1 in dblThree, while the second stores 234.56 in dblFour.

The stack's top pointer (TOP) is incremented after each fstp, shifting the remaining values up.

In summary, this code illustrates how the FST and FSTP instructions allow you to move values between the FPU stack and memory.

FST stores the value in memory without altering the stack, while FSTP stores the value in memory and pops it from the stack.

These instructions are essential for efficient floating-point data handling in assembly programming.

## **FLOATING POINT ARITHMETIC**

The arithmetic instructions in x86 assembly allow you to perform basic arithmetic operations on floating-point numbers.

These instructions support the same memory operand types as **FLD (load)** and **FST (store)**, meaning operands can be **indirect**, **indexed**, **base-indexed**, and more. The key arithmetic instructions are as follows:

**FCHS (Change Sign):** This instruction reverses the sign of the floating-point value in ST(0), effectively changing it from positive to negative or vice versa.

**FABS (Absolute Value):** FABS clears the sign of the number in ST(0) to obtain its absolute value, effectively making it positive.

**FADD (Add):** FADD performs addition. It can add two values from the FPU stack or add a value from memory to a value on the stack.

**FSUB (Subtract):** FSUB subtracts the source value from the destination value.

**FSUBR (Reverse Subtract):** FSUBR subtracts the destination value from the source value.

**FMUL (Multiply):** FMUL multiplies the source value by the destination value.

**FDIV (Divide):** FDIV divides the destination value by the source value.

**FDIVR (Reverse Divide):** FDIVR divides the source value by the destination value.

**FCHS and FABS:**

FCHS is used to change the sign of the value in ST(0), essentially negating it if it was positive or making it positive if it was negative.

FABS, on the other hand, computes the absolute value by clearing the sign of the value in ST(0).

## **FADD (Add)**

FADD can be used in different formats:

### **With no operands**

It adds ST(0) to ST(1), storing the result in ST(1). It then pops ST(0), leaving the result on top of the stack.

fadd	Before:	ST(1)	<table border="1"><tr><td>234.56</td></tr><tr><td>10.1</td></tr></table>	234.56	10.1
234.56					
10.1					
		ST(0)			

	After:	ST(0)	<table border="1"><tr><td>244.66</td></tr></table>	244.66
244.66				

### With a memory operand (REAL4 or REAL8)

It adds the value in memory to the value in ST(0).

With a register number (i), it adds the value in ST(i) to ST(0).

fadd st(1), st(0)	Before:	ST(1)	<table border="1"><tr><td>234.56</td></tr><tr><td>10.1</td></tr></table>	234.56	10.1
234.56					
10.1					
		ST(0)			

	After:	ST(1)	<table border="1"><tr><td>244.66</td></tr></table>	244.66
244.66				
		ST(0)		

The format "FADD ST(i), ST(0)" adds ST(0) to ST(i), effectively **swapping their positions**.

## Memory Operand with FADD

When FADD is used with a memory operand, it adds the value stored in memory to ST(0), the top of the FPU stack.

For instance, "fadd mySingle" adds the value stored in the memory location "mySingle" to ST(0), effectively increasing its value.

```
141 .data
142 mySingle REAL4 5.5          ; Define a single-precision floating-point value in memory
143 myDouble REAL8 10.1         ; Define a double-precision floating-point value in memory
144 result  REAL8 0.0           ; Define a memory location to store the result
145
146 .code
147 main:
148     fld    myDouble          ; Load myDouble onto the FPU stack
149     fadd   mySingle          ; Add mySingle to ST(0)
150
151     fstp   result            ; Store the result in the "result" memory location
152
153     ; Exit the program (endless loop to prevent immediate termination)
154     mov    eax, 1              ; Specify the exit system call
155     int    0x80                ; Call the kernel
156
157     ; Rest of the program here (not shown in this example)
```

In this example, we have defined two floating-point values, `mySingle` and `myDouble`, in memory.

We load `myDouble` onto the FPU stack using the `fld` instruction, and then we use `fadd` to add the value of `mySingle` to the value in `ST(0)`.

Finally, we store the result in the memory location named `result` using `fstp`. The program then exits.

Please note that this code is provided for educational purposes and assumes a Linux environment.

The system call (`int 0x80`) is specific to Linux, and you may need to adjust it for other operating systems. Additionally, make sure to use the appropriate assembly syntax for your assembler and platform.

## FADDP (Add with Pop)

`FADDP`, the "add with pop" instruction, performs an addition operation and then pops the value in `ST(0)` from the stack.

This is particularly useful for efficiently managing the stack.

The syntax for `FADDP` is "`FADDP ST(1), ST(0)`", indicating that the result of the addition is stored in `ST(1)`, and `ST(0)` is removed from the stack.

faddp st(1), st(0)

Before:	ST(1)	234.56
	ST(0)	10.1
After:	ST(0)	244.66

## FIADD (Add Integer)

The FIADD instruction is used to add an integer value to ST(0) after converting the source operand to double-extended precision floating-point format.

It supports two operand types: **m16int** and **m32int**, which refer to 16-bit and 32-bit integers, respectively.

For example, "fiadd myInteger" adds the value stored in the memory location "myInteger" to ST(0) after converting it to a floating-point format.

```
158 .data
159 myInteger DWORD 1          ; Define a 32-bit integer value in memory
160 result REAL8 0.0           ; Define a memory location to store the result
161
162 .code
163 main:
164     finit                ; Initialize the FPU
165     fld      myInteger    ; Load myInteger onto the FPU stack and convert it to a floating-point value
166     fadd                 ; Add ST(0) to ST(1), effectively adding myInteger to ST(0)
167
168     fstp     result       ; Store the result in the "result" memory location
169
170 ; Exit the program (endless loop to prevent immediate termination)
171     mov      eax, 1        ; Specify the exit system call
172     int      0x80          ; Call the kernel
173
174 ; Rest of the program here (not shown in this example)
```

*In this code:*

- We define an integer value, `myInteger`, in memory and a memory location named `result` to store the result.
- We use the `finit` instruction to initialize the FPU.
- We load the integer value `myInteger` onto the FPU stack and convert it to a floating-point value using `fld`.
- Then, we use `fadd` to add the value in `ST(0)` to `ST(1)`, which effectively adds the value of `myInteger` to the value in `ST(0)`.

- Finally, we store the result in the result memory location using fstp.

The system call (int 0x80) is specific to Linux, and you may need to adjust it for Windows or other operating systems. Make sure to use the appropriate assembly syntax for your platform.

## FSUB (Floating-Point Subtract):

The FSUB instruction performs subtraction in the FPU. It subtracts a source operand from a destination operand, storing the difference in the destination operand.

The destination is always an FPU register (ST(0)), and the source can be either an FPU register or memory. It supports the same memory operand types as FADD.

The operation of FSUB is similar to that of FADD, except it performs subtraction.

### Without Operands

For example, if used **without operands**, it subtracts ST(0) from ST(1), temporarily storing the result in ST(1). ST(0) is then popped from the stack, leaving the result on top.

Examples:

- fsub mySingle: This instruction subtracts the value in mySingle from ST(0), effectively performing ST(0) -= mySingle.
- fsub array[edi\*8]: Here, the value stored in array[edi\*8] is subtracted from ST(0). This operation

doesn't pop the stack, leaving the result in ST(0).

### **FSUBP (Floating-Point Subtract with Pop)**

The FSUBP instruction is similar to FSUB but with an additional step.

It performs the subtraction and then pops (removes) ST(0) from the stack.

This means that after executing FSUBP, the result is left in ST(0), and the stack pointer is adjusted accordingly.

```
180 .data
181 mySingle REAL8 5.0          ; Example value for mySingle
182 array REAL8 10.0           ; Example value for array[edi*8]
183
184 .code
185 ; Subtract mySingle from ST(0) using FSUB
186 fld QWORD PTR [mySingle]    ; Load mySingle onto the FPU stack
187 fsub                      ; Subtract mySingle from ST(0)
188 ; The result is now in ST(0)
189
190 ; Subtract array[edi*8] from ST(0) using FSUB
191 fld QWORD PTR [array+edi*8] ; Load array[edi*8] onto the FPU stack
192 fsub                      ; Subtract array[edi*8] from ST(0)
193 ; The result is now in ST(0)
194
195 ; Now, let's use FSUBP to subtract and pop ST(0)
196
197 ; Subtract 5.0 from ST(0) and pop the stack
198 fld QWORD PTR [mySingle]    ; Load mySingle onto the FPU stack
199 fsubp                     ; Subtract mySingle from ST(0) and pop ST(0)
200 ; The result is left in ST(0), and ST(0) is removed from the stack
```

Example:

fsubp ST(1), ST(0): This instruction subtracts ST(0) from ST(1) and leaves the result in ST(0) while

removing the old ST(0) from the stack.

## FISUB (Floating-Point Subtract Integer):

The FISUB instruction is used to subtract an integer from a floating-point value in ST(0).

It first converts the source operand (an integer) to double-extended precision floating-point format before performing the subtraction.

This allows you to perform arithmetic operations involving integers and floating-point numbers in the FPU.

```
202 .data
203 myInteger DWORD 3      ; Example integer value for myInteger
204
205 .code
206 ; Subtract an integer from ST(0) using FISUB
207 fld QWORD PTR [myInteger] ; Load myInteger as a 64-bit integer onto the FPU stack
208 fsub                 ; Subtract myInteger from ST(0)
209 ; The result is now in ST(0)
210
211 ; Use FSUBP to subtract ST(0) from ST(1) and pop ST(0)
212 fsubp ST(1), ST(0)
213 ; After executing this, the result is in ST(0), and the old ST(0) has been removed from the stack
```

These code example show how to use FISUB to subtract an integer from an FPU register and FSUBP to subtract and pop the stack, as well as the explanation you provided for fsubp ST(1), ST(0).

## Examples:

fisub m16int: This instruction subtracts the 16-bit integer from ST(0) after converting it to a floating-point format.

fisub m32int: Similarly, this instruction subtracts the 32-bit integer from ST(0) after converting it to a floating-point format.

```
220 .data
221     my16BitInt WORD 5      ; Example 16-bit integer value
222     my32BitInt DWORD 10    ; Example 32-bit integer value
223
224 .code
225     ; Subtract a 16-bit integer from ST(0) using FISUB
226     fld QWORD PTR [my16BitInt] ; Load my16BitInt as a 16-bit integer onto the FPU stack
227     fisub                  ; Subtract my16BitInt from ST(0)
228     ; The result is now in ST(0)
229
230     ; Reset ST(0) to another value
231     fld QWORD PTR [my32BitInt] ; Load my32BitInt as a 32-bit integer onto the FPU stack
232
233     ; Subtract a 32-bit integer from ST(0) using FISUB
234     fisub                  ; Subtract my32BitInt from ST(0)
235     ; The result is now in ST(0)
```

These code examples demonstrate how to use FISUB to subtract 16-bit and 32-bit integers from an FPU register after converting them to floating-point format.

This provides flexibility for performing arithmetic operations with different data types and operands in floating-point arithmetic.

## FMUL (floating-point multiply)

The FMUL (floating-point multiply) instruction is used to multiply a source operand by a destination operand, and the result is stored in the destination operand, which is always an FPU register.

The source can be either an FPU register or a memory operand, similar to FADD and FSUB:

### FMUL:

FMUL can be used **with no operands**, in which case it multiplies ST(0) by ST(1), and the result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the product in ST(0).

When used **with a memory operand**, FMUL multiplies ST(0) by the value stored in the memory operand.

**FMULP (floating-point multiply with pop)** is similar to FMUL, but it pops ST(0) from the stack after performing the multiplication.

This means that after executing FMULP, the result is left in ST(0), and the stack pointer is adjusted accordingly.

**FIMUL (floating-point integer multiply)** is identical to FIADD, except it performs multiplication instead of addition.

It converts the source operand (an integer) to double-extended precision floating-point format

before performing the multiplication.

*Here's an example in assembly code:*

```
240 .data
241     mySingle REAL4 3.0          ; Example single-precision floating-point value
242     my16BitInt WORD 2          ; Example 16-bit integer
243     my32BitInt DWORD 4          ; Example 32-bit integer
244     result REAL8 0.0           ; Storage for results
245
246 .code
247     ; Multiply ST(0) by a single-precision floating-point value
248     fld QWORD PTR [mySingle]    ; Load mySingle as a single-precision value onto the FPU stack
249     fmul                      ; Multiply ST(0) by mySingle
250     fstp QWORD PTR [result]    ; Store the result in result
251
252     ; Reset ST(0) to another value
253     fld QWORD PTR [my32BitInt]  ; Load my32BitInt as a 32-bit integer onto the FPU stack
254
255     ; Multiply ST(0) by another value
256     fld QWORD PTR [mySingle]    ; Load another single-precision value onto the FPU stack
257     fmul                      ; Multiply ST(0) by the new single-precision value
258     fstp QWORD PTR [result]    ; Store the result in result
259
260     ; Reset ST(0) to another value
261     fld QWORD PTR [my16BitInt]  ; Load my16BitInt as a 16-bit integer onto the FPU stack
262
263     ; Multiply ST(0) by another value
264     fld QWORD PTR [mySingle]    ; Load another single-precision value onto the FPU stack
265     fimul WORD PTR [esp]        ; Multiply ST(0) by the 16-bit integer
266     fstp QWORD PTR [result]    ; Store the result in result
```

Let's explain the provided code:

## **Variable Initialization:**

In the .data section, we define several variables. mySingle is set to a single-precision floating-point value (3.0), my16BitInt to a 16-bit integer (2), my32BitInt to a 32-bit integer (4), and result is initialized to store the results of the various multiplications.

