



Universidad Nacional Autónoma de México

Facultad de Ingeniería

Apuntes de
Estructura y programación de computadoras

Semestre 2015 – 2

Miguel Israel Barragán Ocampo

ESTRUCTURA Y PROGRAMACIÓN DE COMPUTADORAS

Método de Evaluación

♣ *Cuatro exámenes parciales*

En los cuales se evaluarán los siguientes aspectos:

- Teoría
- Programación (análisis y síntesis)

Se realizará un examen cada cuatro semanas.

♣ *Un examen final (NO HAY EXENTOS)*

Oral: Para alumnos que obtengan un promedio mayor o igual a SEIS.
Escrito: Para alumnos que no tengan calificación aprobatoria.

♣ *Tareas*

Son obligatorias y constituyen un requisito para tener derecho a examen parcial.

Contacto

fi_estudiantes@hotmail.com

Antecedentes

- ♣ Ser un lector
- ♣ Tener desarrollada una disciplina de estudio
- ♣ Haber adquirido las habilidades necesarias para la programación en cualquier lenguaje

Programa de Estudio

Materia del cuarto semestre con valor de 9 créditos y clave 1429.

A cursar en 4.5 horas por semana durante 16 semanas haciendo un total de 72 horas.

Asignatura obligatoria antecedente: Ninguna.

Asignatura obligatoria consecuente: Sistemas Operativos.

Objetivo: El alumno explicará los conceptos fundamentales de organización y programación de una computadora, que le permitan llevar a cabo el análisis, diseño y desarrollo de programas del sistema, mismos que facilitarán al usuario del equipo interactuar de una manera más eficiente con éste.

Contenido:

1. Estructura de la máquina
2. Presentación de un caso real
3. Ensambladores
4. Máquinas virtuales
5. Encadenadores y cargadores
6. Asignación de memoria
7. Programación de entrada/salida

Bibliografía:

1. ABEL, Peter. *"Lenguaje ensamblador y programación para Pc, IBM y compatibles"*. Ed. Pearson Educación, Prentice Hall
2. Merril. *"Assembly Language Programming 8086/8088, 80286 y 80486"*
3. *"Systems Programming"*. Mc. Graw Hill EU 1972
4. Brey, Barry B. *"Los microprocesadores Intel: 8086/8088, 80186, 80286, 80386 y 80486. Arquitectura, programación e interfaces."* Tercera Edición. Prentice-Hall Hispanoamericana, S.A. México. 1995.

Agradecimiento:

Quiero agradecer a **Martha Karen González Carvajal** por su generoso aporte a la revisión y corrección de la redacción y forma de estos apuntes para el semestre 2014 – I y el que no haya atentado en contra de la esencia de los mismos, aunque sé también les hace falta ya que siempre será necesario mejorarlos en esencia.

LENGUAJE ENSAMBLADOR

El lenguaje ensamblador NO es un ensamblador.

♣ **Desventajas:**

- Difícil de aprender
- Difícil de leer y entender
- Difícil de mantener
- Difícil de escribir

♣ **Ventajas:**

- Velocidad: es hasta diez veces más rápido que cualquier HLL (high-level programming language)
- Espacio
- Capacidad: permite implementar algoritmos que son difíciles o imposibles en HLLs
- Tener conocimiento del lenguaje ensamblador ayudará a escribir programas aun cuando se usen HLLs

SISTEMAS DE NUMERACIÓN

Los sistemas de cómputo no representan los valores numéricos usando el sistema decimal, éstos usan sistemas binarios usando complemento a dos.

Sistema decimal

$$123_d = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0)$$

$$123.456_d = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (4 \times 10^{-1}) + (5 \times 10^{-2}) + (6 \times 10^{-3})$$

Las máquinas usan normalmente 0 ó 5[V] para representar valores, a los cuales se les da el valor de 0 ó 1 (dígitos del sistema binario). El sistema binario trabaja como el sistema decimal con dos excepciones obvias:


- Sólo se usan los dígitos 0 y 1
- Su base es 2

 Investigar el concepto de BAUD (Bd)

Sistema Binario

$$101101_b = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$111.111_b = (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

 Descargar Ad-Aware Lavasoft

Sistema hexadecimal

El problema del sistema binario es que necesita muchos dígitos para representar un número cualquiera en comparación con cualquier otra base.

Ejemplo:

202_d sólo necesita tres dígitos en base 10 y ocho dígitos en base 2 $\rightarrow 11001010_b$.

El sistema hexadecimal presenta la ventaja de ser compacto, a diferencia del sistema binario; además, su conversión al sistema decimal es sencilla:

$$1234_h = (1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = 4660_d$$

Para realizar la conversión de base dos a base dieciséis, basta con sustituir cada dígito hexadecimal en su posición por su representación binaria.

Ejemplo:

$$1234_h \rightarrow 0001\ 0010\ 0011\ 0100_b$$

$$1_h \rightarrow 0001_b \quad 2_h \rightarrow 0010_b \quad 3_h \rightarrow 0011_b \quad 4_h \rightarrow 0100_b$$

4 bits \rightarrow 16 combinaciones diferentes

4 bits representan 16 dígitos

$$2^4 = 16$$

Tabla de Equivalencias 2-16		
Decimal	Binario	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

La tabla anterior tiene la intención de mostrar la conversión entre base 16 y base 2, la columna de base 10 es sólo una referencia.

Ejemplo:

$$\clubsuit \text{ ABCDEF12}_h \rightarrow 1010 \ 1011 \ 1100 \ 1101 \ 1110 \ 1111 \ 0001 \ 0010_b$$

<u>1010</u>	<u>1011</u>	<u>1100</u>	<u>1101</u>	<u>1110</u>	<u>1111</u>	<u>0001</u>	<u>0010</u>
A	B	C	D	E	F	1	2

$$\clubsuit \ 295_h \rightarrow 0010 \ 1001 \ 0101_b$$

<u>0010</u>	<u>1001</u>	<u>0101</u>
2	9	5

Debido a que el campo de un número en hexadecimal ocupa 4 bits exactamente para la representación de TODOS sus dígitos se puede realizar una conversión de manera directa.

ARITMÉTICA BINARIA

Las operaciones aritméticas y lógicas entre números binarios se realizan bit a bit.

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 1 &= 10 \\ 1 + 1 + 1 &= 11 \\ 1011 + 1100 &= 10111 \end{aligned}$$

Números negativos (complemento a dos)

Un valor negativo se expresa en notación de complemento a dos. Cabe mencionar que existen los complementos a otras bases. Para obtener el complemento a dos de un número binario se realizan dos pasos:

1. Al número propuesto le son intercambiados los unos por ceros y los ceros por unos (este paso se llama complementar el número o aplicarle un operador NOT)
2. Al número complementado se le suma 1.

Ejemplo:

Obtener el complemento a dos del siguiente número binario: $01 \ 0111 \ 0101_b$

$$\begin{array}{rcl} 10 \ 1000 \ 1010_b + 1_b & = & 10 \ 1000 \ 1011_b \quad ; \quad \begin{array}{r} 10 \ 1000 \ 1011_b \\ + \underline{01 \ 0111 \ 0101_b} \\ 100 \ 0000 \ 0000_b \end{array} \quad \left. \vphantom{\begin{array}{r} 10 \ 1000 \ 1011_b \\ + \underline{01 \ 0111 \ 0101_b} \\ 100 \ 0000 \ 0000_b \end{array}} \right\} \text{ Sumados deben dar cero} \end{array}$$

Nota: Si después de la suma hay un **1** de acarreo en la posición más a la izquierda, éste se elimina.

 Hacer la siguiente Resta Binaria con complemento a dos: $65_h - 42_h$

Sumas en hexadecimal

$$\begin{aligned} 6_h + 4_h &= A_h \\ 5_h + 8_h &= D_h \\ F_h + 1_h &= 10_h \\ 10_h + 30_h &= 40_h \\ FF_h + 1_h &= 100_h \end{aligned}$$

Clasificación de los valores en binario con base al número de dígitos que los conforman

X_0	bit = binary unit
$X_3 X_2 X_1 X_0$	nibble: conjunto de cuatro bits
$X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$	byte
$X_{15} X_{14} X_{13} X_{12} X_{11} X_{10} X_9 X_8 X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$	word
$X_{31} X_{30} X_{29} X_{28} X_{27} \dots X_6 X_5 X_4 X_3 X_2 X_1 X_0$	double word

Operaciones lógicas

Tablas de verdad

<table><tr><th>AND</th><th>0</th><th>1</th></tr><tr><th>0</th><td>0</td><td>0</td></tr><tr><th>1</th><td>0</td><td>1</td></tr></table>	AND	0	1	0	0	0	1	0	1	AND: Identificador de unos, verifica que la respuesta sea verdadera. Si el primer operando en 1 y el segundo es 1 el resultado es 1, de otra manera será 0.
AND	0	1								
0	0	0								
1	0	1								
<table><tr><th>OR</th><th>0</th><th>1</th></tr><tr><th>0</th><td>0</td><td>1</td></tr><tr><th>1</th><td>1</td><td>1</td></tr></table>	OR	0	1	0	0	1	1	1	1	OR: Identificador de ceros
OR	0	1								
0	0	1								
1	1	1								
<table><tr><th>NOR</th><th>0</th><th>1</th></tr><tr><th>0</th><td>1</td><td>0</td></tr><tr><th>1</th><td>0</td><td>0</td></tr></table>	NOR	0	1	0	1	0	1	0	0	NOR: NOR = NOT(OR)
NOR	0	1								
0	1	0								
1	0	0								
<table><tr><th>NAND</th><th>0</th><th>1</th></tr><tr><th>0</th><td>1</td><td>1</td></tr><tr><th>1</th><td>1</td><td>0</td></tr></table>	NAND	0	1	0	1	1	1	1	0	NAND: NAND = NOT(AND)
NAND	0	1								
0	1	1								
1	1	0								

<table><tr><td>XOR</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	XOR	0	1	0	0	1	1	1	0	XOR: Sirve para saber qué es diferente
XOR	0	1								
0	0	1								
1	1	0								
<table><tr><td>XNOR</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr></table>	XNOR	0	1	0	1	0	1	0	1	XNOR: Es verdadero para cuando son iguales
XNOR	0	1								
0	1	0								
1	0	1								
NOT 0 = 1 NOT 1 = 0	NOT: Negación									

CÓDIGO ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE)

Para uniformar la representación de caracteres, los fabricantes de microcomputadoras han adoptado el código ASCII, un código uniforme que facilita la transferencia de información entre los diferentes dispositivos de la computadora.

Está dividido en 4 grupos de 32 caracteres cada uno. Los primeros 32, (0_h al F_h) forman un set especial de caracteres no imprimibles llamados *caracteres de control*. Se le llaman así porque ejecutan operaciones de control, de despliegue e impresión. Ejemplo de éstos es el retorno de carro, el cual posiciona el cursor a la izquierda de la línea actual de caracteres, "line feed" (el cual mueve el cursor debajo de una línea, en el dispositivo de salida).

El segundo grupo de 32 caracteres del código ASCII comprende varios símbolos de puntuación, caracteres especiales y los dígitos numéricos.

El tercer grupo de 32 caracteres de código ASCII está reservado para el alfabeto en mayúsculas "A...Z" (41_h a 5A_h) los restantes son símbolos especiales.

El cuarto grupo y último están reservados para los caracteres del alfabeto en minúsculas y 5 símbolos especiales adicionales y otro carácter de control, las minúsculas van de "a...z" (61_h a 7A_h)

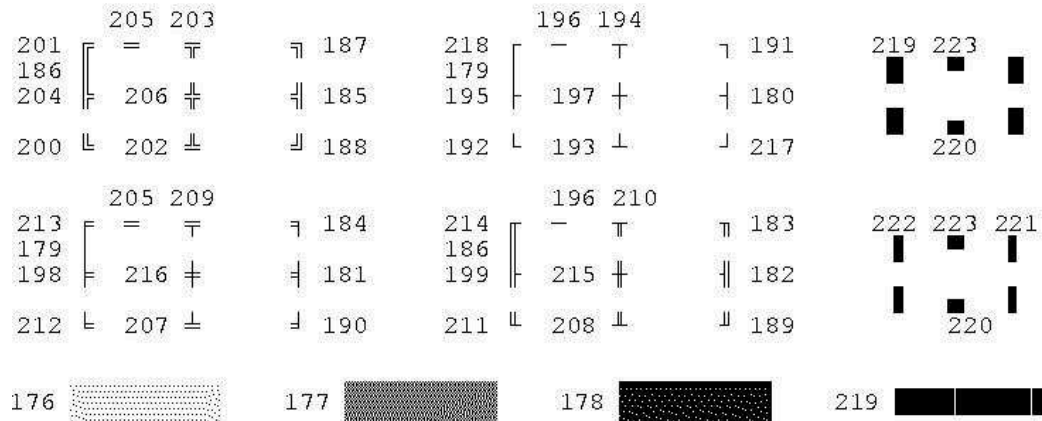
Existe el código ASCII extendido que ocupa 8 bits para su representación. Fue creado en 1963 por el Comité Estadounidense de Estándares o "ASA", este organismo cambio su nombre en 1969 por "Instituto Estadounidense de Estándares Nacionales" o "ANSI", como se lo conoce desde entonces. Este código nació a partir de reordenar y expandir el conjunto de símbolos y caracteres ya utilizados por ese entonces en telegrafía por la compañía Bell. En un primer momento sólo incluía las letras mayúsculas, pero en 1967 se agregaron las letras minúsculas y algunos caracteres de control, formando así lo que se conoce como US-ASCII, es decir los códigos del 0 al 127. Así con este conjunto de sólo 128 caracteres fue publicado en 1967 como estándar, conteniendo todos los necesarios para escribir en idioma inglés.

En 1986, se modificó el estándar para agregar nuevos caracteres latinos, necesarios para la escritura de textos en otros idiomas, como por ejemplo el español, así fue como se agregaron los caracteres que van del ASCII 128 al 255. Extended Binary Coded Decimal Interchange Code (EBCDIC) → competencia del ASCII.

Códigos ASCII (0-127)

Caracteres no imprimibles				Caracteres imprimibles											
Nombre	Dec	Hex	Car.	Dec	Hex	Car.	Dec	Hex	Car.	Dec	Hex	Car.	Dec	Hex	Car.
Nulo	0	00	NUL	32	20	Espacio	64	40	@	96	60	`			
Inicio de cabecera	1	01	SOH	33	21	!	65	41	A	97	61	a			
Inicio de texto	2	02	STX	34	22	"	66	42	B	98	62	b			
Fin de texto	3	03	ETX	35	23	#	67	43	C	99	63	c			
Fin de transmisión	4	04	EOT	36	24	\$	68	44	D	100	64	d			
Enquiry	5	05	ENQ	37	25	%	69	45	E	101	65	e			
Acknowledge	6	06	ACK	38	26	&	70	46	F	102	66	f			
Campanilla (beep)	7	07	BEL	39	27	'	71	47	G	103	67	g			
Backspace	8	08	BS	40	28	(72	48	H	104	68	h			
Tabulador horizontal	9	09	HT	41	29)	73	49	I	105	69	i			
Salto de línea	10	0A	LF	42	2A	*	74	4A	J	106	6A	j			
Tabulador vertical	11	0B	VT	43	2B	+	75	4B	K	107	6B	k			
Salto de página	12	0C	FF	44	2C	,	76	4C	L	108	6C	l			
Retorno de carro	13	0D	CR	45	2D	-	77	4D	M	109	6D	m			
Shift fuera	14	0E	SO	46	2E	.	78	4E	N	110	6E	n			
Shift dentro	15	0F	SI	47	2F	/	79	4F	O	111	6F	o			
Escape línea de datos	16	10	DLE	48	30	0	80	50	P	112	70	p			
Control dispositivo 1	17	11	DC1	49	31	1	81	51	Q	113	71	q			
Control dispositivo 2	18	12	DC2	50	32	2	82	52	R	114	72	r			
Control dispositivo 3	19	13	DC3	51	33	3	83	53	S	115	73	s			
Control dispositivo 4	20	14	DC4	52	34	4	84	54	T	116	74	t			
neg acknowledge	21	15	NAK	53	35	5	85	55	U	117	75	u			
Sincronismo	22	16	SYN	54	36	6	86	56	V	118	76	v			
Fin bloque transmitido	23	17	ETB	55	37	7	87	57	W	119	77	w			
Cancelar	24	18	CAN	56	38	8	88	58	X	120	78	x			
Fin medio	25	19	EM	57	39	9	89	59	Y	121	79	y			
Sustituto	26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z			
Escape	27	1B	ESC	59	3B	;	91	5B	[123	7B	{			
Separador archivos	28	1C	FS	60	3C	<	92	5C	\	124	7C				
Separador grupos	29	1D	GS	61	3D	=	93	5D]	125	7D	}			
Separador registros	30	1E	RS	62	3E	>	94	5E	^	126	7E	~			
Separador unidades	31	1F	US	63	3F	?	95	5F	_	127	7F	DEL			

Se anexa el siguiente compendio de caracteres que sirven para crear cuadros en modo texto.



A continuación el cuadro muestra los caracteres imprimibles.

Caracteres ASCII imprimibles					
32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_		

El cuadro de abajo muestra los caracteres de control y el código que se agregó al código ASCII original para dar como resultado el ASCII extendido.

ASCII extendido (Página de código 437)					
128	Ç	160	á	192	Ł
129	ù	161	í	193	⌞
130	é	162	ó	194	⌟
131	â	163	ú	195	⌠
132	ä	164	ñ	196	—
133	à	165	Ñ	197	+
134	â	166	ª	198	ä
135	ç	167	º	199	Ä
136	ê	168	¿	200	ℒ
137	ë	169	©	201	ℓ
138	è	170	¬	202	⌞
139	ï	171	½	203	⌟
140	î	172	¼	204	⌠
141	ì	173	¡	205	=
142	Ä	174	«	206	≠
143	Å	175	»	207	□
144	É	176	⋮	208	δ
145	æ	177	⋮	209	Ð
146	Æ	178	⋮	210	Ê
147	ô	179	⌞	211	Ë
148	ö	180	⌟	212	Ě
149	ò	181	À	213	Ì
150	ù	182	Â	214	Í
151	û	183	Ã	215	Î
152	ÿ	184	©	216	Ï
153	Ö	185	⌞	217	⌟
154	Û	186	⋮	218	⌠
155	ø	187	⌞	219	■
156	£	188	⌟	220	■
157	Ø	189	¢	221	⋮
158	×	190	¥	222	⋮
159	f	191	¬	223	■
				224	Ó
				225	ß
				226	Ô
				227	Õ
				228	ö
				229	Õ
				230	μ
				231	þ
				232	þ
				233	Ú
				234	Û
				235	Ü
				236	ý
				237	Ý
				238	—
				239	·
				240	≡
				241	±
				242	—
				243	¾
				244	¶
				245	§
				246	÷
				247	¿
				248	°
				249	°
				250	·
				251	¹
				252	²
				253	²
				254	■
				255	nbsp

En los cuadros siguientes se separan en grupos algunos caracteres por sus características, esto con la finalidad de facilitar su búsqueda.

de uso frecuente (idioma español)	vocales con acento (español acento agudo)	vocales con diéresis	símbolos matemáticos	símbolos comerciales	comillas, llaves paréntesis
ñ alt + 164	á alt + 160	ä alt + 132	½ alt + 171	\$ alt + 36	" alt + 34
Ñ alt + 165	é alt + 130	ë alt + 137	¼ alt + 172	£ alt + 156	' alt + 39
@ alt + 64	í alt + 161	ï alt + 139	¾ alt + 243	¥ alt + 190	(alt + 40
¿ alt + 168	ó alt + 162	ö alt + 148	¹ alt + 251	¢ alt + 189) alt + 41
? alt + 63	ú alt + 163	ü alt + 129	² alt + 252	¤ alt + 207	[alt + 91
¡ alt + 173	Ä alt + 181	Å alt + 142	³ alt + 253	© alt + 169] alt + 93
! alt + 33	É alt + 144	Ê alt + 211	f alt + 159	© alt + 184	{ alt + 123
: alt + 58	Í alt + 214	Ï alt + 216	± alt + 241	ª alt + 166	} alt + 125
/ alt + 47	Ö alt + 224	Õ alt + 153	× alt + 158	º alt + 167	« alt + 174
\ alt + 92	Û alt + 233	Ü alt + 154	÷ alt + 246	° alt + 248	» alt + 175

Algunas características del código ASCII

- ♣ Conversión de mayúsculas a minúsculas y viceversa.

E = 45_h

Grupo al que pertenece

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

e = 65_h

Grupo al que pertenece

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

El único bit en donde los códigos son diferentes es en el bit 5: las minúsculas siempre contienen un 1 en el bit 5 y las mayúsculas siempre contienen un 0; por lo tanto, modificando el bit 5 de 0 a 1 y de 1 a 0 se puede ir de mayúsculas a minúsculas y viceversa. Además los bits 5 y 6 determinan en cuál de los cuatro grupos del set del ASCII se encuentra.

- ♣ Clasificación del grupo al que pertenece el carácter.

Bit 6	Bit 5	Grupo
0	0	Caracteres de control
0	1	Dígitos y puntuación
1	0	Mayúsculas y especiales
1	1	Minúsculas y especiales

📖 Buscar el concepto de logaritmo y encontrar la expresión que calcula n a partir de la que calcula x .

$$x = 2^n ; \quad n = 3.32 \log_{10} x$$

- ♣ Obtención del valor numérico de un carácter

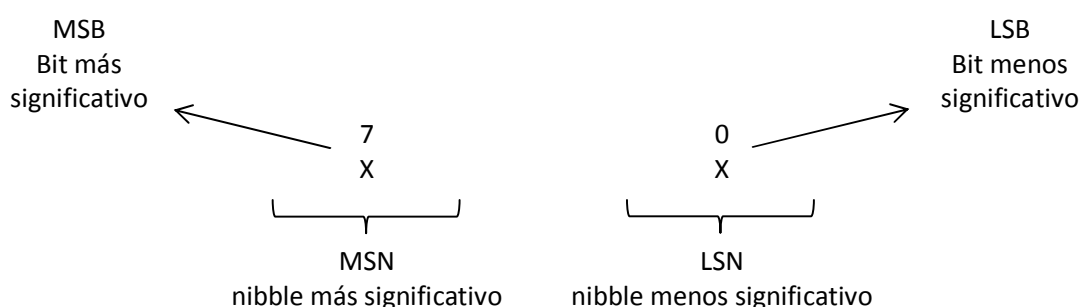
Considere el código ASCII de los caracteres numéricos

CHAR	DECIMAL	HEXADECIMAL
"0"	48	30
"1"	49	31
"2"	50	32
"3"	51	33
"4"	52	34
"5"	53	35
"6"	54	36
"7"	55	37
"8"	56	38
"9"	57	39

Se puede observar que quitando el nibble más significativo del código ASCII de un carácter numérico se puede obtener la representación binaria del número y, en consecuencia, su valor decimal. De manera inversa se puede convertir un valor binario en el rango de 0 a 9 a su representación ASCII agregando en el nibble más significativo un 3_h.

El código ASCII no usa el bit en la posición siete, lo que significa que sólo se usa la mitad de las posibles combinaciones; IBM usa los 128 restantes.

- ♣ Convención de nomenclatura para el peso en valor en un número binario.



Porqué un kilo en cómputo no es 1000:

The K Game

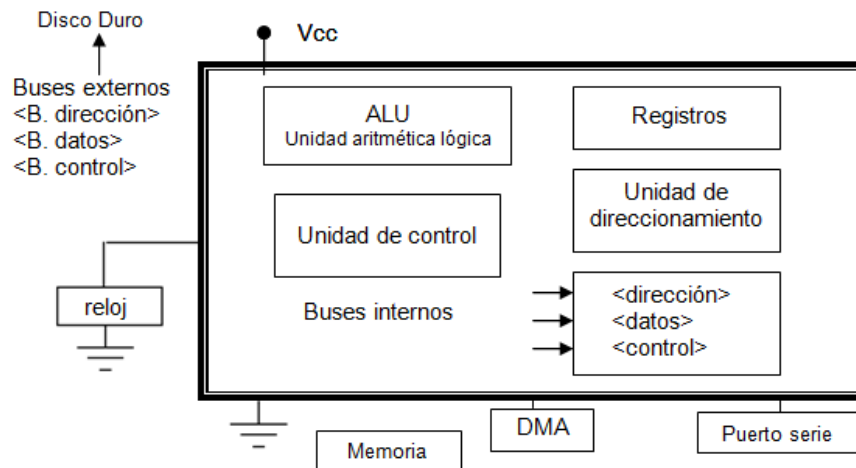
Most people use a convenient shorthand to represent 1,000- byte, or kilobyte, quantities of memory as in 64K, 128K and 640K. These convenient powers of 2 – in all cases equal in binary to a 1 followed by several zeros- have been adopted by computer users everywhere as accurate measurements of RAM, despite the fact that a 64K computer actually has 65, 536 bytes –the full number of values that can be expressed in 16 bits or 2^{16} .