## **Single-Precision Floating-Point Multiplication:**

The first part of the code demonstrates single-precision floating-point multiplication. We load the value stored in mySingle onto the FPU stack using fld.

This places the single-precision value in the top of the FPU stack, which is ST(0). We then use the fmul instruction to multiply the value in ST(0) (the value from mySingle) by itself.

The result remains in ST(0). Finally, the fstp instruction is used to store the result in the result variable.

## **Resetting the FPU Stack:**

We reset the FPU stack to a different value (my32BitInt) using another fld instruction. This prepares the FPU stack for the next multiplication operation.

## **Single-Precision Floating-Point Multiplication (Again):**

Similar to the first part, we load a single-precision value onto the FPU stack, this time using fld to place a new value into ST(0).

We then use the fmul instruction to multiply the value in ST(0) by the new single-precision value. The result remains in ST(0).

Once again, the `fstp` instruction is used to store the result in the `result` variable.

#### **Resetting the FPU Stack Again:**

We reset the FPU stack to a different value (`my16BitInt`) using yet another `fld` instruction.

This prepares the FPU stack for the next multiplication operation.

#### **Integer Multiplication with FIMUL:**

In this part, we load a 16-bit integer value onto the FPU stack using `fld`. We then load another single-precision value onto the FPU stack.

However, instead of using `fmul` as in the previous scenarios, we use the `fimul` instruction with WORD PTR [esp].

This performs integer multiplication between the 16-bit integer in `ST(0)` and the single-precision value, and the result remains in `ST(0)`.

Finally, the `fstp` instruction is used to store the result in the `result` variable.

#### **Completion and Result Storage:**

The code concludes with the result of the integer multiplication stored in the `result` variable.

This code showcases various scenarios of floating-point and integer multiplication operations in the x86 FPU, demonstrating how to load values, perform multiplications, and store the results for different data types and operand combinations.

## FDIV (Floating-Point Division):

The FDIV instruction is used to perform division operations within the x86 FPU.

It divides the value in a destination operand by the value in a source operand, with the result stored in the destination operand.

The destination operand must be an FPU register, while the source operand can be either an FPU register or a memory location.

The syntax is similar to other FPU arithmetic instructions, supporting both memory and register operands.

For example, **fdiv ST(0), ST(i)** divides ST(i) by ST(0), and **fdiv m64fp** divides ST(0) by the value in memory specified by m64fp.

Handling special cases is essential when dealing with division.

For instance, division by zero results in a divide-by-zero exception, so it's crucial to ensure that the divisor is not zero.

Special situations arise when **dividing by infinity, zero, or NaN (Not-a-Number)**, each governed by specific rules as detailed in the Intel Instruction Set Reference manual.

## FIDIV (Floating-Point Divide Integer):

The FIDIV instruction is used to perform integer division within the x86 FPU.

Before the division operation, it converts the source operand (an integer) to double-extended precision floating-point format, ensuring compatibility with the FPU's floating-point arithmetic.

Syntax examples include **fidiv m16int** for dividing a 16-bit integer and **fidiv m32int** for dividing a 32-bit integer.

The FIDIV instruction is particularly useful when you need to perform arithmetic operations involving integers and floating-point numbers in the FPU.

*Now, let's provide a code that showcases various division scenarios, including division by different data types and operand combinations.*

```
270 .data
271     dblOne REAL8 1234.56
272     dblTwo REAL8 10.0
273     intDivisor WORD 2
274     quotient REAL8 ?
275
276 .code
277     ; Single-Precision Floating-Point Division
278     fld dblOne          ; Load dblOne onto the FPU stack
279     fdiv dblTwo          ; Divide ST(0) by dblTwo
280     fstp quotient        ; Store the result in quotient
281
282     ; Reset the FPU stack
283     fld dblOne          ; Load dblOne again
284
285     ; Double-Precision Floating-Point Division
286     fdiv dblTwo          ; Divide ST(0) by dblTwo
287     fstp quotient        ; Store the result in quotient
288
289     ; Integer Division Using FIDIV
290     fld intDivisor       ; Load the integer divisor onto the FPU stack
291     fidiv intDivisor     ; Divide ST(0) by intDivisor
292     fstp quotient        ; Store the result in quotient
```

Let's describe the provided assembly code in paragraphs to understand its functionality.

In the provided assembly code, we are performing various division operations using the x86 Floating-Point Unit (FPU).

These operations involve both floating-point and integer division. The code showcases different scenarios of division and stores the results in the quotient variable.

The code begins by defining some data elements in the .data section. It sets up two floating-point variables, dblOne and dblTwo, with respective values.

Additionally, it defines an integer variable, intDivisor, and an empty variable named quotient, which will hold the results of the division operations.

The .code section is where the actual division operations are performed. First, it loads the value of dblOne onto the FPU stack using the fld (load) instruction.

This value represents a double-precision floating-point number.

Next, it performs single-precision floating-point division by dividing ST(0) (top of the stack) by the value of dblTwo using the fdiv (division) instruction.

The result is then stored in the quotient variable using fstp (store and pop).

The code resets the FPU stack by loading dblOne again. This time, it performs double-precision floating-point division by dividing ST(0) by dblTwo.

This demonstrates division with larger precision.

Lastly, the code performs integer division using the FIDIV (integer divide) instruction.

It loads the value of intDivisor onto the FPU stack and divides ST(0) by this integer value. The result is then stored in the quotient variable.

In summary, the code illustrates various division scenarios using the x86 FPU.

It covers single-precision and double-precision floating-point division as well as integer division, providing a comprehensive example of division operations within the FPU.

The results are stored in the quotient variable, making it a versatile demonstration of division in assembly language.

## ***Comparing Floating-Point Values***

The passage below describes how to compare floating-point values and branch to a label based on the conditions.

To compare floating-point values, you can use the **FCOM instruction**.

The **FCOM instruction** compares the value in ST(0) to the source operand, which can be a memory operand or an FPU register.

After executing FCOM, you can use the **FNSTSW instruction** to move the FPU status word into AX.

The FPU status word contains the condition codes, which indicate the results of the comparison.

Once the **condition codes are in AX**, you can use the SAHF instruction to copy AH into the EFLAGS register.

The **EFLAGS register** contains the CPU status flags, including the Zero, Parity, and Carry flags.

Finally, you can use **conditional jump instructions to branch** to a label based on the condition codes.

For example, the following code branches to the label greater\_than if the value in ST(0) is greater than the value in ST(1):

```
295 ; Load the value in ST(0) to ST(0) (no change)
296 fld st(0)
297
298 ; Load the value in ST(1) to ST(0)
299 fld st(1)
300
301 ; Compare ST(0) to ST(1) and set FPU condition codes
302 fcom
303
304 ; Move the FPU status word into the AX register
305 fnstsw ax
306
307 ; Copy the AH register (containing FPU condition codes) to EFLAGS
308 sahf
309
310 ; Check if the result of the comparison was greater (JG stands for "jump if greater")
311 jg greater_than
312
313 ; Your code for handling the case where the comparison result was not greater would go here
314
315 greater_than:
316 ; Your code for handling the case where the comparison result was greater goes here
```

In this code, the `greater_than` label marks the point where you'll continue if the result of the comparison (using FCOM) is greater.

If the result is not greater, you can place your code for handling that case in the section labeled "Your code for handling the case where the comparison result was not greater goes here."

This code sequence loads two values from the FPU stack, compares them using FCOM, transfers the FPU status word into the AX register (to obtain the FPU condition codes), copies the relevant condition code to EFLAGS using SAHF, and then uses the JG instruction to check if the result was greater, jumping to the greater\_than label accordingly.

---

The following table shows the condition codes and the corresponding conditional jump instructions:

Condition code	Description	Conditional jump instruction
CF = 1 and ZF = 0	ST(0) is greater than the source operand.	JG
CF = 0 and ZF = 0	ST(0) is less than the source operand.	JL
CF = 0 and ZF = 1	ST(0) is equal to the source operand.	JE
CF = 1 and ZF = 1	The comparison is unordered.	JAE

The comparison is unordered if either operand is a NaN (Not a Number) or if the two operands have different signs and one of them is zero.

The ability to compare floating-point values and branch to a label based on the conditions is essential for many applications, such as graphics, scientific computing, and financial modeling.

## FCOM and FCOMI instruction

Let's describe the FCOMI instruction and how to use it to branch to a label based on the condition codes after comparing two floating-point values.

The FCOMI instruction is a P6 family instruction that compares floating-point values and sets the Zero, Parity, and Carry flags directly. This eliminates the need to use the FNSTSW and SAHF instructions to move the FPU status word into AX and copy AH into the EFLAGS register.

To use the FCOMI instruction, you simply pass the two floating-point values that you want to compare as operands. The FCOMI instruction will then set the Zero, Parity, and Carry flags based on the result of the comparison.

The following table shows the condition codes set by the FCOMI instruction:

Condition	C3 (Zero Flag)	C2 (Parity Flag)	C0 (Carry Flag)	Conditional Jump to Use
ST(0) > SRC	0	0	0	JA, JNBE
ST(0) < SRC	0	0	1	JB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
Unordered <sup>a</sup>	1	1	1	(None)

The unordered condition is a special case that occurs when either operand is a NaN (Not a Number) or if the two operands have different signs and one of them is zero.

In this case, the condition codes are set to 111, which does not correspond to any conditional jump instruction.

Once the condition codes have been set by the FCOMI instruction, you can use conditional jump instructions to branch to a label based on the result of the comparison.

For example, the following code branches to the L1 label if the value in ST(0) is less than the value in ST(1):

```
330 ; Load the value of Y onto the FPU stack
331 fld Y          ; ST(0) = Y
332
333 ; Load the value of X onto the FPU stack, creating a stack with X on top and Y below
334 fld X          ; ST(0) = X, ST(1) = Y
335
336 ; Compare the values on the FPU stack (X and Y)
337 fcomi ST(0), ST(1) ; Compare ST(0) to ST(1)
338
339 ; Jump to label L1 if not below (if X is not less than Y), skipping the next instruction
340 jnb L1          ; Jump if not below (ST(0) not < ST(1)?), skip to L1
341
342 ; If the jump condition is not met, set the integer N to 1
343 mov N, 1        ; N = 1
344
345 ; Label L1 for reference
346 L1:
```

Here's a more detailed explanation of the code snippet:

**fld Y:** This instruction loads the value of Y (the second floating-point number) onto the FPU stack. As a result, ST(0) now contains the value of Y.

**fld X:** The next instruction loads the value of X (the first floating-point number) onto the FPU stack. This action shifts Y into ST(1), and X occupies ST(0), making it ready for comparison.

**fcomi ST(0), ST(1):** The fcomi instruction performs a comparison between the values in ST(0) and ST(1). In this case, it compares X (ST(0)) to Y (ST(1)). If ST(0) is not less than ST(1) (in other words, X is not less than Y), it sets the carry flag (CF) to 1.

**jnb L1:** The jnb (jump if not below) instruction checks the carry flag (CF) and transfers control to the label L1 if CF is not set. In this context, it means that if X is not less than Y, the code will skip the next instruction and move to the label L1.

**mov N, 1:** When the comparison determines that X is indeed less than Y, the code sets the integer N to 1. This assignment means that N will take the value 1 when the condition is met.

**L1::** This label serves as a marker in the code, allowing you to return to this point if needed. In this particular scenario, it signifies the end of the conditional block.

The code snippet effectively compares two floating-point values (X and Y) and, depending on the result, sets the integer N to 1 if X is less than Y.

It demonstrates how the FPU condition codes, particularly the carry flag, can be used to control program flow based on floating-point comparisons.

---

FCOM compares two floating-point values on the FPU stack and sets condition flags based on the

result.

Here's an example that illustrates how to perform a comparison between two double-precision floating-point numbers, X and Y, and then set an integer N based on the result. In this case, if X is less than Y, N is set to 1.

```
350 .data
351     X REAL8 1.2
352     Y REAL8 3.0
353     N DWORD 0
354
355 .code
356     ; if( X < Y )
357     ;
358     N = 1
359     fld X    ; Load X into ST(0)
360     fcomp Y   ; Compare ST(0) to Y
361     fnstsw ax   ; Move status word into AX
362     sahf    ; Copy AH into EFLAGS
363     jnb L1    ; X not < Y? Skip
364     mov N, 1    ; N = 1
365 L1:
```

This code snippet demonstrates how the FPU condition codes are set and how conditional jumps are used to control program flow based on the comparison results.

An improvement to consider is the usage of the FCOMI instruction, available on newer processors like the Intel P6 family (e.g., Pentium Pro and Pentium II).