The address range of the 8086 processor, by the way is 2^{20} or 1, 048, 576 bytes –a so-called megabyte plus change. As you'll learn in later chapters, the 8086 uses some hocus-pocus to reduce two 16-bit address values down to a 20-bit physical address that actually locates individual bytes within this memory range. The 80486 processor can address up to 2^{32} bytes. That's four gigabytes of memory or exactly 4, 294, 967, 296 bytes. (I don't know why they call a billion bytes a gigabyte. Maybe it should be a *billybyte*).

When working with address values in binary, try to get used to thinking in powers of 2. Measuring memory in K is quick and easy, but it is just too vague for the exacting world of assembly language programming.

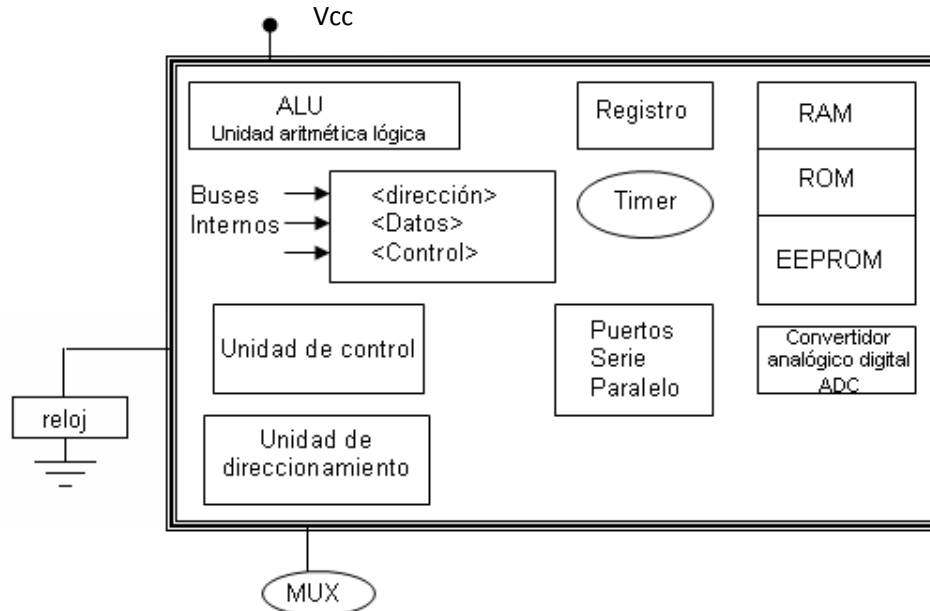
DIFERENCIAS A NIVEL DE BLOQUES ENTRE UN MICROPROCESADOR Y UN MICROCONTROLADOR

Microprocesador



Microcontrolador

Arranca solo y es utilizado para controlar eventos físicos.



TIMER: Para periodos de tiempo programados (temporizador).

RAM: Memoria principal.

ROM: arranque (sólo lectura).

EEPROM: puede ser borrada eléctricamente (electric erasable).

MUX: Multiplexor.

📖 Investigar el concepto de multiplexor y símbolos de tierra física, lógica y analógica.

📖 Introducir el siguiente código en el DEBUG en Shell de DOS y sacar las conclusiones correspondientes, observar con detenimiento los resultados del código en pantalla.

Nota: todo se maneja en hexadecimal dentro del DEBUG

C:/debug

		1	2	3	A	
e	100	FF	FF	FF	...	FF
		100	101	102		10A
d	100	P U M A S				// donde PUMAS estará formado por el correspondiente código de cada letra en hexadecimal
d	100	L 5				// mostrar, a partir de la dirección 100, 5 bytes

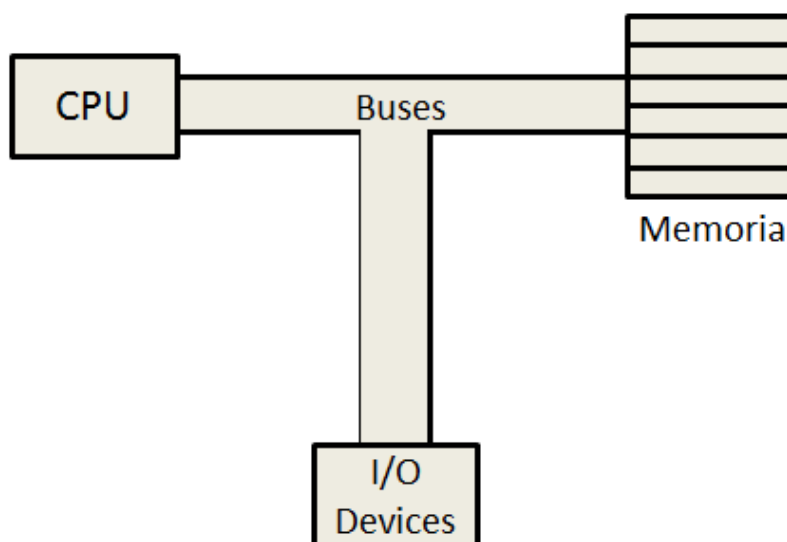
ARQUITECTURA DE UNA MÁQUINA

Familia 80 X 86

Al diseño básico operacional de un sistema se le llama arquitectura. La arquitectura John von Neumann tiene tres componentes principales:

- Unidad Central de Proceso (CPU)
- Memoria
- Dispositivos de entrada / salida (I/O)

La arquitectura de una máquina define su eficiencia.



Máquina típica de Von Neumann

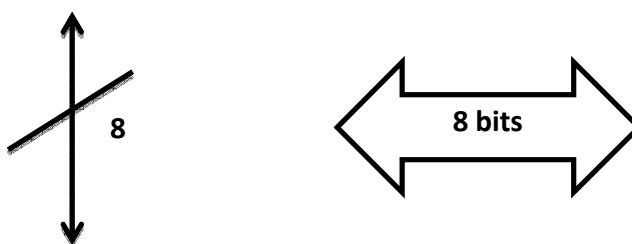
- ♣ **CPU:** Es donde todas las acciones toman lugar. Todos los cálculos ocurren en el CPU. Los datos y las instrucciones del CPU residen en memoria hasta que el CPU las requiera. Para el CPU la mayoría de los dispositivos de entrada/salida (I / O) son como elementos de memoria ya que éste puede guardar datos en un dispositivo de salida y leerlos de un dispositivo de entrada.

Discutiremos la organización de buses y de memoria, estos dos elementos son los que determinan el desempeño del software, aún más que la velocidad del procesador.

- ♣ **SISTEMA DE BUSES:** La familia 80 X 86 tiene tres buses principales:
 - El de direccionamiento
 - El de datos
 - El de control
- ♣ **BUS:** Es una colección de *cables* por los cuales las señales eléctricas pasan entre componentes en el sistema y éstos transportan datos entre el procesador, I/O y la memoria. Los buses son un cuello de botella, para el desempeño del procesador.
- ♣ **BUS DE DATOS:** Los procesadores 80X86 usan el bus de datos para intercambiar datos entre los diferentes componentes del sistema. El tamaño del bus varía ampliamente en la familia 80X86 y este bus define el **TAMAÑO DEL PROCESADOR**, típicamente 16, 32 o 64 líneas (líneas = bits porque es un sistema digital).

NOTA: El tener 8 bits en un bus **NO** significa que limita al procesador a datos de ocho bits solamente, sólo significa que el procesador **puede acceder UN BYTE por CICLO DE MEMORIA**. Por lo tanto, un procesador con un bus de 16 bits será naturalmente **más rápido** que aquellos que tienen ocho bits.

Representación gráfica de un bus



- ♣ **BUS DE DIRECCIONES:** Para diferenciar entre ubicaciones de memoria y dispositivos de I/O, el diseñador del sistema asigna una dirección de memoria a los dispositivos de I/O. Cuando el software desea acceder alguna dirección de memoria o dispositivo I/O pone la dirección correspondiente en el bus de direcciones.

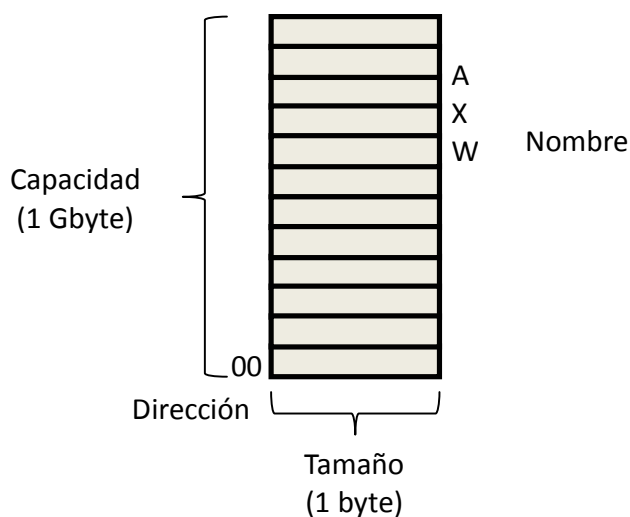
$$2^n = X$$

donde:

- n = número de líneas del bus de direcciones
- X = cantidad de memoria (CAPACIDAD)

Un mapa de memoria contiene:

- direcciones
- contenido
- nombres



Mapa de memoria

- ♣ **BUS DE CONTROL:** El bus de control es una colección de señales que controlan cómo el procesador se comunica con el resto del sistema:

Otras señales:

- relojes (clocks)
- líneas de interrupciones
- líneas de estado

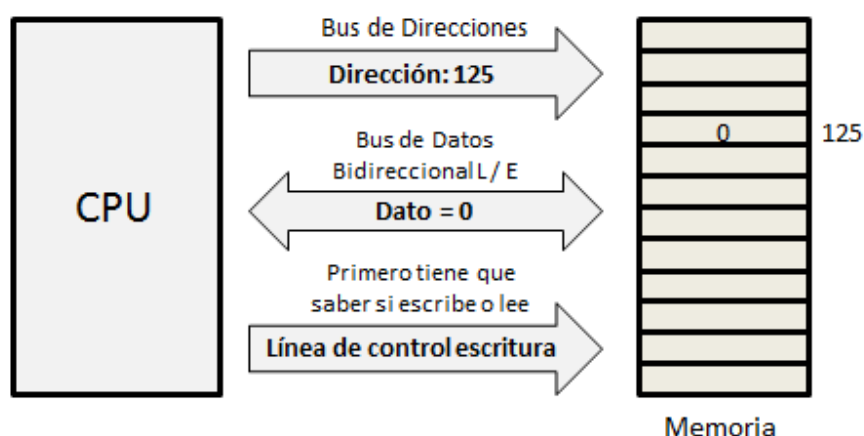
- ♣ **LÍNEAS DE LECTURA Y ESCRITURA:** Controlan la dirección en el bus de datos. Cuando ambos tienen un 1, el CPU y la memoria I/O no se están comunicando entre sí. Si la línea de lectura es cero, el CPU está leyendo datos desde la memoria. Si la línea de escritura es cero el sistema transfiere datos del CPU a la memoria.

NOTA: Escribir y leer en memoria son las tareas más importantes de una computadora y las que llevan más tiempo.

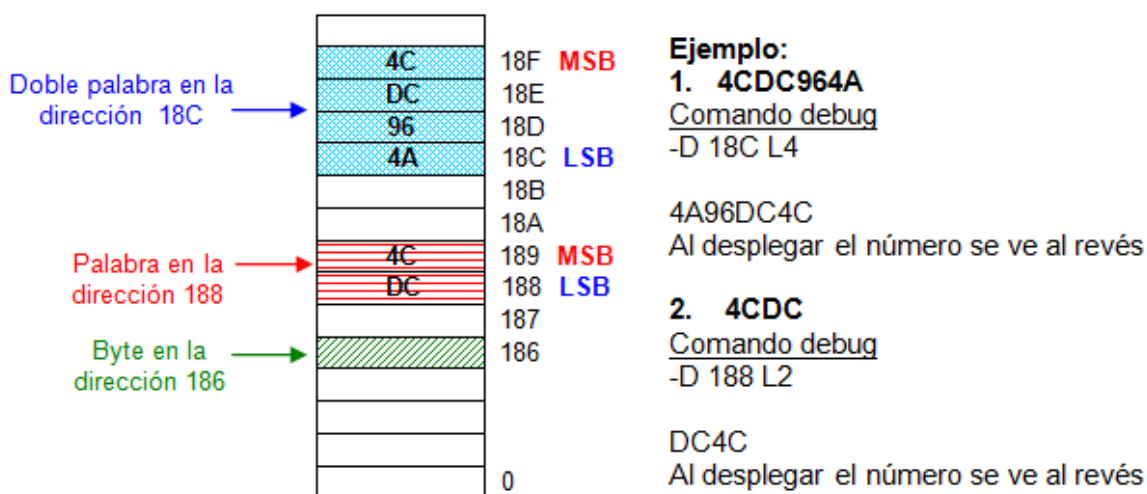
El subsistema de memoria

La memoria, vista como un arreglo lineal de bytes, tiene como dirección de la primera localidad la dirección cero y la dirección del último byte es $2^n - 1$, donde n es la longitud del bus de direcciones.

Para asignar el valor 0 a la dirección 125 el CPU pone el valor cero en el bus de datos, la dirección 125 en el bus de direcciones y pone la línea de escritura (operación de escritura a memoria).



Almacenamiento de un byte, palabra y doble palabra en memoria (formato Little-endian)




El término *Arreglo de Memoria Direccional por byte* significa que la unidad más pequeña posible de memoria que se puede acceder es de **ocho bits**. Un procesador de ocho bits (bus de datos = 8), puede manipular palabras y palabras dobles, esto requiere de operaciones múltiples sobre memoria debido a que este procesador puede mover ocho bits de datos en un paso, para cargar una palabra requiere de dos operaciones sobre memoria.

Los procesadores que tienen 16 bits en el bus de datos en memoria, organizan su memoria en dos bancos:

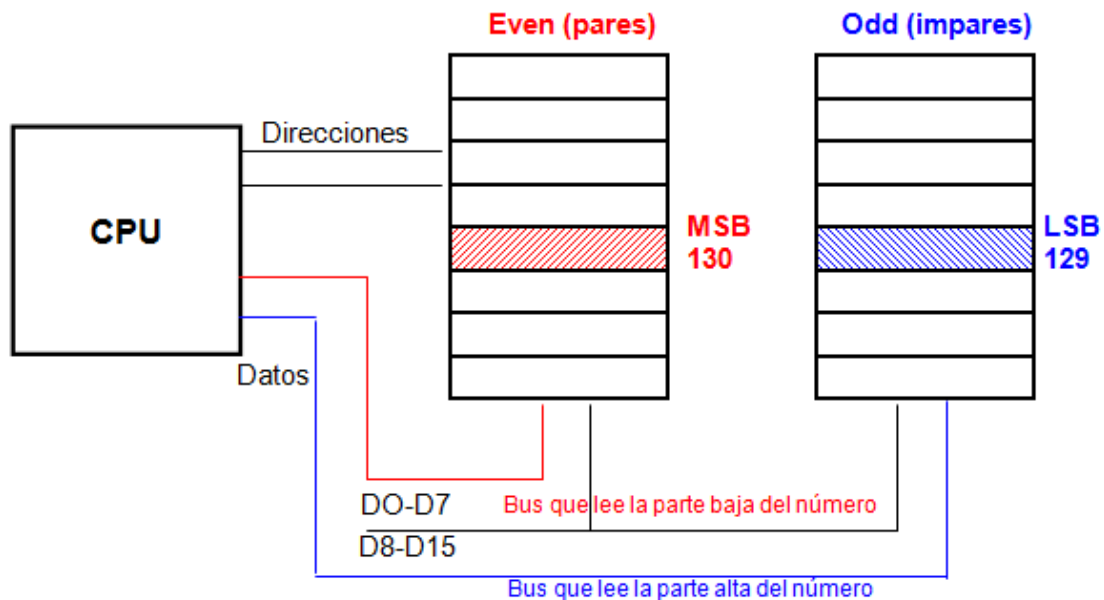
1. Par
2. Impar

Even	Odd
6	7
4	5
2	3
0	1

 Introducir el siguiente código en el DEBUG y sacar las conclusiones correspondientes, observar con detenimiento los resultados en pantalla y en los registros.

```
-debug
- a 100
XXXX:0100 mov ax, FF ←
XXXX:010X mov bx, FF ←
XXXX:01XX mov cx, FF ←
XXXX:    mov dx, FF ←
XXXX:    (para salir)
-r ← (anotar el contenido de los registros AX, BX, CX, DX)
-t ← (muestra el contenido de los registros)
```

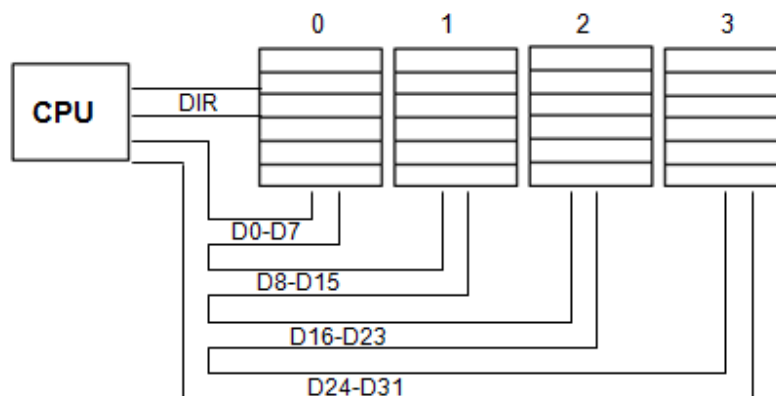
ORGANIZACIÓN DE LA MEMORIA DE PROCESADORES DE 16 BITS



NOTAS:

- La memoria está estructurada en bancos de bytes.
- Ubicar los datos en direcciones múltiples de cuatro, ya que en la primera lectura resuelve el número y no necesita que el procesador realice la función de swap, por lo tanto es más rápido.


Procesador de 32 bits



NOTA: Como regla general, siempre se deberán ubicar valores de palabras en direcciones pares y las dobles palabras en direcciones que son divisibles entre cuatro.

Subsistema de I/O

Además de las 20, 24 o 32 líneas de dirección por las cuales se accede a la memoria, la familia 80X86 provee un bus de direcciones de 16 bits de I/O. Esto le da al 80X86 dos espacios de direccionamiento separados: uno para operaciones de memoria y otro para operaciones de I/O. Estos dos sistemas comparten el mismo bus de datos y las 16 líneas (LSB) del bus de direccionamiento. El direccionamiento de I/O se comporta de la misma manera que el de memoria.

 Ensamblar el siguiente código en el DEBUG y sacar las conclusiones correspondientes, observar con detenimiento los resultados en pantalla y en los registros.

```
-debug
-a 100
XXXX:0100 mov ax, 02
XXXX:010X mov bx, 03
XXXX:01XX add ax, bx
XXXX:      nop
XXXX:      nop
XXXX:
-r ip
(¿Qué aparece?)
-r      (anotar el contenido de los registros AX, BX, CX, DX)
-t      (para ejecutar cada instrucción, y observar dónde se almacenan los datos y dónde queda el resultado)
```

Sistema de tiempo

Aun cuando las computadoras modernas son muy rápidas y cada vez lo son más, requieren de un tiempo finito para terminar o cumplir una tarea.

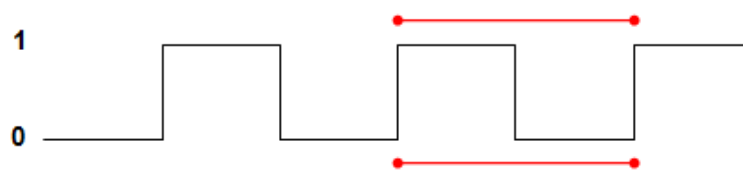
En las máquinas Von Neumann las operaciones son en serie, esto significa que las computadoras ejecutan las instrucciones en un orden predeterminado, es decir, no harán $I : I * 5 + 2$, (donde $:$ es asignación) antes de $I = J$ en la siguiente secuencia:

$$I: = J;$$
$$I: = I * 5 + 2$$

Y cada una de estas operaciones lleva su tiempo específico. Para asegurar que todas las operaciones ocurran en el momento exacto, el CPU usa una señal llamada el sistema de reloj o reloj.

El reloj (clock): éste maneja la sincronización dentro de un sistema de cómputo.

PERIODO = CICLO DE RELOJ



One clock = ciclo de reloj

Tiempo

$$f = 1.7\text{GHz}$$

$$T = \frac{1}{f}$$

$$T = \frac{1}{1.7 \times 10^9} = 0.5882 \times 10^{-9}$$

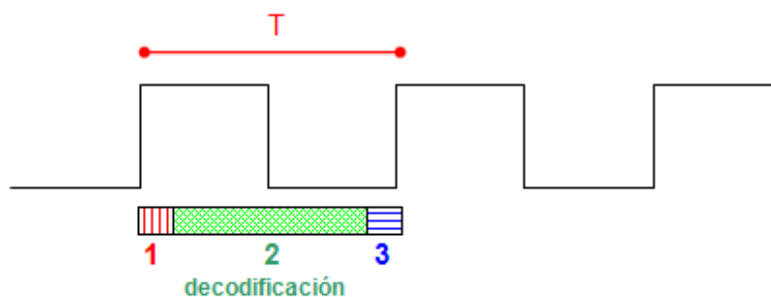
Es una máquina secuencial gobernada por esta señal. Su frecuencia determina la frecuencia del sistema. A un periodo se le llama *ciclo de reloj*.

Ejemplo: Un CPU a 50 MHz tendrá un período de reloj de 20 ns, para asegurar la sincronización del sistema empieza en la subida o bajada de la señal.

Acceso de memoria y el sistema de reloj

El acceso de memoria es probablemente la actividad más común del CPU. Toma varios ciclos de reloj acceder a una localidad de memoria. El tiempo de acceso de memoria es el número de ciclos de reloj que el sistema requiere para acceder a una localidad de memoria; éste es un valor importante, ya que un tiempo largo en esta operación resulta en un bajo desempeño. En el 8088 y 8086 se requerían de 4 ciclos de reloj para acceder a la memoria, en el 80486 se requería sólo de uno.

Ciclo de memoria del 80486

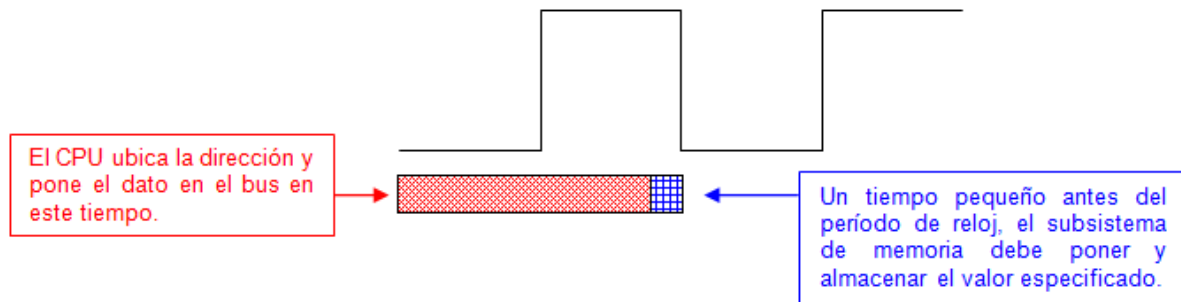


1. El CPU pone la dirección en el bus de direcciones durante este período.
2. El sistema de memoria debe decodificar la dirección, poner el dato en el bus de datos durante este período de tiempo.

3. Ciclo de Lectura de memoria: El CPU lee los datos del bus de datos durante este tiempo.

Ciclo de escritura en memoria

$$C = 3 \times 10^8 \left[\frac{m}{s} \right] \text{ Velocidad de la luz}$$



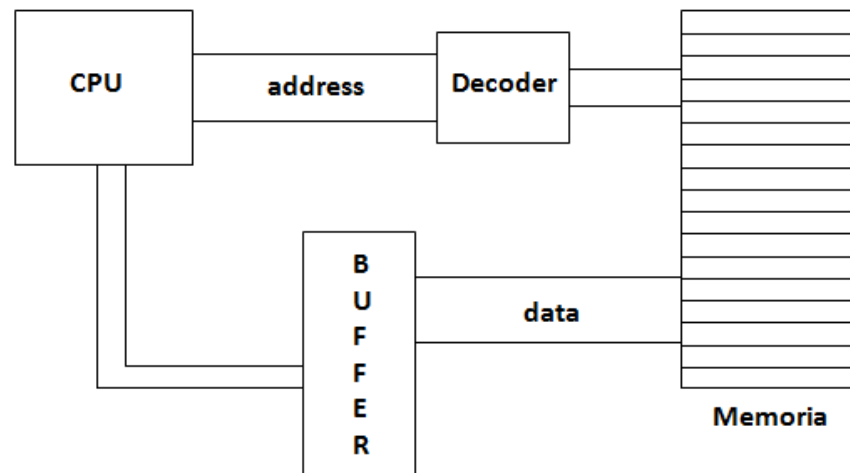
Cabe resaltar que el CPU no espera a la memoria, el tiempo de acceso está dado por la frecuencia del reloj. Si el subsistema no trabaja suficientemente rápido el CPU leerá basura y no escribirá apropiadamente los datos sobre memoria.