FCOMI is a more efficient instruction that directly compares two values on the FPU stack and sets condition flags. It's designed for improved performance in comparison operations compared to FCOM.

The FCOMI instruction can perform floating-point comparisons and set the Zero, Parity, and Carry flags directly. Here's the same comparison using FCOMI:

```
350 .data
351     X REAL8 1.2
352     Y REAL8 3.0
353     N DWORD 0
354
355 .code
356     ; if( X < Y )
357     ;
358     N = 1
359     fld Y    ; Load Y into ST(0)
360     fld X    ; Load X into ST(0), Y is now in ST(1)
361     fcomi ST(0), ST(1)    ; Compare ST(0) to ST(1)
362     jnb L1    ; ST(0) not < ST(1)? Skip
363     mov N, 1    ; N = 1
364 L1:
```

## COMPARING FOR EQUALITY

The passage you sent describes how to compare floating-point values for equality in assembly language.

The proper way to compare floating-point values for equality is to take the **absolute value of their difference**,  $|x - y|$ , and compare it to a small user-defined value called **epsilon**.

This is because floating-point numbers are represented approximately in computers, and even small rounding errors can accumulate over time.

The following code in assembly language compares two floating-point values, val2 and val3, for equality using a tolerance of epsilon:

```
375 .data
376     epsilon REAL8 1.0E-12    ; Define epsilon as the tolerance
377     val2 REAL8 0.0           ; Define val2 as the first value to compare
378     val3 REAL8 1.001E-13    ; Define val3 as the second value, considered equal to val2
379
380 .code
381     fld epsilon             ; Load epsilon into the FPU stack
382     fld val2                ; Load val2 into the FPU stack
383     fsub val3               ; Subtract val3 from val2
384     fabs                  ; Compute the absolute value of the difference
385     fcomi ST(0), ST(1)      ; Compare ST(0) to ST(1) and set condition flags
386     ja skip                ; Jump if the absolute difference is greater
387     ; If we reach here, the values are considered equal
388     mWrite <"Values are equal", 0dh, 0ah> ; Display "Values are equal"
389     skip:
```

This assembly code is designed to compare two floating-point values, val2 and val3, for equality while considering a tolerance level defined as epsilon.

The purpose of this code is to address a common challenge when working with floating-point numbers, which is that due to precision limitations, exact equality comparisons may not yield the expected results.

Therefore, it's a common practice to compare values within a certain tolerance range instead.

### Here's a step-by-step explanation of the code:

Data Definitions: In the .data section, we define three variables:

- **epsilon**: This is set to 1.0E-12, representing the tolerance level for considering two values as equal.
- **val2**: This variable holds the first value for comparison, initialized with 0.0.
- **val3**: This variable holds the second value, which is considered equal to val2 and is set to 1.001E-13.

Code Section: In the .code section, the actual comparison is performed.

- **fld epsilon**: The fld instruction loads the value of epsilon onto the FPU stack. This value represents the maximum allowable difference between two values for them to be considered equal.
- **fld val2**: The next instruction loads the value of val2 onto the FPU stack. This sets up the first value for comparison.
- **fsub val3**: Here, we subtract the value of val3 from the value of val2. This computes the difference between the two values.
- **fabs**: The fabs instruction computes the absolute value of the difference obtained in the previous step. This ensures that we're only looking at the magnitude of the difference, regardless of its sign.
- **fcomi ST(0), ST(1)**: The fcomi instruction performs the actual comparison between the top of the FPU stack (ST(0)) and the next value (ST(1)). It sets certain condition flags based on the result of this comparison. Specifically, it sets the Carry Flag (CF) and Zero Flag (ZF) based on the relationship between the two values.
- **ja skip**: The ja instruction is a conditional jump that stands for "jump if above." It checks the CF and ZF flags to determine whether the absolute difference (ST(0)) is greater than the second

value (ST(1)), considering the defined tolerance. If the jump condition is met (i.e., if CF=1 and ZF=0), it means the absolute difference is greater than epsilon, and we jump to the skip label.

- **mWrite <"Values are equal", 0dh, 0ah>**: If the ja condition is not met, it implies that the absolute difference is within the tolerance range defined by epsilon, and the values are considered equal.
- In this case, the code proceeds to display the message "Values are equal." The mWrite instruction is not a standard x86 assembly instruction but is used here to represent a placeholder for a function that would display the message.
- The 0dh and 0ah are carriage return and line feed characters for formatting.
- **skip::**: This label is where the code jumps to if the absolute difference is greater than epsilon. It serves as the point to continue execution after the comparison.

In summary, this code demonstrates how to compare floating-point values with a tolerance level (epsilon) to determine if they are equal within a certain range.

It accounts for the precision limitations of floating-point arithmetic and is a common practice when dealing with real-world numerical computations.

---

#### The code works summary:

- Load the tolerance value, epsilon, onto the FPU stack.
- Load the first value, val2, onto the FPU stack.
- Subtract the second value, val3, from val2.

- Take the absolute value of the difference.
- Compare the absolute value of the difference to the tolerance value.
- If the absolute value of the difference is greater than the tolerance value, jump to the skip label.
- Otherwise, display the message "Values are equal".
- This code will display the message "Values are equal" if the difference between val2 and val3 is less than or equal to the tolerance value, epsilon. Otherwise, the program will skip over the mWrite instruction and continue execution.

It is important to note that the choice of an appropriate tolerance value depends on the specific application.

For example, in a graphics application, a tolerance value of  $1.0E^{-6}$  might be sufficient.

However, in a financial modeling application, a tolerance value of  $1.0E^{-12}$  or less might be required.

## ***Floating-point input-output procedures***

The passage you sent describes two procedures for floating-point input/output in assembly language:

- **ReadFloat:** Reads a floating-point value from the keyboard and pushes it on the floating-point stack.

invertedtomato/feather

## #1 **ReadFloat()** makes the stream not readable?



- **WriteFloat:** Writes the floating-point value at ST(0) to the console window in exponential format.

protocolbuffers/protobuf

## #8476 **WriteFloat and WriteDouble failed for Unity game on Android.**



The ReadFloat procedure accepts a wide variety of floating-point formats, including:

35  
+35.  
-3.5  
.35  
3.5E5  
3.5E005  
-3.5E+5

3.5E-4

+3.5E-4

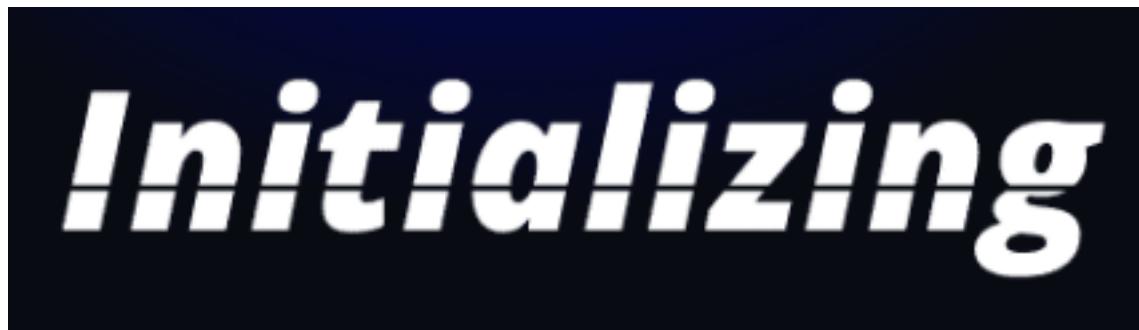
The WriteFloat procedure writes the floating-point value at ST(0) to the console window in exponential format.

Here's the example program that demonstrates the use of the ReadFloat and WriteFloat procedures in assembly language. This program pushes two floating-point values onto the FPU stack, displays the FPU stack, takes user input for two values, multiplies them, and displays their product. I'll provide you with the assembly code:

```
410 ; 32-bit Floating-Point I/O Test (floatTest32.asm)
411 INCLUDE Irvine32.inc
412 INCLUDE macros.inc
413 .data
414     first REAL8 123.456
415     second REAL8 10.0
416     third REAL8 ?
417
418 .code
419     main PROC
420         finit          ; Initialize FPU
421         ; Push two floats and display the FPU stack.
422         fld first       ; Push the first value onto the FPU stack.
423         fld second      ; Push the second value onto the FPU stack.
424         call ShowFPUStruct ; Display the FPU stack.
425         ; Input two floats and display their product.
426
427         mWrite "Please enter a real number: "
428         call ReadFloat    ; Read the first floating-point number.
429         mWrite "Please enter a real number: "
430         call ReadFloat    ; Read the second floating-point number.
431
432         fmul           ; Multiply ST(0) by ST(1).
433
434         mWrite "Their product is: "
435         call WriteFloat   ; Display the product.
436         call CrLf        ; Add a line break.
437
438         exit
439     main ENDP
440 END main
```

This assembly code is designed to demonstrate the use of floating-point input and output procedures while performing basic arithmetic operations on these floating-point values using the FPU (Floating-Point Unit). Let's break down the code step by step.

**Initialization (finit):** The program begins by initializing the FPU using the finit instruction. This is a necessary step to prepare the FPU for floating-point operations.



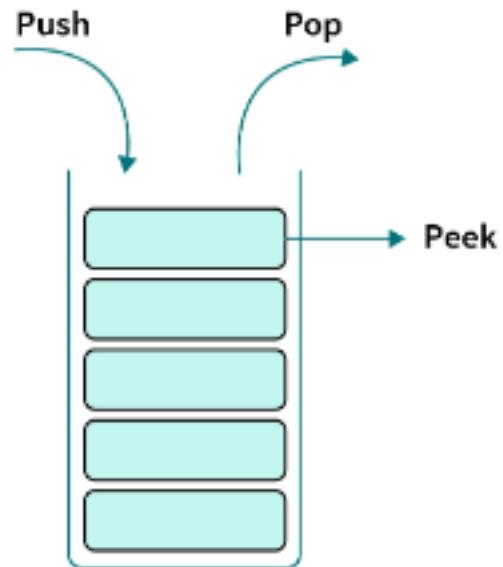
**Pushing Values onto the FPU Stack:** Two floating-point values are pushed onto the FPU stack. These values are stored in memory as first and second. The fld (floating-point load) instructions are used to load these values onto the FPU stack. fld first pushes the value of first onto the stack, and fld second pushes the value of second onto the stack.

## The FPU Stack

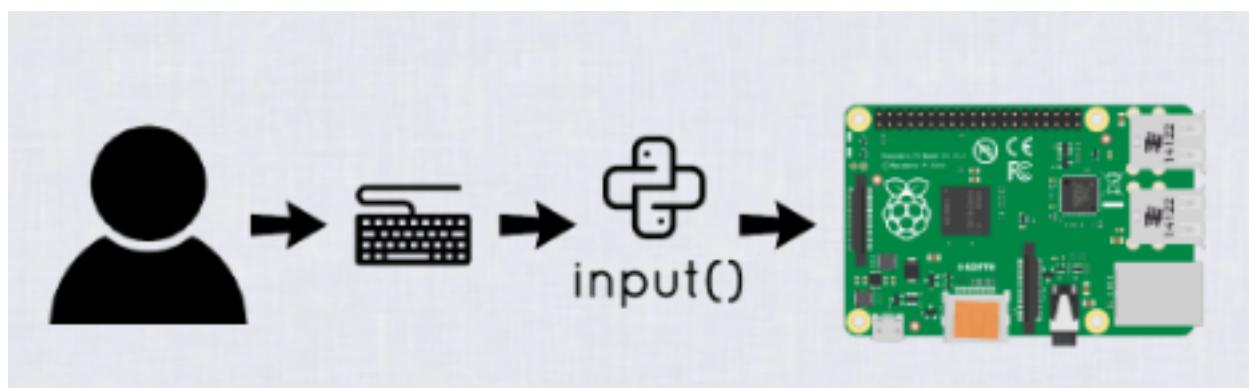
- When we push, it refers to the next register.



**Displaying the FPU Stack:** After pushing the values onto the FPU stack, the program calls a custom procedure called `ShowFPUStruct`. This procedure is responsible for displaying the contents of the FPU stack. It helps visualize the values stored on the stack at this point.