Los dispositivos de memoria tienen tres características principales:

1. Capacidad
2. Velocidad (tiempo de acceso)
3. Tamaño

Estados de espera

Los estados de espera no son más que ciclos de reloj extra para dar a algún dispositivo tiempo para completar una operación (además del tiempo de acceso hay retardos por lógica de decodificación y buffereo). Si se pierden 10 ns por el buffereo y el procesador necesita los datos en 20 ns, la memoria debe responder en menos de 10 ns.



Buffer

- ♣ **Memoria:** Guardar datos de manera temporal; es una memoria más o menos rápida.
- ♣ **Digital:** Mantener los niveles de voltaje y lógico de una señal.
- ♣ **Analógico:** Amplificador operacional.

Memoria caché

El objetivo de la memoria caché es reducir los estados de espera a cero (**cero estados de espera**); es decir, esta memoria va a la velocidad del procesador, puede leer o escribir desde memoria. Existen actualmente tres niveles de memoria caché y éstos son:

- L0 - L1 en el chip
- L2 fuera del procesador
- L3 fuera del procesador

Si se observa un programa típico, se descubrirá que se tiende a acceder a las mismas ubicaciones de memoria repetidamente, incluso se puede ver que un programa con frecuencia accede localidades de memoria adyacentes. Los nombres técnicos de estos fenómenos son *localidad temporal de referencia* (temporal locality of reference) y *localidad espacial de referencia* (spatial locality of reference).

Ambas formas se presentan en el siguiente código:

```
for i = 0 to i = 10 do
    a[i] = 0
```

El programa hace referencia a la variable *i* varias veces, el lazo de *for* compara *i* con 10 para ver si el lazo está completo. También incrementa *i* en uno en la parte inicial del lazo. La configuración usa *i* como un índice de arreglo. Esto muestra la *localidad temporal de referencia* ya que el CPU accede a *i* en tres puntos en un periodo corto de tiempo. Este programa también exhibe *localidad espacial de referencia*. El lazo escribe ceros

en los elementos del arreglo A, asumiendo que los elementos de A se almacenan consecutivamente en localidades de memoria adyacentes. Las instrucciones de cómputo también aparecen en memoria de manera secuencial (localidad espacial) y también las instrucciones son ejecutadas repetidamente una vez por cada iteración del lazo (localidad temporal).

Localidad Temporal de Referencia: Se accede a una localidad de memoria varias veces en un período corto de tiempo.

Localidad Espacial de Referencia: Se accede a localidades de memoria adyacentes.

Nota: Ruptura del ciclo caché, ocurre en saltos o programas muy grandes. La memoria caché tiene direcciones dinámicas.

Si se pudiera observar el perfil de ejecución de un programa típico se vería que se ejecutan menos de la mitad de las sentencias. Generalmente un programa puede usar sólo el 10 o 20% de la memoria que se le asigna.

Un programa de 1 Mbyte en memoria principal puede usar solamente 8 kB de datos y de código, por lo tanto, no se está usando toda la memoria en un momento dado, por ello se usa una pequeña cantidad de memoria rápida y que dinámicamente reasigna sus direcciones conforme el programa se ejecuta.

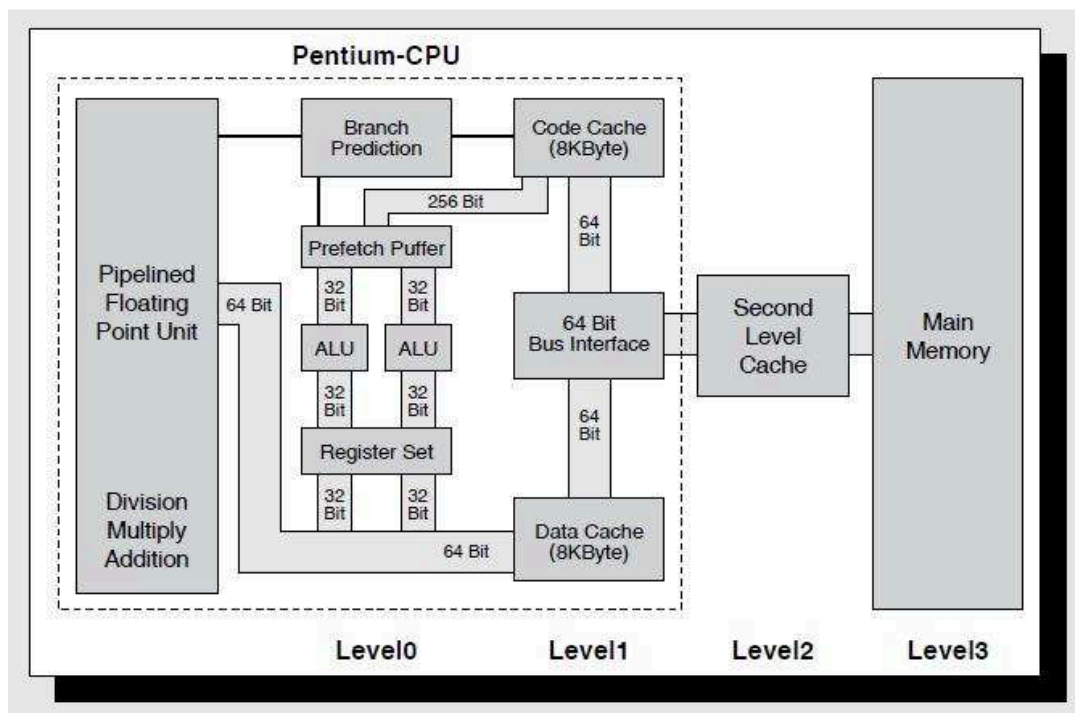
Concepto de Memoria Caché

La memoria caché se encuentra entre el CPU y la memoria principal. Es una pequeña porción de memoria muy rápida (cero estados de espera = velocidad del procesador).

A diferencia de la memoria normal, los bytes en esta memoria no tienen direcciones fijas. En su lugar, la memoria caché puede reasignar las direcciones de datos del objeto. Esto permite al sistema mantener los valores que fueron accedidos recientemente en la caché. Las direcciones a las cuales el CPU nunca ha accedido o no ha accedido en algún tiempo se mantienen en la memoria principal (lenta), los datos de memoria accedidos recientemente aparecen en la memoria caché.

Ruptura de la memoria caché

Sin embargo, cuando el programa salta a otra sección de memoria tiene que recurrir a direcciones alejadas de los contenidos en la memoria caché, por lo cual accede a la memoria principal y esto requiere de estados de espera.



Efectividad de la memoria caché

La calidad y la efectividad de la caché es medida como la relación entre los aciertos (caché hits) y los errores (caché misses). Un acierto ocurre cuando los datos solicitados por el procesador se encuentran en la caché. Así el procesador no tiene que acceder a una memoria más lenta. Un error significa que los datos no se encuentran en la caché, por lo que primero debe cargarse desde memoria a la caché, antes de que pueda ser pasada al procesador.

La relación entre los aciertos y los errores principalmente depende de tres factores:

1. Organización de la caché
2. El tipo de código en el programa que está siendo ejecutado, y
3. Obviamente, el tamaño de la caché

El tercer factor es fácil de analizar, ya que a mayor tamaño de la memoria mayor probabilidad de que los datos a usar estén contenidos en la caché. Para el segundo factor se aplican los fenómenos anteriores.

Hay otros dos factores que son prerequisites para el uso eficiente de la memoria caché. Estos dos factores entran en la categoría de la organización de la caché. El primer factor es la estrategia de la memoria caché, en relación con escribir y leer accesos, mientras el segundo factor es la arquitectura de la caché, por ejemplo la forma en que se almacena la información en ésta.

Ciclo de Instrucción (básico)

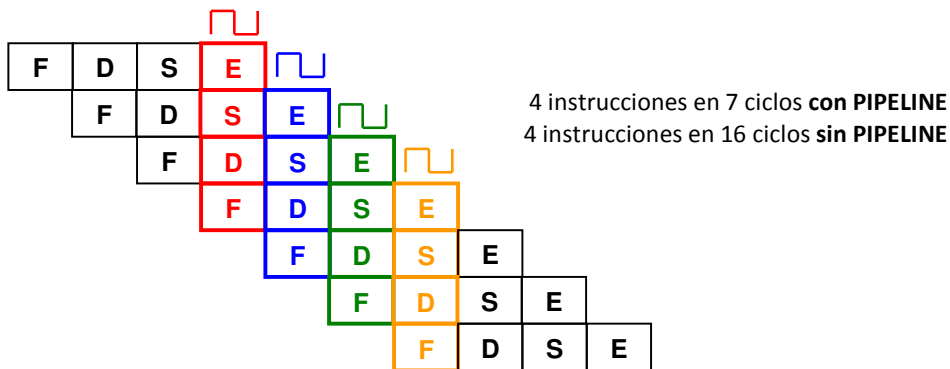
1. *Fetch*: búsqueda de la instrucción en memoria programa.
2. *Decodificación*: la unidad de control descifra las acciones a tomar y envía señales de control.
3. *Búsqueda de operandos*.
4. *Ejecución de instrucción*.

Pipeline

El pipeline entuba instrucciones. Tiene como objetivo ejecutar una instrucción por cada ciclo de reloj.

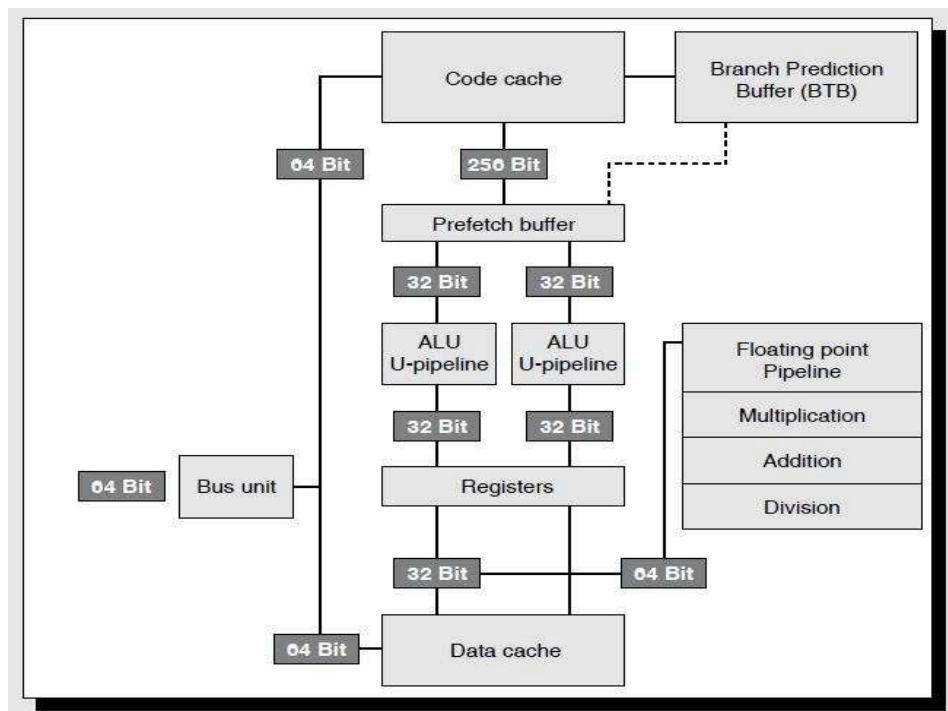


Frecuencia con la que se ejecuta una instrucción: $= \frac{1 \text{ GHz}}{4}$



Superscalar pipeline architecture

Se le llama *Superscalar pipeline architecture* a la arquitectura de una máquina cuando se agrega otro pipeline de manera paralela.