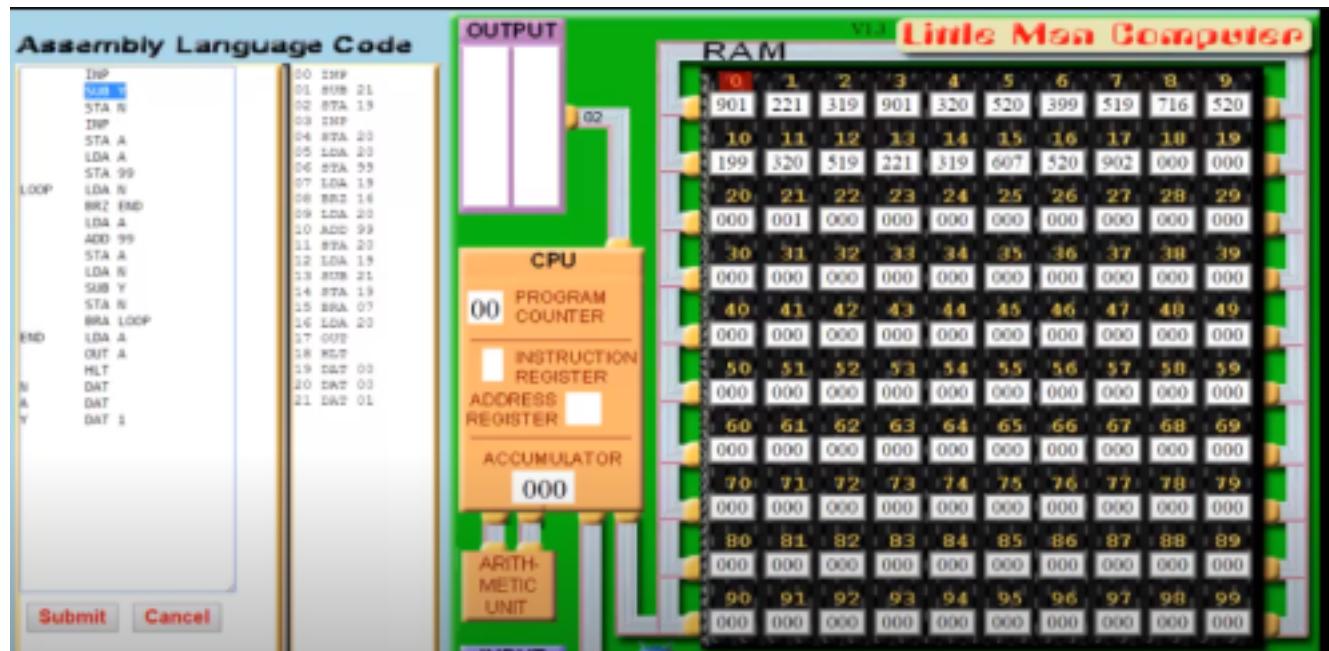


**User Input:** The program prompts the user to enter two real numbers. It uses the `mWrite` function to display the input prompt. Then, it calls the `ReadFloat` procedure, which reads the user's input as a floating-point number. This process is repeated for the second number.



**Multiplication (fmul):** Once both user inputs are on the FPU stack, the program uses the `fmul` instruction to multiply these values. The `fmul` instruction multiplies the value on top of the stack (`ST(0)`) by the next value (`ST(1)`) and stores the result in `ST(0)`. In this case, it effectively

calculates the product of the two numbers entered by the user.



**Displaying the Result:** After the multiplication is performed, the program uses mWrite to display the text "Their product is: " to the console. Then, it calls the WriteFloat procedure to display the result of the multiplication in exponential format.

# DISPLAY

## Program in Assembly language

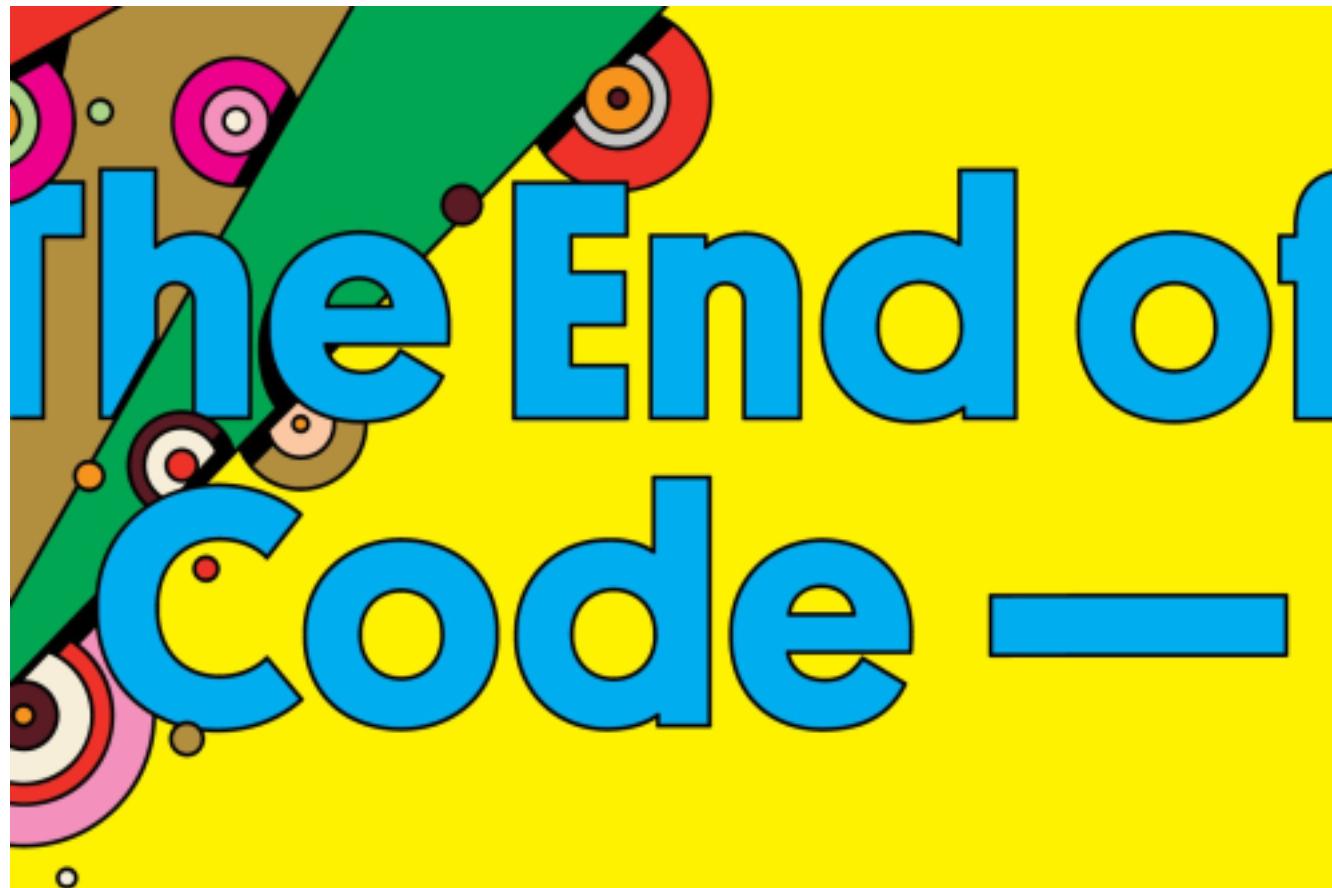
0102 0000	ADD	[BX+SI],AL	DS:0018	FF FF FF FF FF FF FF FF
0104 0000	ADD	[BX+SI],AL	DS:0020	FF FF FF FF FF FF FF FF
0106 0000	ADD	[BX+SI],AL	DS:0028	FF FF FF FF 96 11 E4 FF
0108 0000	ADD	[BX+SI],AL	DS:0030	92 01 14 00 18 00 9C 11
010A 0000	ADD	[BX+SI],AL	DS:0038	FF FF FF FF 00 00 00 00
010C 0000	ADD	[BX+SI],AL	DS:0040	05 00 00 00 00 00 00 00
010E 0000	ADD	[BX+SI],AL	DS:0048	00 00 00 00 00 00 00 00

---

Z	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
DS:0000	CD	20	9C	11	00	EA	FD	FF	AD	DE	ED	04	92	01	00	00
DS:0010	18	01	10	01	18	01	92	01	FF							
DS:0020	FF	96	11	E4	FF											
DS:0030	92	01	14	00	18	00	9C	11	FF	FF	FF	FF	00	00	00	00
DS:0040	05	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

1 Step 2StepProc 3Retrieve 4 Help 5Set BRK 6 7 up 8 dn 9 le 6 ri

**End of Program:** Finally, the program adds a line break using Crlf for a clean console output and exits.



The primary purpose of this program is to showcase the handling of floating-point values, input, and output, as well as basic arithmetic operations on these values using the FPU.

The FPU stack is crucial in managing these floating-point values during the operations, and the code illustrates the sequence of actions involved in working with the FPU for floating-point calculations.

# **Exception Synchronization**

The passage below describes the potential problem of floating-point exceptions in concurrent systems and how to use the WAIT and FWAIT instructions to solve it.

In a concurrent system, two or more tasks can execute at the same time. This can be a problem for floating-point exceptions because the FPU and the CPU are separate units.

If an unmasked **floating-point exception occurs while the CPU is executing an integer or system instruction**, the exception will not be handled until the next floating-point instruction or the FWAIT instruction is executed.

This can lead to problems if the floating-point instruction that caused the exception is followed by an integer or system instruction that modifies the same memory operand as the floating-point instruction.

For example, the following code could cause a problem:

```
445 .data
446     intValue DWORD 25 ; Define an integer value
447
448 .code
449     fild intValue      ; Load the integer value into ST(0)
450     inc intValue       ; Increment the integer value
```

This code loads the integer value stored in intValue into the FPU stack register ST(0) and then increments the integer value by 1.

Please note that in this context, the inc instruction increments the integer value stored in memory at the address specified by `intVal`.

If an unmasked floating-point exception occurs while the FILD instruction is executing, the exception will not be handled until the INC instruction is executed.

If the INC instruction modifies the same memory operand as the FILD instruction, the exception handler will not be able to access the correct value of the memory operand.

The WAIT and FWAIT instructions can be used to solve this problem.

Both instructions force the processor to check for pending, unmasked floating-point exceptions before proceeding to the next instruction.

This ensures that the exception handler will have a chance to execute before any integer or system instructions that modify the same memory operand as the floating-point instruction that caused the exception.

To solve the problem in the example code above, you could add a WAIT or FWAIT instruction after the FILD instruction. For example:

```
454 .data  
455     intValue DWORD 25  
456  
457 .code  
458     ; Load the integer into ST(0)  
459     fld intValue  
460  
461     ; Wait for pending exceptions  
462     fwait  
463  
464     ; Increment the integer  
465     inc intValue
```

In this version, the comments are placed next to the relevant instructions.

The FLD instruction loads the integer into the FPU stack, followed by the FWAIT instruction to ensure any pending exceptions are handled.

Then, the INC instruction increments the integer value in memory. This sequence of instructions allows for proper exception handling before modifying the shared memory operand.

This would ensure that the exception handler would have a chance to execute before the INC instruction is executed.

### EXAMPLE CODE FOR FPU OPERATIONS:

Here's the code for the expression  $\text{valD} = -\text{valA} + (\text{valB} * \text{valC})$

```
468 .data
469     valA REAL8 1.5
470     valB REAL8 2.5
471     valC REAL8 3.0
472     valD REAL8 ? ; Initialize valD as a placeholder for the result
473
474 .code
475     ; Load valA on the FPU stack and negate it
476     fld valA          ; ST(0) = valA
477     fchs              ; Negate the value in ST(0)
478
479     ; Load valB into ST(0) and multiply by valC
480     fld valB          ; Load valB into ST(0)
481     fmul valC         ; Multiply ST(0) by valC, leaving the product in ST(0)
482
483     ; Add the two values on the stack (ST(0) and ST(1))
484     fadd              ; Add ST(0) and ST(1), result in ST(0)
485
486     ; Store the result in valD
487     fstp valD         ; Store the result in valD
```

In this code, we calculate  $\text{valD}$  as the result of the expression  $-\text{valA} + (\text{valB} * \text{valC})$ .

We start by loading valA onto the FPU stack and negating it using fchs.

Then, we load valB into ST(0) and multiply it by valC, resulting in the product in ST(0).

Finally, we add the two values on the stack and store the result in valD.

The fstp instruction pops the result from the stack and stores it in valD.

### **SUM OF ARRAY OF DOUBLE PRECISION NUMBERS**

In the code below, you are calculating the sum of an array of double-precision real numbers.

```
510 ; Define the size of the array
511 ARRAY_SIZE = 20
512
513 .data
514     sngArray REAL8 ARRAY_SIZE DUP(?)
515
516 .code
517     mov esi, 0          ; Initialize array index
518     fldz                ; Push 0.0 onto the FPU stack
519     mov ecx, ARRAY_SIZE ; Set the loop counter to ARRAY_SIZE
520
521 L1:
522     fld sngArray[esi]   ; Load the current element into ST(0)
523     fadd                ; Add ST(0) to the accumulator (ST(0)), and pop
524     add esi, TYPE REAL8 ; Move to the next element
525     loop L1              ; Continue the loop until all elements are processed
526
527     call WriteFloat      ; Display the sum in ST(0)
```

In this code, we are calculating and displaying the sum of an array of double-precision real numbers.

The size of the array is defined by the symbolic constant `ARRAY_SIZE`, which has a value of 20.

This constant makes the code more maintainable, allowing us to change the array size in a single

place if needed.

We initialize a loop counter esi to 0 and push 0.0 onto the FPU stack using fldz. This will be our accumulator for the sum of the array elements.

The loop counter ecx is set to the value of ARRAY\_SIZE, which determines the number of elements in the array.

*We enter a Loop Labeled as L1, where we perform the following operations for each element of the array:*

Load the current array element into the FPU stack using fld sngArray[esi].

Add the value in the FPU stack to the accumulator (ST(0)) using fadd. This operation also pops the top of the stack.

Move to the next element by adding the size of a REAL8 (8 bytes) to esi using add esi, TYPE REAL8.

The loop continues until all elements of the array have been processed.

After the loop, we call WriteFloat to display the sum of the array elements in ST(0).

## **CALCULATING THE SUM OF SQUARES OF TWO NUMBERS**

In this code, we are calculating the sum of the square roots of two numbers, valA and valB. Here are the steps involved:

We define two real numbers, valA and valB, which have the values 25.0 and 36.0, respectively.

```
535 .data
536     valA REAL8 25.0
537     valB REAL8 36.0
538
539 .code
540     fld valA      ; Load valA onto the FPU stack
541     fsqrt        ; Replace ST(0) with the square root of valA
542     fld valB      ; Load valB onto the FPU stack
543     fsqrt        ; Replace ST(0) with the square root of valB
544     fadd         ; Add the two square roots and leave the result in ST(0)
```

We use the FPU instructions to perform the calculation:

**fld valA:** This instruction loads the value of valA onto the FPU stack, making it the top element (ST(0)) on the stack.

**fsqrt:** This instruction replaces the value in ST(0) with its square root, so now ST(0) contains the square root of valA.

**fld valB:** We load the value of valB onto the FPU stack, replacing the previous value in ST(0).

**fsqrt:** This instruction calculates the square root of the new value in ST(0), which is the square root of valB.

**fadd:** Finally, we add the two square roots together, and the result is left in ST(0). This means ST(0) now contains the sum of the square roots of valA and valB.

ST(0) now contains the sum of the square roots of valA and valB.

This code efficiently calculates the sum of the square roots of valA and valB, utilizing the FPU instructions to load, compute square roots, and add the results. The final sum is stored in ST(0).

## CALCULATING THE DOT PRODUCT OF TWO PAIRS OF NUMBERS

Here, you have a code snippet that calculates the dot product of two pairs of numbers from an array. The input data contains two pairs of numbers stored in the "array" variable.

```
550 .data
551     array REAL4 6.0, 2.0, 4.5, 3.2
552
553 .code
554     fld dword ptr [array]      ; Load the first number of the first pair
555     fmul dword ptr [array+4]   ; Multiply with the second number of the first pair
556     fld dword ptr [array+8]    ; Load the first number of the second pair
557     fmul dword ptr [array+12]   ; Multiply with the second number of the second pair
558     fadd                      ; Add the two products in ST(0)
```

**fld array:** This instruction loads the first number from the "array" into ST(0), which is the top of the FPU stack.

**fmul [array+4]:** Here, we multiply the value at the memory location array+4 with the value in ST(0). This corresponds to multiplying the first pair ( $6.0 * 2.0$ ) and leaving the result in ST(0).

**fld [array+8]:** We load the second number from the second pair (4.5) into ST(0).

**fmul [array+12]:** This instruction multiplies the value at memory location array+12 (which is 3.2) with the value in ST(0). This is equivalent to multiplying the second pair ( $4.5 * 3.2$ ) and leaves the result in ST(0).

**fadd:** Finally, we add the two results in ST(0), which represents the dot product  $(6.0 * 2.0) + (4.5 * 3.2)$ .

## ***Mixed-Mode Arithmetic***

The passage below describes mixed-mode arithmetic in assembly language.

Mixed-mode arithmetic is arithmetic involving both integers and reals.

The Intel instruction set provides instructions that promote integers to reals and load the values onto the floating-point stack.

To perform mixed-mode arithmetic in assembly language, you must first use an instruction to promote the integer to a real.

For example, the following instruction promotes the integer in the N register to a real and loads it onto the floating-point stack:

**fld N**

Once you have finished performing arithmetic operations on the real, you can use an instruction to store it back to memory as an integer.

For example, the following instruction stores the real in the ST(0) register to the Z integer variable:

**fist Z**

It is important to note that the FIST instruction rounds the real to the nearest integer. If you need to truncate the real to an integer, you can use the following code:

```
575 fstcw ctrlWord      ; Store the FPU control word in ctrlWord
576 or ctrlWord, 110000000000b ; Set RC (Rounding Control) to truncate
577 fldcw ctrlWord      ; Load the modified control word to change the rounding mode
578
579 ; Perform arithmetic operations here using the modified rounding mode
580 fild N              ; Load the integer N into ST(0)
581 fadd X              ; Add the real X to ST(0)
582 fist Z              ; Store the truncated result in integer Z
583
584 fstcw ctrlWord      ; Store the FPU control word in ctrlWord again
585 and ctrlWord, 00111111111b ; Reset the rounding mode to default (round to nearest)
586 fldcw ctrlWord      ; Load the control word to restore the default rounding mode
```

In this code, we first store the FPU control word in the variable `ctrlWord`.

Then, we modify the **RC field** in `ctrlWord` to set the **rounding mode to truncate**.

After changing the **rounding mode**, we perform the desired arithmetic operations.

After completing the arithmetic operations, we store the FPU control word again and reset the rounding mode to the default setting (round to nearest) by using a bitwise AND operation to clear the bits that control the rounding mode.

Finally, we load the modified control word to ensure the default rounding mode is restored.

### Example 1: Adding an Integer to a Double

In this example, you're adding an integer (`N`) to a double (`X`) and storing the result in a double (`Z`).

The C++ code automatically promotes the integer to a real before performing the addition. The equivalent assembly code for this operation is as follows:

```
590 .data
591     N SDWORD 20
592     X REAL8 3.5
593     Z REAL8 ?
594
595 .code
596     fild N      ; Load the integer into ST(0)
597     fadd X      ; Add the memory value (X) to ST(0)
598     fstp Z      ; Store ST(0) to memory (Z)
```

This code loads the integer N into ST(0), adds the real value X, and stores the result in the real Z.

### Example 2: Promoting and Truncating

In this example, you're promoting an integer (N) to a double and evaluating a real expression involving N and X.

The result is stored in an integer (Z). C++ typically performs the conversion automatically, but in assembly, you use FIST to convert the result to an integer. Here's the code:

```
607 fild N      ; Load the integer into ST(0)
608 fadd X      ; Add the real X to ST(0)
609 fist Z      ; Store ST(0) to the integer Z
```

This code loads the integer N into ST(0), adds the real X, and then stores the truncated result in

the integer Z.

### Changing the Rounding Mode:

The FPU control word's RC field allows you to specify the rounding mode. You can use the **FSTCW instruction to store the control word in a variable**, **modify the RC field** to change the rounding mode (e.g., truncate or round to nearest), and **then use the FLDCW instruction to load the modified control word back into the FPU**. This allows you to control how rounding is performed during floating-point operations.

### Example 3:

```
620 INCLUDE Irvine32.inc
621 .data
622     N           SDWORD 20      ; Integer value
623     X           REAL8 3.5    ; Double-precision real value
624     Z           REAL8 ?      ; Result stored as a double-precision real
625     ctrlWord    WORD 0       ; FPU control word
626
627 .code
628     main PROC
629         ; Initialize FPU
630         finit
631
632         ; Store the FPU control word
633         fstcw ctrlWord
634         ; Modify the control word to set the rounding mode to truncate
635         or ctrlWord, 110000000000b
636         fldcw ctrlWord
637         ; Perform mixed-mode arithmetic
638         fild N          ; Load integer N into ST(0)
639         fadd X          ; Add real X to ST(0)
640         fstp Z          ; Store the result in Z
641         ; Display the result
642         mov edx, OFFSET Z
643         call WriteFloat
644         ; Reset the rounding mode to default (round to nearest)
645         and ctrlWord, 001111111111b
646         fldcw ctrlWord
647         exit
648     main ENDP
649 END main
```

## Initialization:

**INCLUDE Irvine32.inc:** This includes the Irvine32 library, which provides various I/O and utility functions for assembly language programming.

This section is used for defining data variables.

**N SDWORD 20:** Declares a signed doubleword (32-bit) integer variable named N with an initial value of 20.

**X REAL8 3.5:** Declares an 8-byte (64-bit) double-precision real variable named X with an initial value of 3.5.

**Z REAL8 ?:** Declares another double-precision real variable named Z without an initial value. It will store the result of the computation.

**ctrlWord WORD 0:** Declares a 16-bit variable named ctrlWord to store the FPU control word.

## FPU Initialization:

**finit:** Initializes the FPU (Floating-Point Unit), ensuring a clean start for FPU operations.

Storing the FPU Control Word:

**fstcw ctrlWord:** Stores the current FPU control word in the variable ctrlWord. This allows us to modify the control word without affecting the system's default FPU settings.

Modifying the Rounding Mode:

**or ctrlWord, 110000000000b:** This binary OR operation modifies bits 10 and 11 of the control word to set the rounding control (RC) to truncate. Truncate mode discards the fractional part when converting real numbers to integers.

#### Mixed-Mode Arithmetic:

**fild N:** Loads the integer N into the FPU stack, promoting it to a double-precision real value in ST(0).

**fadd X:** Adds the real X to the value in ST(0), resulting in a mixed-mode addition. The result remains in ST(0).

**fstp Z:** Stores the result from ST(0) into the double-precision real variable Z.

#### Displaying the Result:

**mov edx, OFFSET Z:** Prepares the address of the Z variable for displaying the result.

**call WriteFloat:** Calls the WriteFloat procedure to display the result stored in Z.

#### Resetting the Rounding Mode:

**and ctrlWord, 001111111111b:** This binary AND operation resets bits 10 and 11 of the control word to their default values, effectively setting the rounding mode back to round to nearest even.

#### Program Exit:

**exit:** Exits the program.

In summary, this program demonstrates the following concepts:

Changing the FPU control word to set the rounding mode for FPU operations.

Performing mixed-mode arithmetic by adding an integer and a double-precision real value using FPU instructions.

Controlling the rounding mode to ensure the desired behavior for the arithmetic operations.

The program calculates the result of mixed-mode arithmetic and displays it, considering the specified rounding mode, and then resets the rounding mode to its default state before exiting.

## ***UnMasking Floating Point Exceptions***

The passage below describes how to mask and unmask floating-point exceptions in assembly language.

Floating-point exceptions are masked by default, which means that when a floating-point exception is generated, the processor assigns a default value to the result and continues executing code.

To unmask a floating-point exception, you must clear the appropriate bit in the FPU control word. The following code unmasks the divide by zero exception:

```
652 .data  
653     ctrlWord WORD ?  
654  
655 .code  
656     ; Store the current FPU control word in a 16-bit variable (ctrlWord).  
657     fstcw ctrlWord  
658  
659     ; Clear bit 2 (Divide by zero exception mask) in the ctrlWord.  
660     and ctrlWord, 111111111111011b  
661  
662     ; Load the modified control word back into the FPU.  
663     fldcw ctrlWord
```

This code stores the current FPU control word in `ctrlWord`, then clears the appropriate bit (bit 2) to unmask the "Divide by zero" exception.

Finally, it loads the modified control word back into the FPU, ensuring that the exception is unmasked for subsequent operations.

Once the divide by zero exception is unmasked, the processor will try to execute an appropriate exception handler.

If no exception handler is installed, the processor will display an error message and terminate the program.

Here is an example of code that divides by zero and generates an unmasked exception:

```
667 ; Code that intentionally divides by zero, generating an unmasked exception  
668 fild val1      ; Load a value into ST(0)  
669 fdiv val2      ; Attempt to divide by zero (unmasked exception)  
670 fst val2      ; Store the result (won't be reached due to exception)
```

This code snippet intentionally divides by zero, which will generate an unmasked exception. Here's how it works step by step:

**fild val1:** Loads a value from the memory location `val1` into the FPU's top stack register `ST(0)`.

**fdiv val2:** Attempts to divide the value in `ST(0)` by the value stored in `val2`. Since the divisor is zero, this operation triggers a "Divide by zero" exception, which is unmasked as per the previous instructions.

**fst val2:** This instruction is intended to store the result back into the memory location `val2`. However, it won't be reached due to the unmasked exception generated in the previous step.

The program will trigger a system dialog indicating "Floating-Point Division by Zero," giving the user the option to retry or terminate the program.

## Exceptions.exe - Application Error



The exception Floating-point division by zero.  
(0xc000000e) occurred in the application at location 0x0040102e.

Click on OK to terminate the program  
Click on CANCEL to debug the program

OK

CANCEL

**Q: Write an instruction that loads a duplicate of ST(0) onto the FPU stack.**

**A:** You can load a duplicate of ST(0) onto the FPU stack with the fild ST(0) instruction.

**Q: If ST(0) is positioned at absolute register R6 in the register stack, what is the position of ST(2)?**

**A:** When ST(0) is at absolute register R6, ST(2) is positioned at absolute register R4 in the register stack.

**Q: Name at least three FPU special-purpose registers.**

**A:** Three FPU special-purpose registers are FSW (Floating-Point Status Word), FSTP (Floating-Point Control), and FTW (Floating-Point Tag Word).