Arquitectura del procesador Intel Pentium

Con la ayuda de estos dos pipelines, el Pentium teóricamente podría ejecutar dos instrucciones simultáneamente y, como resultado, duplicar la velocidad de ejecución. Sin embargo, este proceso no es tan fácil.

Frecuentemente dos instrucciones secuenciales pueden ser ejecutadas solamente en secuencia debido a que una depende de la otra. Existen reglas que hacen que la ejecución de dos instrucciones de manera simultánea sea imposible. Una de esas reglas es la limitación de la ejecución paralela de instrucciones simples. Algunos de los ejemplos de instrucciones sencillas son instrucciones MOV, suma y resta de enteros, instrucciones push y pop entre otras, de tal manera que el código de programa determina si dos instrucciones pueden o no ser ejecutadas de manera simultánea o éstas tienen que pasar las diferentes etapas de la secuencia del pipeline U. Las optimizaciones a los compiladores para el Pentium consideran esto organizando el código de tal manera que las instrucciones secuenciales permitan la ejecución simultánea tanto como sea posible.

Branch Target Buffer

La eficiencia del principio del pipeline está basada sobre la constante provisión de nuevas instrucciones al pipeline. Sólo cuando las etapas o pasos del pipeline están permanentemente llenos parece ser que las instrucciones pueden ser ejecutadas en un ciclo. Sin embargo, esto no es útil si el procesador tiene que ejecutar una instrucción de salto. En este caso, en lugar de continuar con la siguiente instrucción, la ejecución del programa continúa con una instrucción completamente diferente. Como resultado, la ejecución de las siguientes instrucciones, que están ya en el pipeline, deben cancelarse y el pipeline debe cargarse con nuevas instrucciones.

El Pentium usa el Branch Target Buffer (BTB) para evitar este problema de las instrucciones jump. Este buffer es usado en la etapa D1 (decodificación) de la ejecución de instrucción para todos los tipos de jump NEAR. Si el procesador encuentra tal instrucción en la etapa D1, usa la dirección de la instrucción en la memoria para buscar el BTB para la instrucción. Cada vez que el procesador ejecuta uno de estos saltos guarda la dirección de la instrucción y la dirección del destino del salto en el BTB. Si la instrucción está registrada ahí es porque ya ha sido ejecutada, el procesador asume que el salto deberá ejecutarse nuevamente. En lugar de cargar el sucesor de la instrucción jump en el pipeline, el procesador carga el comando a la dirección objetivo.

Sin embargo, si la instrucción no está registrada en el BTB, la instrucción subsecuente es cargada en el pipeline. Durante la etapa de ejecución, el procesador determinará si ejecuta o no el jump. Si el procesador predijo exactamente con la dirección en el BTB, la instrucción que sigue a la instrucción jump estará ya en el pipeline. Por tanto, la ejecución del programa puede continuar inmediatamente. Incluso cuando la ejecución del salto condicional sólo tome un ciclo en este caso.

No obstante, si la predicción es incorrecta, esto significa que las instrucciones incorrectas están en el pipeline. Por lo que el pipeline debe hacer un “flush”. Esto implica la cancelación de la ejecución de las instrucciones que se encuentran actualmente en el pipeline y recargar completamente el pipeline. Esto resulta en que en lugar de que tome un ciclo, al menos deben realizarse tres ciclos para ejecutar la instrucción jump.

Ejercicio:

Se tiene un microprocesador con 20 Gb de memoria en disco duro, 512 Mb en RAM y procesador de 32 bits. Obtener, de ser posible, los siguientes datos; de otra manera argumentar por qué no.

Bus de datos =

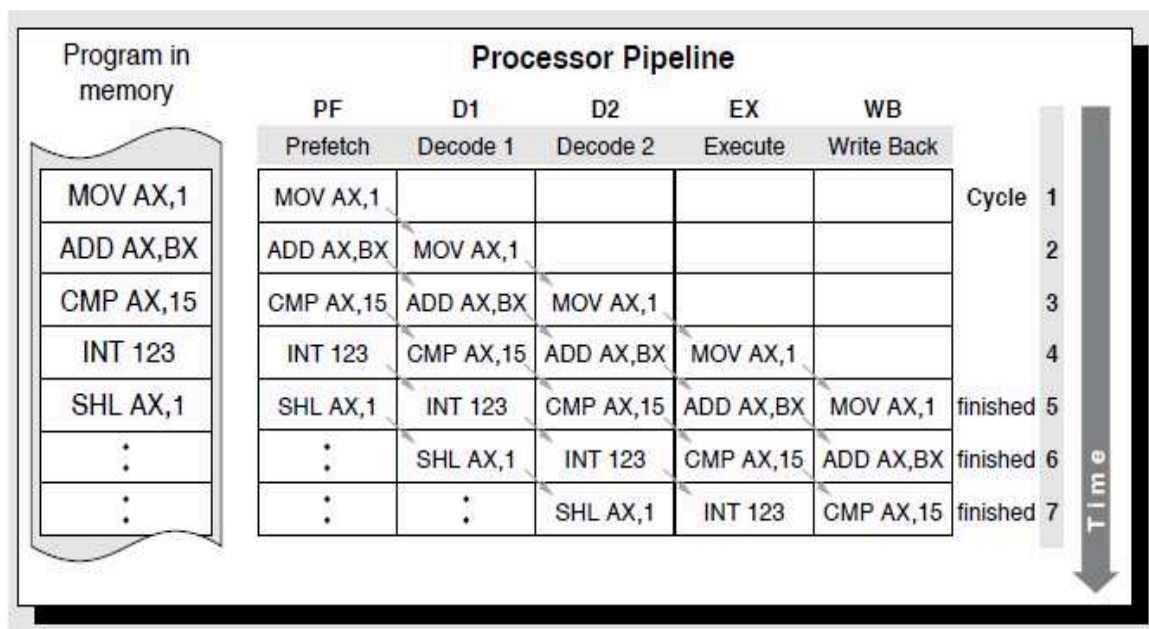
Bus de control =

Bus de direcciones =

Si el microprocesador va a ejecutar un programa de 200 líneas de código, ¿cuánto tardará en ejecutarlo si su ciclo de instrucción es estático y de 8 pasos?

- a) Sin pipeline
- b) Con un pipeline simple
- c) Con triple pipeline (superscalar pipeline architecture)

Pipeline en el procesador Pentium

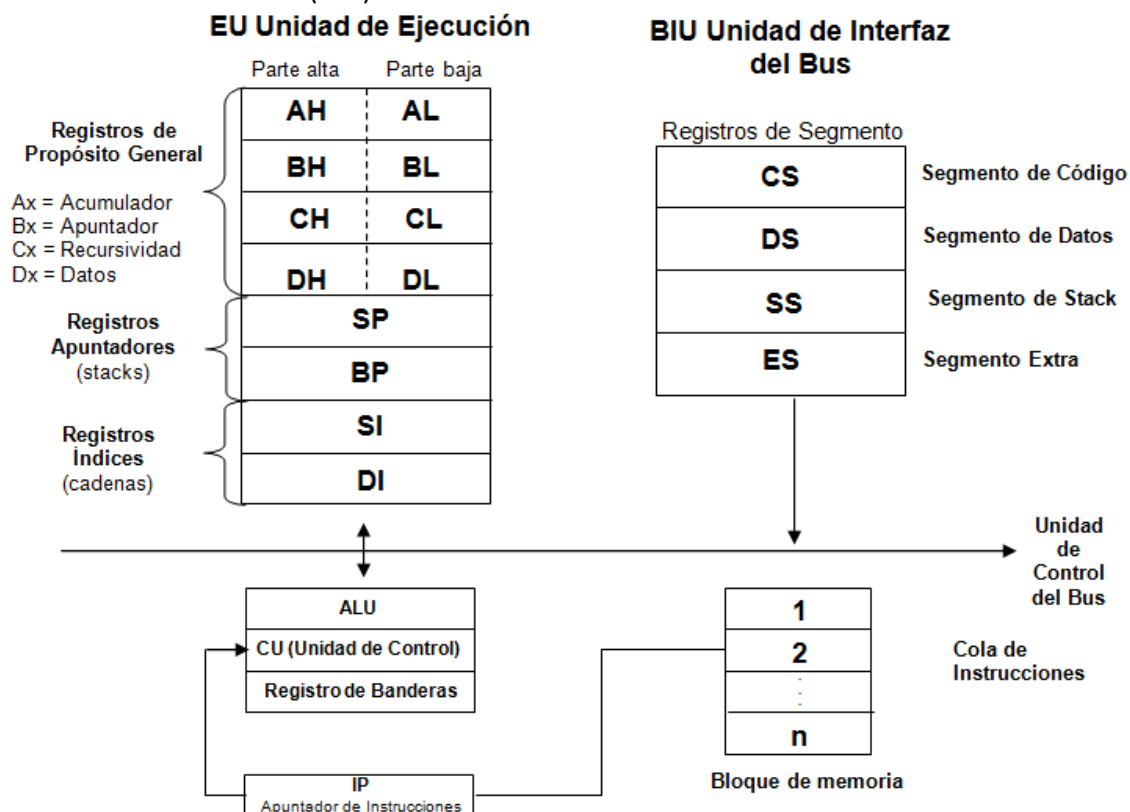


Registros del CPU

Unidad de ejecución y Unidad de interfaz del bus

El procesador se divide en dos unidades lógicas:

1. Unidad de Ejecución (EU)
2. Unidad de Interfaz del Bus (BIU)



El papel de la EU es ejecutar instrucciones, mientras que la BIU envía instrucciones y datos a la EU. La EU contiene una unidad aritmético-lógica (ALU), una unidad de control (UC) y varios registros. Estos elementos ejecutan instrucciones y operaciones aritmético-lógicas.

La función más importante de la BIU es manejar la unidad de control del bus, los registros de segmentos y la cola de instrucciones. Otra de sus funciones es permitir el acceso a instrucciones, esta unidad debe ir por las instrucciones a la memoria y colocarlos en la cola de instrucciones.

Puesto que el tamaño de esta cola es de 4 a 32 bytes, dependiendo del procesador, la BIU es capaz de adelantarse y buscar con anticipación instrucciones, de manera que siempre haya una cola de instrucciones lista para ejecutarse.

Segmentación de la memoria en el microprocesador 80X86

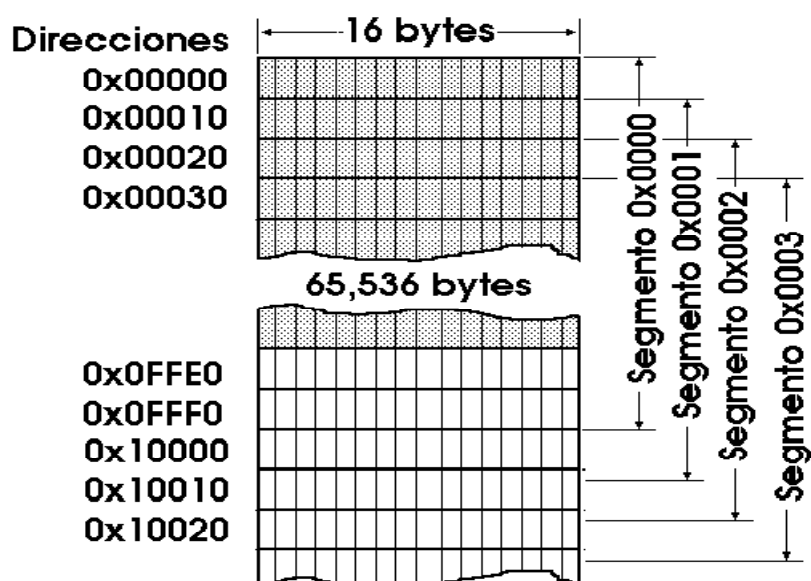
El CPU empleado en las primeras computadoras IBM PC y compatibles, el microprocesador 8086 de Intel, permitía direccionar $1\text{ MB} = 2^{20} = 1,048,576$ bytes. Los procesadores más modernos de Intel de la familia 80X86 (80286, 80386, 80486 y Pentium) empleados en las computadoras PC pueden direccionar un mayor número de localidades (hasta $16\text{ MB} = 2^{24} = 16,777,216$ bytes para el 80286 y hasta $4\text{ GB} = 2^{32} = 4,294,967,296$ bytes para el 80386, 80486 y Pentium) pero para mantener compatibilidad con el microprocesador 8086 tienen dos formas de operación: Una que imita al microprocesador 8086, llamado *modo real*, y otro que explota todo el poder del procesador, llamado *modo protegido*. El sistema operativo MSDOS, a fin de mantener compatibilidad con el software existente, sólo permite la operación de los microprocesadores en modo real y para fines prácticos una computadora PC bajo ese sistema operativo se comportará como si tuviese un procesador 8086 (aunque más rápido).

En el microprocesador 8086 (o en los microprocesadores más modernos de la familia 80x86 operando en el modo real) visualiza la memoria como dividida en segmentos, los cuales tienen las siguientes características:

- Los segmentos no son divisiones físicas. Un segmento es una ventana lógica a través de la cual el programa visualiza porciones de la memoria.
- Los segmentos empiezan cada 16 bytes. Como a un bloque de memoria de 16 bytes se le conoce como párrafo, los segmentos empiezan a cada párrafo.
- Hay hasta 65,536 segmentos ($1,048,576 / 16 = 65,536$).
- El primer segmento es el segmento 0, el siguiente es el 1, etc. El número de segmento se conoce como la *dirección del segmento*.
- La dirección real, también llamada dirección efectiva, en la que empieza un segmento, se obtiene multiplicando la dirección del segmento por 16.
- Los segmentos pueden ser tan pequeños como 16 bytes y tan grandes como $64\text{ KB} = 65,536$ bytes.
- Los segmentos pueden traslaparse. Esto ocurre cuando un segmento es mayor a 16 bytes.
- Al cargar un programa a memoria, éste ocupa uno o más segmentos. Los segmentos asignados a un programa no tienen que estar uno junto al otro aunque frecuentemente lo están.

Se acostumbra expresar la dirección de una localidad de memoria mediante la combinación formada por la dirección del segmento en que se encuentra la localidad de memoria y el desplazamiento de la localidad con respecto al principio del segmento.

Segmento:Desplazamiento



Ambos, la dirección de segmento y el desplazamiento, son valores de 16 bits. Estos dos valores se combinan para formar la dirección real o efectiva de la localidad de la siguiente manera:

$$\text{Dirección real} = \text{segmento} \ll 4 + \text{desplazamiento}$$

Debido al traslape entre los segmentos muchas combinaciones *segmento:desplazamiento* nos pueden dar la misma dirección real, por ejemplo: la localidad de memoria cuya dirección real es 0x01000 puede expresarse mediante las siguientes combinaciones *segmento:desplazamiento*:

Segmento:Desplazamiento Dirección real

0x0000:0x1000	$0x00000 + 0x1000$	= 0x01000
0x0001:0x0FF0	$0x00010 + 0x0FF0$	= 0x01000
0x0002:0x0FE0	$0x00020 + 0x0FE0$	= 0x01000
...		
0x0010:0x0F00	$0x00100 + 0x0F00$	= 0x01000
...		
0x0020:0x0E00	$0x00200 + 0x0E00$	= 0x01000
...		
0x00FF:0x0010	$0x00FF0 + 0x0010$	= 0x01000
0x0100:0x0000	$0x01000 + 0x0000$	= 0x01000

Cuando el microprocesador quiere calcular una dirección real, carga en un registro el valor del segmento y en otro registro el valor del desplazamiento y realiza las operaciones de corrimiento y suma.

Mapa de memoria física

Se dispone de algo similar a una *hoja de trabajo* para el almacenamiento temporal y ejecución de programas. Ésta se pierde al no tener polarización.

Segmentos y direccionamiento

Un segmento es un área especial en un programa que inicia en un límite de párrafo (16 bytes), esto es, en una localidad regularmente divisible entre la 010_h (ya que de esta manera lee de forma natural sobre los buses, esto representa mayor velocidad), aunque un segmento puede estar ubicado casi en cualquier lugar de la memoria y en modo real (existe también el modo protegido) puede ser hasta de 64 k, sólo necesita tanto espacio como el programa requiera para su ejecución. Hay tres segmentos principales:

1. Segmento de Código
2. Segmento de Datos
3. Segmento de Pila

♣ Segmento de código (CS)

Contiene las instrucciones de máquina que son ejecutadas. Generalmente la primera instrucción ejecutable se encuentra en el inicio del segmento y el Sistema Operativo enlaza a esta localidad para iniciar la ejecución del programa. CS es el registro que apunta al inicio de este segmento.

Nota: Si se requieren más de 64 k de código, es posible usar más de un segmento de código.

♣ Segmento de datos (DS)

Contiene datos, variables y áreas de trabajo definidas por el programa. DS es el registro que apunta al inicio de este segmento.

Diferencias entre *constante* y *variable*:

Constante	Variable
Se resuelve en tiempo de compilación (preproceso)	Se resuelve en tiempo de ejecución
No ocupa espacio en memoria	Sí ocupa espacio en memoria

♣ Segmento de pila (SS)

En términos simples, la pila contiene los datos y direcciones que nosotros necesitamos guardar temporalmente o para su uso en *llamadas* a subrutinas (procedimientos). SS es el registro que apunta al inicio de este segmento.

Diferencias entre *segmento* y *registro*:

Segmento	Registro
Es un bloque de memoria	Es una sola dirección

Límite de los segmentos

Los registros de segmentos contienen la dirección inicial de cada segmento.

SD	Dirección
DS	Dirección
CS	dirección

000F

000F

Segmento de pila
Segmento de datos
Segmento de código
Memoria

Desplazamiento de segmentos

En un programa todas las localidades de memoria están referidas a una dirección inicial de segmento. La distancia en bytes desde la dirección del segmento se define como desplazamiento (offset).

El primer byte del segmento de código contiene un desplazamiento 00, el segundo un desplazamiento de 01. Para referir cualquier dirección de memoria en un segmento, el procesador combina la dirección del segmento en un registro de segmento con un valor de desplazamiento.

Registro DS → 045F [0]_h y una instrucción hace referencia a una localidad con un desplazamiento de 0032_h bytes dentro de DS.

La localidad real de memoria del byte referido por la instrucción es 04622_h.

Direccionamiento del DS

$$\begin{array}{r}
 \text{DS} \quad 045F0_{\text{h}} \\
 + \text{Desplazamiento} \quad 0032_{\text{h}} \\
 \hline
 4622_{\text{h}}
 \end{array}$$

Un programa tiene uno o más segmentos, los cuales pueden iniciar casi en cualquier lugar de memoria, variar en tamaño y estar en cualquier orden.

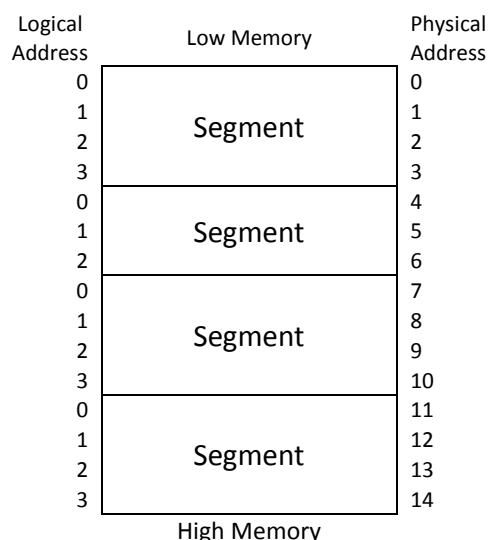
Direcciones lógicas y físicas

No matter how much memory the processor can address, and no matter how many memory chips are installed inside the computer, the smallest memory unit remains the 8-bit byte. As mentioned earlier, each byte has a unique location, called the physical address, which programs specify to read and write the bytes they need. Obviously, you need a greater number of bits to represent the physical addresses of greater amounts of memory. If your computer had only 64K, then the address of any byte would comfortably fit in 16 bits, which can represent values from 0 to 65, 535 ($2^{16} - 1$) –or 64K in round numbers. To address the PC's maximum 1 megabyte of memory requires a minimum of 20 bits. ($2^{20} - 1$ equals 1, 048, 575, or hexadecimal FFFFF). The problem is: 8086 registers are only 16 bits wide. How is it possible for the 8086 processor to access the full megabyte of memory in a typical PC?

The answer is memory segmentation, a method the 8086 uses to divide its large address space into logical 64K chunks. With this method, the address of a specific byte can be expressed in two values: the address of the chunk, or segment, plus a 16-bit offset from the beginning of the segment. Together the combination of segment and offset values is called the logical address. The first byte in a segment is at offset 0000, the second at offset 0001, the third at 0002, and so on –no matter where the segment physically begins in memory. Figure 4.1 illustrates this idea, showing that each location in memory has both a physical address (right) and a logical address (left), expressed as an offset from the beginning of a segment boundary. With segmentation the 8086 processor can efficiently address up to 1 megabyte of memory while using relatively small, 16-bit registers. As an additional benefit, segmentation makes it easy to move programs to new physical locations by changing only the segment base address. The offset values within a segment require no adjustments, allowing *for relocatable programs* that can run identically in different memory locations.

Figure 4.1

Logical addresses all have equivalent physical addresses in memory.



Paragraphs, Segments and Offsets

To locate the beginnings of memory segments, the 8086 processor contains four 16-bit segment registers. Internally, the processor combines the value of one segment register with a 16-bit offset (the logical address) to create a 20-bit physical address. It does this by first multiplying the segment value by 16 and then adding the offset to the result. Because of the multiplication –equivalent to shifting the bits left four times, as you recall –segment boundaries fall on physical address multiples of 16 bytes. Each of these 16-byte memory tidbits is called a paragraph. A simple calculation proves there are a maximum of 65, 536 paragraphs –and, therefore, an equal number of segment boundaries– in the 8086's 1-megabyte address space (1, 048, 576 / 16). (Notice that this also equals the number of values you can express in one 16-bit segment register). Here are a few other important facts about segments to keep in mind:

- ♣ Segments are not physically etched in memory –a common misconception. A segment is a logical window through which programs view portions of memory in convenient 64K chunks.
- ♣ A segment's starting location (that is, the segment's logical address) is up to you and can be any value from 0000 to FFFF hex. Each logical segment value (0, 1, 2, ..., 65 535) corresponds to a physical paragraph boundary (0, 16, 32, ..., 1 048 560).
- ♣ Segments can be as small as 16 bytes or as large as 64K (65 536 bytes). The actual size of a segment is up to you and your program.
- ♣ Segments do not have to butt up against each other physically in memory, although they often do.
- ♣ Segments can overlap with other segments; therefore, the same byte in memory can have many different logical addresses specified with different but equivalent segment and offset pairs. Even so, each byte has one and only one 20-bit physical address.

This last point confuses almost everyone on their introduction to memory segmentation. Two different segment and offset pairs can (and often do) refer to the same byte in memory. If you remember how the processor creates a 20-bit physical address –multiplying the segment value by 16 and adding the offset –you can see that the *segment:offset* hexadecimal values 0000:0010 and 0001:0000 refer to the same physical location. Duplicating in decimal how the 8086 processor converts these logical addresses to physical addresses, each calculation –(0000 x 16) and (0001 x 16) + 0 –gives the same result, 16.

- 📖 Investigar las diferentes memorias que existen y el significado de sus siglas (Flash, EEPROM, SDRAM, RAM, EPROM, ROM)
- 📖 Ensamblar el siguiente código en el DEBUG y sacar las conclusiones correspondientes, observar con detenimiento los resultados en pantalla y en los registros.

```

-debug
-r      (observar el contenido de los registros AX, BX, CX, DX)
-a 100
XXXX:0100 mov ax, FF
XXXX:010X mov bl, al
XXXX:01XX mov al, 00
XXXX:01XX mov cl, bl
XXXX:01XX mov bl, 00
.
.
XXXX:      (para terminar de editar 2)
-r
-t
-t
-t (hasta que termine)

Nota: antes de ejecutar verificar que el registro IP contenga 100
  
```

AH	AL
BH	BL
CH	CL
DH	DL

Pasar el contenido entre las partes altas y bajas de los registros

REGISTROS

Se emplean para controlar instrucciones en ejecución, manejar direccionamiento de memoria y proporcionar capacidad aritmética, los registros son direccionables por medio de un nombre.

♣️ Registros de segmento

Tienen 16 bits de longitud y facilitan un área de memoria para direccionamiento conocida como el segmento actual.

♣️ Registro CS

El DOS almacena la dirección inicial del segmento de código de un programa en el registro CS (no necesita ser referenciado).

$$CS \times 16 + \text{desplazamiento} = \text{instrucción a ejecutar}$$

♣️ Registro DS

La dirección inicial de un segmento de datos de programa es almacenada en el registro DS.