**Q: When the second letter of a floating-point instruction is B, what type of operand is indicated?**

**A:** When the second letter of a floating-point instruction is B, it indicates that the instruction

operates on BCD (Binary Coded Decimal) operands.

**Q: Which floating-point instructions accept immediate operands?**

A: Some floating-point instructions that accept immediate operands (constants) include fld1 (load +1.0), fldz (load +0.0), fldpi (load pi), fldl2t (load the base-2 logarithm of 10), and fldl2e (load the base-2 logarithm of e).

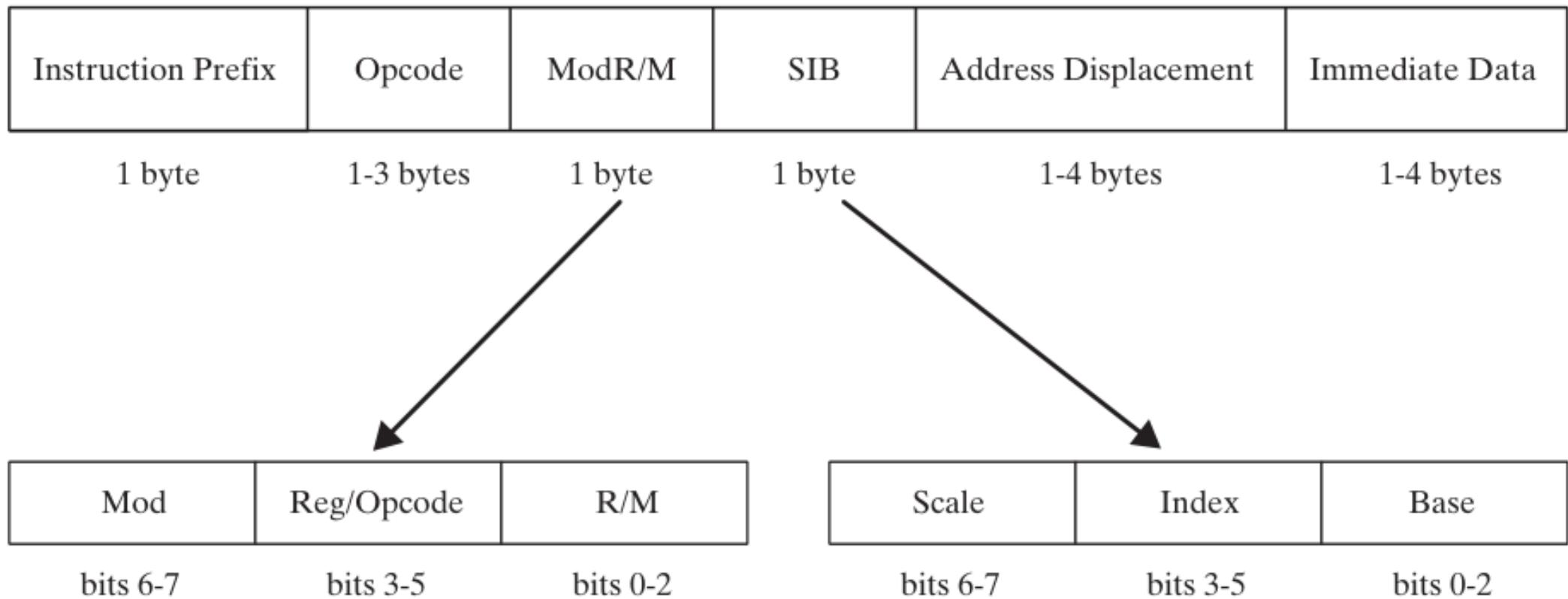
## **x86 Instruction Format**

The passage you sent describes the general x86 machine instruction format and its components.

The general x86 machine instruction format contains the following fields:

- **Instruction prefix byte:** Overrides default operand sizes.
- **Opcode (operation code):** Identifies a specific variant of an instruction.
- **Mod R/M field:** Identifies the addressing mode and operands.
- **Scale index byte (SIB):** Used to calculate offsets of array indexes.
- **Address displacement field:** Holds an operand's offset, or it can be added to base and index registers in addressing modes such as base-displacement or base-index-displacement.
- **Immediate data field:** Holds constant operands.

## x86 Instruction Format.



### ***Instruction prefix byte***

The instruction prefix byte is a single byte that can be used to override the default operand sizes. For example, the REX prefix byte can be used to override the default operand size from 16 bits to 32 bits.

### ***Opcode***

The opcode is a single byte or multiple bytes that identifies a specific variant of an instruction. For example, the opcode for the ADD instruction is 01.

### ***Mod R/M field***

The Mod R/M field is a single byte that identifies the addressing mode and operands. The first two bits of the Mod R/M field indicate the addressing mode, and the remaining bits indicate the operands.

### ***Scale index byte (SIB)***

The scale index byte (SIB) is a single byte that is used to calculate offsets of array indexes. The SIB byte is used in conjunction with the base and index registers to calculate the effective address of an operand.

### ***Address displacement field***

The address displacement field is a single byte or multiple bytes that holds an operand's offset, or it can be added to base and index registers in addressing modes such as base-displacement or base-index-displacement.

### ***Immediate data field***

The immediate data field is a single byte or multiple bytes that holds constant operands. Immediate data operands are used in instructions such as MOV and ADD.

**NB:** Not all instructions contain all of the fields described above. For example, the ADD instruction only contains the opcode and Mod R/M field.

The x86 machine instruction format is a complex topic, but it is important to understand the basics of the format in order to write assembly language code.

## MOD FIELD

The MOD field values shown in the table below are used to specify the addressing mode for the operand in the Mod R/M field. The MOD field is two bits wide, and the possible values are:

MOD value	Addressing mode
00	Register direct
01	Register indirect with 8-bit displacement
10	Register indirect with 16-bit displacement
11	R/M field contains the operand address

In the context of x86 assembly language instruction encoding, these are the meanings of the Mod (Mode) values along with their corresponding Displacement (DISP) representations:

**Mod 00: DISP = 0**

This mode implies that the displacement (address offset) is 0.

Disp-low and disp-high bytes are absent in the encoding, unless the R/M (Register/Memory) field equals 110, in which case the displacement is 32 bits long.

#### **Mod 01: DISP = disp-low, sign-extended to 16 bits; disp-high is absent**

In this mode, the displacement is included and consists of the disp-low byte sign-extended to 16 bits.

The disp-high byte is absent.

#### **Mod 10: DISP = disp-high and disp-low are used**

In this mode, both the disp-high and disp-low bytes are used to form the displacement.

The full 32-bit displacement is present.

#### **Mod 11: R/M field contains a register number**

In this mode, the R/M field does not indicate memory addressing but instead contains a register number.

No displacement is used in this case.

These modes are used to specify how memory operands are addressed and whether or not displacement values are included in the instruction encoding. The specifics of how these modes are used can vary depending on the particular instruction and addressing needs.

#### **If mod field is 11:**

If the MOD field is 11, then the R/M field contains the address of the operand. Otherwise, the R/M

field contains a register code, and the operand address is calculated using the following formula:

`operand_address = [register] + displacement`

where [register] is the value of the register specified by the R/M field, and displacement is the displacement value specified in the instruction.

Here are some examples of how the MOD field is used to specify the addressing mode for the operand in the Mod R/M field:

Instruction	MOD field	R/M field	Operand address
MOV EAX, EAX	00	00	EAX register
MOV EAX, [EBX]	00	03	The memory location at the address in the EBX register
MOV EAX, [EBX + 10]	01	03	The memory location at the address in the EBX register plus 10 bytes
MOV EAX, [EBX + ESI * 4]	10	03	The memory location at the address in the EBX register plus the value of the ESI register multiplied by 4 bytes

Let's continue...

R/M	Effective Address
000	[ BX + SI ] + D16 <sup>a</sup>
001	[ BX + DI ] + D16
010	[ BP + SI ] + D16
011	[ BP + DI ] + D16
100	[ SI ] + D16
101	[ DI ] + D16
110	[ BP ] + D16
111	[ BX ] + D16

<sup>a</sup>D16 indicates a 16-bit displacement.

I was using the table above, but it seems this is the new table:

R/M value	Effective Address
000	$[BX] + D16$
001	$[BX + SI] + D16$
010	$[BP + SI] + D16$
011	$[BP + DI] + D16$
100	$[SI] + D16$
101	$[DI] + D16$
110	$[BP] + D16$
111	$[BX] + D16$

The table on 16-bit R/M field values (for MOD = 10) shows the values of the R/M field for each mode. The R/M field is a single byte, and the first two bits of the R/M field indicate the addressing mode, and the remaining bits indicate the operands.

Here is an explanation of each row in the table:

The D16 in the table refers to a 16-bit displacement. The displacement is added to the base register or index register (or both) to calculate the effective address of the operand.

For example, the instruction **MOV EAX, [BX + 10]** has an R/M value of 000. This means that the operand address is calculated by adding the 16-bit displacement 10 to the BX register.

The table also shows that the R/M value 111 is the same as the R/M value 000. This is because the R/M field is only 3 bits wide, so there are only 8 possible values. The R/M value 111 is used to indicate that the base register is BX.

The 16-bit R/M field table is a useful reference for understanding how addressing modes are used in assembly language instructions.

## **SINGLE BYTE INSTRUCTIONS**

The simplest type of instruction is one with either no operand or an implied operand. Such instructions require only the opcode field, the value of which is predetermined by the processor's instruction set.

Table below lists a few common single-byte instructions. It might appear that the INC DX instruction slipped into the table by mistake, but the designers of the instruction set decided to supply unique opcodes for certain commonly used instructions. As a consequence, register increments are optimized for code size and execution speed.

Instruction	Opcode
AAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

The table of single-byte instructions that you provided lists the following instructions:

- **AAA:** ASCII adjust for addition
- **AAS:** ASCII adjust for subtraction
- **CBW:** Convert byte to word
- **LODSB:** Load byte from string
- **XLAT:** Translate
- **INC DX:** Increment DX

The **AAA** and **AAS** instructions are used to adjust the carry flag after performing addition or subtraction operations on ASCII characters.

The **CBW instruction** converts a byte to a word by filling the upper 8 bits of the word with the sign bit of the byte.

The **LODSB instruction** loads a byte from the current position in the string pointer (SI).

The **XLAT instruction** translates a byte using the ASCII translation table in memory.

The **INC DX instruction** increments the DX register by one.

The **INC DX instruction** is included in the table of single-byte instructions because it is a commonly used instruction that is optimized for code size and execution speed.

The designers of the instruction set decided to supply unique opcodes for certain commonly used instructions, such as INC DX, in order to improve the performance of programs that use these instructions frequently.

Here is an example of how the INC DX instruction can be used:

```
687 MOV DX, 10  
688 INC DX
```

This code will increment the DX register by one, so the value of DX will be 11 after the code is executed.

Single-byte instructions are the simplest type of instruction, but they can be used to perform a variety of operations. By understanding how to use single-byte instructions, you can write efficient and effective assembly language code.

## **MOV IMMEDIATE TO REGISTER**

The passage you sent describes how to encode MOV instructions that move immediate values to

registers in x86 assembly language.

The encoding format for a MOV instruction that moves an immediate word into a register is:

**B8 + rw dw**

where:

**B8** is the opcode for moving an immediate value to a register.

**rw** is the register code for the destination register.

**dw** is the immediate word operand, low byte first.

To encode a MOV immediate instruction, you must first determine the register code for the destination register. The register codes are listed in the following table:

<b>Register</b>	<b>Code</b>
AX/AL	0
CX/CL	1
DX/DL	2
BX/BL	3
SP/AH	4
BP/CH	5
SI/DH	6
DI/BH	7

Once you have determined the register code, you can encode the MOV immediate instruction as follows:

- Add the register code to B8 to get the opcode.
- Append the immediate operand to the opcode, low byte first.

For example, the following instruction moves the immediate value 1 to the AX register:

**MOV AX, 1**

The register code for AX is 0, so the opcode for this instruction is  $B8 + 0 = B8$ .

The immediate operand is 1, so the machine code for this instruction is B8 01 00.

The machine instruction is B8 01 00 (in hexadecimal).

#### ***Encoding Steps:***

The opcode for moving an immediate value to a 16-bit register is B8.

The register number for AX is 0, so 0 is added to B8.

The immediate operand (0001) is appended to the instruction in little-endian order (01, 00).

Another example is the following instruction, which moves the immediate value 1234h to the BX register:

#### **MOV BX, 1234h**

The register code for BX is 3, so the opcode for this instruction is B8 + 3 = BB.

The immediate operand bytes are 34 12, so the machine code for this instruction is BB 34 12.

The machine instruction is BB 34 12.

#### ***Encoding Steps:***

The opcode for moving an immediate value to a 16-bit register is B8.

The register number for BX is 3, so add 3 to B8, producing opcode BB.

The immediate operand bytes are 34 12.

You can practice encoding MOV immediate instructions by hand to improve your skills.

Once you have encoded an instruction, you can check your work by inspecting the code generated by MASM in a **source listing file**.

### **PUSH CX:**

The machine instruction is 51.

#### *Encoding Steps:*

The opcode for PUSH with a 16-bit register operand is 50.

The register number for CX is 1, so add 1 to 50, producing opcode 51.

## **REGISTER MODE INSTRUCTIONS**

If you want to understand this subtopic better, check the last line of this subtopic, go to that reference.

The passage you sent describes how to encode MOV instructions that use register operands in x86 assembly language.

The encoding format for a MOV instruction that moves a value from one register to another is

89/r reg

where:

- **89** is the opcode for moving a value from one register to another.
- **/r** indicates that a Mod R/M byte follows the opcode.
- **reg** is the register code for the destination register.

The Mod R/M byte contains a 3-bit identifier for each register operand.

The Mod R/M byte is made up of three fields (mod, reg, and r/m). A Mod R/M value of D8, for example, contains the following fields:

mod	reg	r/m
11	011	000

The **Mod field is 11**, indicating that the r/m field contains a register number.

The **reg field is 011**, indicating that the source operand is the BX register.

The **r/m field is 000**, indicating that the destination operand is the AX register.

Therefore, the machine code **89 D8** is for the instruction **MOV AX, BX**, which moves the value in the BX register to the AX register.

The register codes are listed in the following table:

<b>R/M</b>	<b>Register</b>	<b>R/M</b>	<b>Register</b>
000	AX or AL	100	SP or AH
001	CX or CL	101	BP or CH
010	DX or DL	110	SI or DH
011	BX or BL	111	DI or BH

For example, the following instruction moves the value in the BX register to the AX register:

**MOV AX, BX**

The register code for BX is 3 and the register code for AX is 0, so the machine code for this instruction is 89 D8.

Another example is the following instruction, which moves the value in the CL register to the AL register:

**MOV AL, CL**

The register code for CL is 1 and the register code for AL is 0, so the machine code for this instruction is 89 C8.

## Sample MOV Instruction Encodings, Register Operands.

Instruction	Opcode	mod	reg	r/m
mov ax,dx	8B	11	000	010
mov al,dl	8A	11	000	010
mov cx,dx	8B	11	001	010
mov cl,dl	8A	11	001	010

You can practice encoding MOV instructions that use register operands by hand to improve your skills. Once you have encoded an instruction, you can check your work by inspecting the code generated by MASM in a source listing file.

=====

*To calculate the value 89 D8 from the R/M fields, you need to follow these steps:*

Identify the addressing mode. The Mod field is 11, which indicates that the addressing mode is register direct.

Identify the source operand. The reg field is 011, which indicates that the source operand is the BX register.

Identify the destination operand. The r/m field is 000, which indicates that the destination operand is the AX register.

Encode the opcode. The opcode for the MOV instruction is 89.

Encode the Mod R/M byte. The Mod R/M byte is encoded as follows:

- **Mod field:** 11 (register direct)
- **reg field:** 011 (BX register)
- **r/m field:** 000 (AX register)

Therefore, the machine code for the instruction MOV AX, BX is 89 D8.

Here is a table that summarizes the encoding of the MOV instruction:

Opcode	Mod R/M byte	Instruction
89	00 /r	MOV register, register
89	01 /r	MOV register, memory
89	10 /r	MOV memory, register
89	11 /r	MOV memory, memory

The reg field and r/m field of the Mod R/M byte identify the source and destination operands, respectively. The Mod field indicates the addressing mode for the operand.

## *So, this only applies to mov instruction?*

Yes, the encoding scheme you described only applies to the MOV instruction. Other instructions may have different encoding schemes.

For example, the ADD instruction has the following encoding scheme:

Opcode	Mod R/M byte	Instruction
01	00 /r	ADD register, register
01	01/r	ADD register, memory
01	10 /r	ADD memory, register
01	11 /r	ADD memory, memory

The ADD instruction is similar to the MOV instruction, but it adds the source operand to the destination operand instead of moving the source operand to the destination operand.

Different instructions have different encoding schemes because they perform different operations. The encoding scheme for each instruction is specified in the [Intel instruction set reference](#).

# ***Processor Operand-Size Prefix***

The **operand-size prefix (66h)** is used to override the default segment attribute for the instruction it modifies.

This was necessary because the 8088/8086 instruction set used almost all 256 possible opcodes to handle instructions using 8- and 16-bit operands.

When Intel introduced 32-bit processors, they needed to find a way to invent new opcodes to handle 32-bit operands, yet retain compatibility with older processors.

For programs targeting 16-bit processors, a prefix byte was added to any instruction that used 32-bit operands.

For programs targeting 32-bit processors, 32-bit operands were the default, so a prefix byte was added to any instruction using 16-bit operands. Eight-bit operands need no prefix.

*Here is an example of how to use the operand-size prefix in 16-bit mode:*

```
715 .model small
716 .286
717 .stack 100h
718 .code
719 main PROC
    mov ax, dx ; 8B C2
    mov al, dl ; 8A C2
722 main ENDP
723 END main
```

The `mov ax,dx` instruction uses 16-bit operands, so it does not need a prefix byte.

The `mov al,dl` instruction uses 8-bit operands, so it also does not need a prefix byte.

Here is an example of how to use the operand-size prefix in 32-bit mode:

```
726 .model small
727 .386
728 .stack 100h
729 .code
730 main PROC
    mov ax, dx ; 66 8B C2
    mov al, dl ; 8A C2
733 main ENDP
734 END main
```

The `mov ax,dx` instruction now needs a prefix byte to indicate that it is using 16-bit operands.

The `mov al,dl` instruction still does not need a prefix byte because it is using 8-bit operands.

The operand-size prefix is a powerful tool that allows programmers to control the operand size of instructions. This can be useful for improving performance or for ensuring compatibility with older processors.

## **MEMORY MODE INSTRUCTIONS**

In x86 assembly language, the Memory Mode instructions are used to access memory operands with various addressing modes.

These instructions involve loading or storing data between registers and memory locations, with different combinations of registers and memory addresses.

The Mod field in the Mod R/M byte specifies the addressing mode for the memory operand.

*Here are some key points about Mod Memory Mode instructions:*

**Mod R/M Byte:** These instructions rely on the Mod R/M byte, which is a part of the instruction encoding, to specify the addressing mode and operands. The Mod field (bits 6-7) in the Mod R/M byte defines the addressing mode.

**Mod Field:** The Mod field can take on various values:

- **00:** Register-indirect addressing mode. The memory operand is accessed via a register.
- **01:** Displacement-only addressing mode. An immediate value (displacement) is added to the base register.
- **10:** SIB (Scale-Index-Base) addressing mode. This is typically used for complex addressing calculations.
- **11:** Register mode. The operands are registers, not memory.
- **Registers and Memory:** Mod Memory Mode instructions can involve different combinations of registers and memory locations, making them versatile for reading from or writing to memory.

#### ***Code Explanation:***

Now, let's go over the code provided in the earlier example:

```
770 main PROC  
771     mov ax, [si]          ; Load AX with the value at memory address pointed to by SI  
772     ; Mod 00, R/M 101 (SI is used as the offset address)  
773  
774     mov [si], al          ; Store the value in AL at the memory address pointed to by SI  
775     ; Mod 00, R/M 101 (SI is used as the offset address)  
776  
777     add bx, 10h           ; Add 10h to BX  
778     mov bx, [bx]          ; Load BX with the value at memory address [BX + 10h]  
779     ; Mod 00, R/M 010 (Offset BX is used)  
780  
781     mov [bx + si], cx    ; Store the value in CX at memory address [BX + SI]  
782     ; Mod 00, R/M 110 (Offset BX+SI is used)
```

**mov ax, [si]:** This instruction loads the value from the memory address pointed to by SI into the AX register. In the Mod R/M byte, it uses Mod 00 (register-indirect addressing mode) and R/M 101 (SI is the offset address).

**mov [si], al:** This instruction stores the value in AL into the memory address pointed to by SI. It also uses Mod 00 and R/M 101, as it's effectively performing a register-indirect memory operation using SI as the offset.

**add bx, 10h:** This instruction adds 10h to the BX register.

**mov bx, [bx]:** This instruction loads the value from the memory address [BX + 10h] into the BX register. It uses Mod 00 (register-indirect addressing mode) and R/M 010 (BX is the base address with a displacement).

**mov [bx + si], cx:** This instruction stores the value in CX into the memory address [BX + SI]. It uses Mod 00 (register-indirect addressing mode) and R/M 110 (BX+SI is used as the offset).

These examples illustrate how to use Mod Memory Mode instructions to access memory operands using different addressing modes. The specific Mod R/M byte values indicate the addressing mode and operands used in each instruction.

---

### **REPEATING FOR SOME ADDITIONAL INFO:**

The Mod R/M byte is a powerful tool that allows x86 assembly language programmers to specify a wide variety of memory-addressing modes.

The Mod field of the Mod R/M byte indicates the group of addressing modes, and the R/M field identifies the specific addressing mode within the group.

The following table shows the Mod R/M bytes for Mod 00:

R/M	Effective address
000	Register
001	[Base register]
010	[Base register + Displacement]
011	[Base register + Index register]
100	[Base register + Index register + Displacement]
101	[SI]
110	[DI]
111	[Base register + Displacement + 8]

The following examples show how to use the Mod R/M byte to encode MOV instructions that use different memory-addressing modes:

```
740 .model small
741 .data
742     data1    dw 05h          ; Define a data item
743     data2    dw 0A0Bh        ; Define another data item
744
745 .code
746 main PROC
747     mov ax, [si]           ; Load AX with the value at memory address pointed to by SI
748     ; Mod 00, R/M 101 (SI is used as the offset address)
749
750     mov [si], al            ; Store the value in AL at the memory address pointed to by SI
751     ; Mod 00, R/M 101 (SI is used as the offset address)
752
753     add bx, 10h             ; Add 10h to BX
754     mov bx, [bx]             ; Load BX with the value at memory address [BX + 10h]
755     ; Mod 00, R/M 010 (Offset BX is used)
756
757     mov [bx + si], cx       ; Store the value in CX at memory address [BX + SI]
758     ; Mod 00, R/M 110 (Offset BX+SI is used)
759
760     ; Terminate the program
761     mov ah, 4Ch
762     int 21h
763
764 main ENDP
765 END main
```

mov ax, [si]:

This instruction loads the value from the memory address pointed to by SI into the AX register.

Mod 00 indicates register-indirect addressing mode, and R/M 101 signifies SI as the offset address.

The value at memory address [SI] is read into AX.

### **mov [si], al:**

This instruction stores the value in AL into the memory address pointed to by SI.

It uses Mod 00 (register-indirect addressing mode) and R/M 101 (SI is the offset address).

The content of AL is written to the memory location pointed to by SI.

### **add bx, 10h:**

BX is loaded with the immediate value 10h.

The add instruction uses an immediate value to add 10h to the value in BX.

### **mov bx, [bx]:**

This instruction loads the value from the memory address [BX + 10h] into the BX register.

Mod 00 signifies register-indirect addressing, and R/M 010 indicates the use of BX as the base address with a displacement of 10h.

BX now contains the value from memory address [BX + 10h].

**mov [bx + si], cx:**

This instruction stores the value in CX into the memory address [BX + SI].

Mod 00 indicates register-indirect addressing, and R/M 110 implies that both BX and SI are used as offsets.

The content of CX is written to the memory location at the address [BX + SI].

These examples illustrate how the Mod Memory Mode instructions enable the movement of data between registers and memory locations using different addressing modes.

The **Mod R/M byte**, as indicated in the comments, is key in specifying the addressing mode and operands for each instruction.

*Use these tables as references when hand-assembling MOV instructions. (For more details, refer to the Intel manuals.)*

## Partial List of Mod R/M Bytes (16-Bit Segments).

Byte:		AL	CL	DL	BL	AH	CH	DH	BH		
Word:		AX	CX	DX	BX	SP	BP	SI	DI		
Register ID:		000	001	010	011	100	101	110	111		
Mod	R/M	Mod R/M Value									Effective Address
00	000	00	08	10	18	20	28	30	38	[ BX + SI ]	
	001	01	09	11	19	21	29	31	39	[ BX + DI ]	
	010	02	0A	12	1A	22	2A	32	3A	[ BP + SI ]	
	011	03	0B	13	1B	23	2B	33	3B	[ BP + DI ]	
	100	04	0C	14	1C	24	2C	34	3C	[ SI ]	
	101	05	0D	15	1D	25	2D	35	3D	[ DI ]	
	110	06	0E	16	1E	26	2E	36	3E	16-bit displacement	
	111	07	0F	17	1F	27	2F	37	3F	[ BX ]	

Table 2:

## MOV Instruction Opcodes.

<b>Opcode</b>	<b>Instruction</b>	<b>Description</b>
88/r	MOV eb,rb	Move byte register into EA byte
89/r	MOV ew,rw	Move word register into EA word
8A/r	MOV rb,eb	Move EA byte into byte register
8B/r	MOV rw,ew	Move EA word into word register
8C/0	MOV ew,ES	Move ES into EA word
8C/1	MOV ew,CS	Move CS into EA word
8C/2	MOV ew,SS	Move SS into EA word
8C/3	MOV ew,DS	Move DS into EA word
8E/0	MOV ES,mw	Move memory word into ES
8E/0	MOV ES,rw	Move word register into ES
8E/2	MOV SS,mw	Move memory word into SS
8E/2	MOV SS,rw	Move register word into SS
8E/3	MOV DS,mw	Move memory word into DS
8E/3	MOV DS,rw	Move word register into DS
A0 dw	MOV AL,xb	Move byte variable (offset dw) into AL
A1 dw	MOV AX,xw	Move word variable (offset dw) into AX
A2 dw	MOV xb,AL	Move AL into byte variable (offset dw)

A3 dw	MOV xw,AX	Move AX into word register (offset dw)
B0 +rb db	MOV rb,db	Move immediate byte into byte register
B8 +rw dw	MOV rw,dw	Move immediate word into word register
C6 /0 db	MOV eb,db	Move immediate byte into EA byte
C7 /0 dw	MOV ew,dw	Move immediate word into EA word

Table 3:

## Key to Instruction Opcodes.

/n:	A Mod R/M byte follows the opcode, possibly followed by immediate and displacement fields. The digit n (0–7) is the value of the reg field of the Mod R/M byte.
/r:	A Mod R/M byte follows the opcode, possibly followed by immediate and displacement fields.
db:	An immediate byte operand follows the opcode and Mod R/M bytes.
dw:	An immediate word operand follows the opcode and Mod R/M bytes.
+rb:	A register code (0–7) for an 8-bit register, which is added to the preceding hexadecimal byte to form an 8-bit opcode.
+rw:	A register code (0–7) for a 16-bit register, which is added to the preceding hexadecimal byte to form an 8-bit opcode.

Table 4:

## Key to Instruction Operands.

db	A signed value between –128 and +127. If combined with a word operand, this value is sign-extended.
dw	An immediate word value that is an operand of the instruction.
eb	A byte-sized operand, either register or memory.
ew	A word-sized operand, either register or memory.
rb	An 8-bit register identified by the value (0–7).
rw	A 16-bit register identified by the value (0–7).
xb	A simple byte memory variable without a base or index register.
xw	A simple word memory variable without a base or index register.

Table 12-28 contains a few additional examples of MOV instructions that you can assemble by hand and compare to the machine code shown in the table. We assume that myWord begins at offset 0102h.

## Sample MOV Instructions, with Machine Code.

Instruction	Machine Code	Addressing Mode
mov ax,myWord	A1 02 01	direct (optimized for AX)
mov myWord,bx	89 1E 02 01	direct
mov [di],bx	89 1D	indexed
mov [bx+2],ax	89 47 02	base-disp
mov [bx+si],ax	89 00	base-indexed
mov word ptr [bx+di+2],1234h	C7 41 02 34 12	base-indexed-disp

## QUESTIONS

These questions are already answered, so just read:

**Provide the opcodes for the following MOV instructions.**

Instruction	Opcode
MOV AX, @DATA	A1
MOV DS, AX	8E D8
MOV AX, BX	89 D0
MOV BL, AL	88 C3
MOV AL, [SI]	8A 36
MOV MYBYTE, AL	88 44 01
MOV MYWORD, AX	89 84 01

Mod R/M bytes for MOV instructions:

Instruction	Mod R/M byte
MOV AX, @DATA	00 00 00
MOV DS, AX	8E D8
MOV DL, BL	00 10
MOV BL, [DI]	00 3E
MOV AX, [SI + 2]	04 36 02
MOV AX, ARRAY[SI]	00 46
MOV ARRAY[DI], AX	00 3C

## Explanation

The Mod R/M byte is a complex topic, but it is essential to understand how to use it in order to write efficient and effective x86 assembly language code. The following is a brief explanation of how to calculate the Mod R/M byte for each of the MOV instructions in the question:

**MOV AX, @DATA**

The @DATA operand is a symbolic address that represents the global variable DATA. The Mod R/M byte for this instruction is 00 00 00. This is because the DATA operand is a 32-bit absolute address, and the Mod R/M byte for 32-bit absolute addresses is 00 00 00.

#### **MOV DS, AX**

The DS register is a segment register that stores the base address of the data segment. The Mod R/M byte for this instruction is 8E D8. This is because the DS register is a special register, and the Mod R/M byte for special registers is 8E. The D8 in the Mod R/M byte indicates that the destination operand is the DS register.

#### **MOV AX, BX**

The AX and BX registers are general-purpose registers. The Mod R/M byte for this instruction is 00 10. This is because the BX register is register number 3, and the Mod R/M byte for register-to-register moves is 00. The 10 in the Mod R/M byte indicates that the source operand is the BX register.

#### **MOV BL, AL**

The BL and AL registers are general-purpose registers. The Mod R/M byte for this instruction is 00 10. This is because the AL register is register number 0, and the Mod R/M byte for register-to-register moves is 00. The 10 in the Mod R/M byte indicates that the source operand is the AL register.

#### **MOV AL, [SI]**

The AL register is a general-purpose register, and the [SI] operand is a memory operand. The Mod R/M byte for this instruction is 00 36. This is because the SI register is register number 6, and the Mod R/M byte for memory operands is 00. The 36 in the Mod R/M byte indicates that the base register

is the SI register.

**MOV MYBYTE, AL**

The MYBYTE operand is a global variable, and the AL register is a general-purpose register. The Mod R/M byte for this instruction is 88 44 01. This is because the MYBYTE operand is a memory operand, and the Mod R/M byte for memory operands is 00. The 44 in the Mod R/M byte indicates that the destination operand is the MYBYTE operand. The 01 in the Mod R/M byte indicates that the destination operand is a byte.

**MOV MYWORD, AX**

The MYWORD operand is a global variable, and the AX register is a general-purpose register. The Mod R/M byte for this instruction is 89 84 01. This is because the MYWORD operand is a memory operand, and the Mod

***Why I skipped the rest of the book.. and more stories***

This is the book!

# *Assembly Language*

FOR x86 PROCESSORS

*Seventh Edition*



*Kip Irvine*

## Reasons why did I decide to skip BIOS level programming?

- Not just BIOS level programming, but BIOS Level programming in ASSEMBLY. 💀💀 That's a cocktail for death!
- I heard BIOS level programming is like trying to read hieroglyphs from an ancient computer tomb. I'll stick to modern code, no thank you.
- My computer programming philosophy: "No BIOS level, no problem!" 🚀

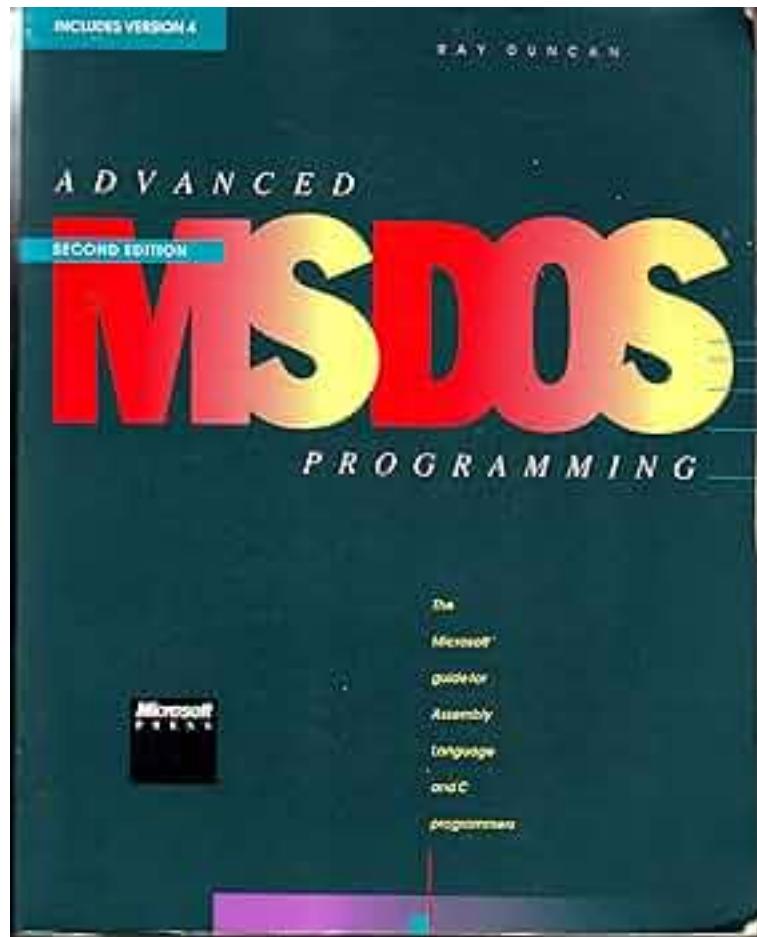
The rest of the topics are super advanced and I haven't really got a grasp of assembly to a level that I can deal with the following topics that remain:

**Assembly with C/C++:** I will use this book later on in life ... 😊

# **X86 Assembly Language and C Fundamentals**

**Joseph Cavanagh**

16-bit MS-DOS programming, I don't think I need that topic right now... or even in the future. It's a topic to learn for fun maybe...



Disk fundamentals, I will cover that topic when I learn about operating systems storage management, using this book...

# Operating System Concepts

TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE



## What about MS-DOS??

- Coz Advanced MS-DOS programmers never get lost; they just change directories! Getting lost is a necessity in life for me 
- Why did the advanced MS-DOS programmer bring a ladder to work? To reach the higher memory!
- Advanced MS-DOS programmers don't play hide and seek; they play "dir and find"!
- Why did the programmer use MS-DOS for his love letter? Because he wanted it to be "SYStematically" romantic!
- Advanced MS-DOS programmers never argue; they just "batch" their differences! patch .... 
- MS-DOS programmers tell the best bedtime stories: "Once upon a time in a directory far, far away..."

=====

Greetings, Future Code Conjurers,

As I prepare to take my leave, let's reminisce about the wild ride we've had in the enchanting realm of assembly programming and operating systems. It's been a journey filled with more 0s and 1s than a binary-themed disco party, but we've managed to decode the secrets of this digital universe. 

Within my treasure chest of meticulously updated notes and code, updated @ 2023, you'll find the keys to the kingdom of coding. It's like having a GPS for the labyrinth of algorithms and a magic wand to summon those elusive syscalls. 

Now, as you venture into the unpredictable landscape of real-world programming, remember that unlike the sterile environment of your IDE, reality is like a zoo. It's got bugs of all shapes and sizes, crashes that pop up like surprise parties, and sometimes, even blue screens, which are like the ultimate "You shall not pass" barriers. 🗡️🛡️

But fear not! With this repository of knowledge, you'll wield debugging powers that even Thor's hammer would envy. 🔧🔨 And should you ever feel stuck, just remember: Ctrl + Z is your magic "Undo" spell, and Google is your friendly wizard mentor.

As you continue this epic adventure, may your code compile smoother than butter on a hot pancake, and may your systems run as smoothly as a well-oiled robot doing the moonwalk. 🎈🕺💃

Now, as I make my exit, I can't help but grin at the thought of all the fantastic and, dare I say, "punny" code you'll create. Your journey is just beginning, and the tech universe is your playground. So go forth and code like it's the last line of defense against a robot uprising! 🌎💻🌐

Farewell, my fellow jesters of Java, sorcerers of C, and assembly alchemists. The tech world is your oyster; go forth and code like a wizard casting spells.

My fellow wizards and tech explorers, until we meet again in the binary dreamscape of ones and zeros, keep smiling, keep coding, and may the tech odds be ever in your favor. 😊🧙‍♂️🧙‍♀️✍️

---

***Crucial!! You'll use this all the time...***

- **BYTE (8 bits)**: Short form - **B**
  - **SBYTE (8 bits, signed)**: Short form - **SB**
  - **WORD (16 bits)**: Short form - **W**
  - **SWORD (16 bits, signed)**: Short form - **SW**
  - **DWORD (32 bits)**: Short form - **D**
  - **SDWORD (32 bits, signed)**: Short form - **SD**
  - **FWORD (48 bits)**: Short form - **FW**
  - **QWORD (64 bits)**: Short form - **Q**
  - **TBYTE (80 bits)**: Short form - **T**
  - **REAL4 (32-bit floating-point)**: Short form - **F**
  - **REAL8 (64-bit floating-point)**: Short form - **FF**
  - **REAL10 (80-bit floating-point)**: Short form - **FT**
- 

A data definition statement in assembly language is used to reserve memory for a variable and to specify its data type. The general syntax for a data definition statement is as follows:

**[label] directive value**

Where:

- **label** is an optional name for the variable.
- **directive** is the data type of the variable.
- **value** is the initial value of the variable.

For example, the following data definition statement reserves 4 bytes of memory for a variable named **count** and initializes it to the value 12345:

```
count DWORD 12345  
int count = 12345;
```

So, **label** is variable, **directive** is the datatype, and **value** is the value.

The following are some other examples of data definition statements in assembly language:

```
message DB "Hello, world!"  
age BYTE 25  
salary SDWORD 100000
```

-----

The **DUP operator takes two arguments: a count and a value**. The count is an integer expression that specifies the number of times to duplicate the value.