♣️ Registro SS

Este registro permite la colocación en memoria de una pila, para almacenamiento temporal de direcciones y datos, el DOS lo controla cuando es un programa ejecutable .com.

♣ Registro ES

Algunas operaciones con cadenas de caracteres utilizan el registro extra de segmento para manejar direccionamiento de memoria. El registro ES está asociado con el registro DI (índice); puede ser inicializado.

♣ Registro apuntador de instrucciones (IP)

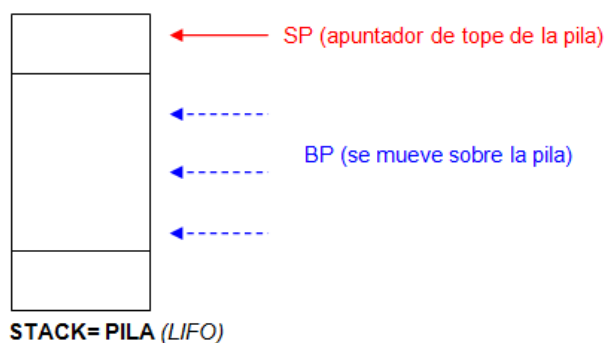
Registro de 16 bits. Contiene el desplazamiento de dirección de la siguiente instrucción que se ejecutará. Para saber qué instrucción será ejecutada se usan CS e IP.

$$\begin{array}{r} \text{CS} \\ + \text{IP} \\ \hline \text{Dirección de la siguiente} \\ \text{instrucción} \end{array} \qquad \begin{array}{r} 25A40_h \\ + 412_h \\ \hline 25E52_h \end{array}$$

$$\begin{array}{r} \text{CS} \times 16_d \\ + \text{offset} \\ \hline \text{Dirección real} \end{array}$$

♣ Registros Apuntadores

Los registros SP (apuntador de pila) y BP (apuntador de base), permiten al sistema acceder datos en el segmento de la pila.



- *Registro SP:* Proporciona un valor de desplazamiento de 16 bits que se refiere a la palabra actual que está siendo procesada en la pila. El procesador se hace cargo de la pila de forma automática. Si queremos usarlo, su contenido debe ser guardado antes.
- *Registro BP:* Registro de 16 bits. Permite la referencia a parámetros, los cuales son datos y direcciones transmitidos vía pila (saltos a procedimientos).

♣ REGISTROS DE PROPÓSITO GENERAL

(AX, BX, CX, DX). Se pueden direccionar como una palabra o como una parte (byte). El byte de la parte izquierda es la parte *alta* y el último byte de la derecha es la parte *baja*.

CX → 16 bits

CH → 8 bits = 1 byte

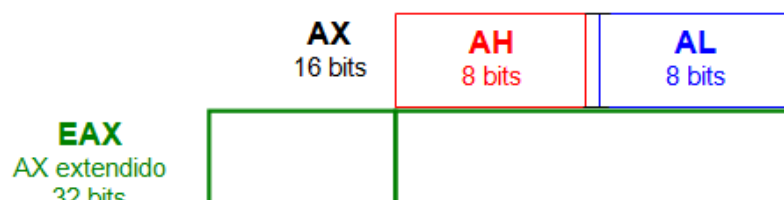
CL → 8 bits = 1 byte

Por ejemplo, las siguientes instrucciones mueven ceros a los siguientes registros CX, CH y CL respectivamente.

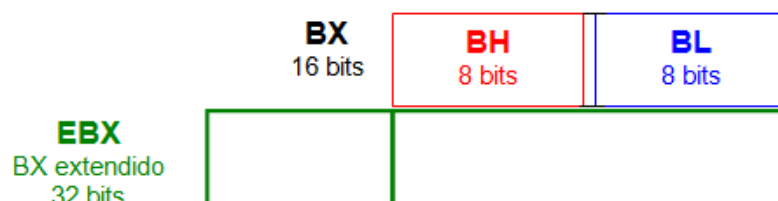
```
mov CX, 00h;    inicializando o limpiando
mov CH, 00h
mov CL, 00h
```

destino origen

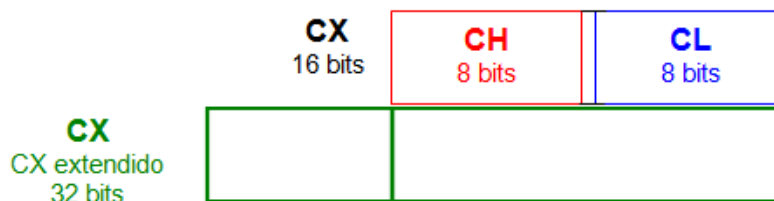
- ♠ **Registro AX:** Es el acumulador principal, es utilizado para operaciones que implican entrada/salida y la mayor parte de la aritmética, (operaciones de suma, resta, multiplicación, división además de operaciones lógicas).



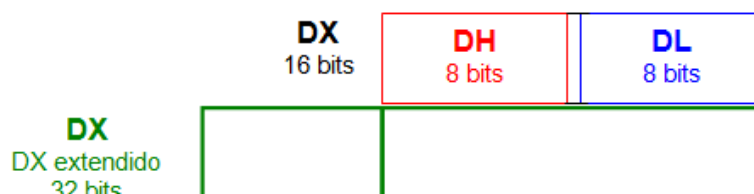
- ♠ **Registro BX:** Conocido como el registro base ya que es el único registro de propósito general que puede ser un índice. Para direccionamiento indexado usado también para cálculos como (operaciones de suma, resta, multiplicación, división además de operaciones lógicas).



- ♠ **Registro CX:** Es conocido como el registro contador. Puede contener un valor para controlar el número de veces que un ciclo se repite o un valor para corrimiento de bits hacia la derecha o hacia la izquierda; es usado también para cálculos.



- ♠ **Registro DX:** El DX es conocido como el registro de datos, algunas operaciones requieren su uso y las operaciones de multiplicación y división con cifras grandes. Suponen al DX y al AX trabajando juntos.



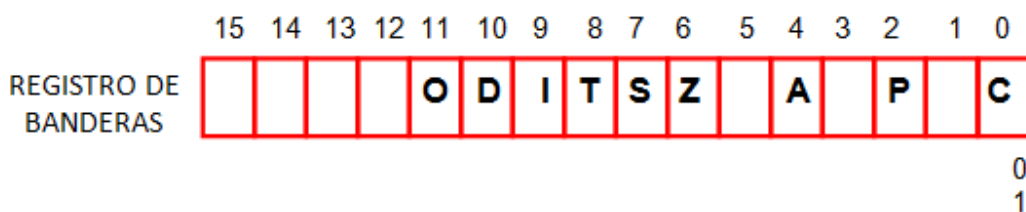
♣ Registros índice

- Registro SI (source index): El registro índice (fuente de 16 bits) es requerido por algunas operaciones con cadenas de caracteres.
- Registro DI (destination index): El registro índice destino, también es requerido por algunas operaciones con cadena de caracteres.

♣ Registro de banderas

Es de 16 bits, las banderas sirven para indicar el estado actual de la máquina y el resultado del procesamiento. Muchas instrucciones que piden comparaciones y realizan aritmética cambian el estado de las banderas, con cuyas instrucciones se pueden realizar pruebas para determinar la acción subsecuente.

BANDERAS	
OF (Overflow, desbordamiento)	Índice de desbordamiento de un bit del orden más alto (más a la izquierda MSB) después de una operación aritmética.
DF (Dirección)	Designa la dirección hacia la izquierda o a la derecha para mover o comparar cadenas de caracteres.
IF (Interrupción)	Indica que una interrupción externa como la entrada desde el teclado sea procesada o ignorada (enmascaramiento de instrucciones).
TF (Trampa)	Permite la operación del procesador paso a paso. Los programas depuradores como DEBUG activan esta bandera, de manera que la ejecución pueda llevarse de una sola instrucción a un tiempo para examinar el efecto de dicha instrucción sobre los registros y la memoria.
SF (Signo)	Contiene el signo resultante de una operación aritmética (0 = positivo, 1 = negativo).
ZF (Cero)	Indica el resultado de una operación aritmética o de comparación (0 = resultado ≠ cero y 1 = resultado = 0).
AF (Acarreo Auxiliar)	Contiene un acarreo externo del bit 3 en un dato de 8 bits para aritmética especializada.
PF (Paridad)	Indica la paridad, para 0 impar de una operación en datos de 8 bits de bajo orden (más a la derecha). Paridad: existencia de un par de unos en el número.
CF (Acarreo)	Contiene el acarreo de orden más alto (más a la izquierda) después de una operación aritmética, también lleva el contenido del último bit en una operación de corrimiento o rotación.



Estado de las banderas

Representación del estado de las banderas	Representa el contenido del bit de la bandera en el registro
(OF) Overflow	NV = no hay desbordamiento → 0 OV = sí hay desbordamiento → 1
(DF) Dirección	UP = Hacia delante → 0 DN = Hacia atrás → 1
(IF) Interrups	DI = desactivadas → 0 EI = activadas → 1
(SF) Sign	PL = positivo → 0 NG = negativo → 1
(ZF) Zero	NZ = no es cero → 0 ZR = sí lo es → 1
(AF) Auxiliary carry	NA = no hay acarreo auxiliar → 0 AC = sí hay acarreo auxiliar → 1
(PF) Paridad	PO = paridad non → 0 PE = paridad par → 1
(CF) Carry	NC = no hay acarreo → 0 CY = sí hay acarreo → 1
(TF) Trampa	No aparece en el DEBUG ni en TD

Nota: Cada bandera ocupa un bit.

 Demostrar que:

$$01010 \rightarrow \text{derecha } 1 = \frac{01010}{2} \quad \text{y} \quad 01010 \rightarrow \text{izquierda } 1 = 01010 \times 2$$

DEBUG

Permite ejecutar un programa en modo de *paso sencillo* (un paso a la vez), de manera que puede verse el efecto de cada instrucción sobre las localidades de memoria y los registros con la idea de corregir un programa desde el punto de vista de comportamiento, es decir, que realice correctamente la tarea para el que fue hecho.

Contiene un intérprete, es decir, al DEBUG se le dan órdenes por medio de **COMANDOS**. No hace distinción entre mayúsculas o minúsculas y maneja números hexadecimales por default.

Intérprete: Es independiente del Sistema Operativo y corre en *casi* cualquier plataforma; compila línea por línea y es más lento en ejecutarse.

COMANDOS	
A	Ensamblar instrucciones simbólicas y pasarlas a código máquina (proceso de ensamblar).
D	Mostrar el contenido de un área de memoria.
E	Introducir datos en memoria, iniciando en una localidad específica.
G	Correr el programa ejecutable que se encuentra en memoria.
N	Nombrar un programa. Cargar un programa.
P	Proceder o ejecutar un conjunto de instrucciones relacionadas.
Q	Salir de sesión con DEBUG.
R	Mostrar el contenido de uno o más registros.
T	Rastrear la ejecución de una instrucción.
U	"Desensamblar" código de máquina y pasarlo a código simbólico (Ingeniería Inversa)
W	Escribir o grabar un programa en disco.
L	Cargar un programa a memoria.

Nota: Para información sobre los comandos usar - ? ↵

Debug no distingue entre letras minúsculas y mayúsculas, se ponen espacios sólo para separar parámetros en un comando.

Ejemplo:

D DS : 200 Muestra el área de memoria iniciando en el
 DDS : 200 desplazamiento 200_h en el segmento de datos (DS)
 dds : 200

Segmento : desplazamiento

Despliegue

Dirección Segmento	---Representación Hexadecimal---	---ASCII---
XXXX:XX10	...XXXX...XXX-XXX...XXXXXXX	(texto)

Cada línea despliega 16 bytes de memoria. El área de representación hexadecimal muestra dos caracteres hexadecimales por cada byte seguidos por un espacio en blanco por legibilidad.


Si se necesita localizar el byte en el desplazamiento XX13_h, iniciar con XX10_h y contar 3 bytes sucesivos a la derecha.

$\begin{matrix} & & & 13h \\ & & & \text{XX} \\ \text{XX} & \text{XX} & \text{XX} & \text{XX} \\ 10h & 11h & 12h & 13h \end{matrix}$

-D DS: 200 L5 (despliega cinco elementos a partir de la dirección de memoria dada).

Instrucción de Máquina:		Código Simbólico	Explicación
Esto es el código objeto, archivo intermedio a ser ejecutado.			
B8 23 01	mov ax, 0123	Mueve un 0123 al registro ax.	
05 25 00	add ax, 0025	Suma al registro ax el número 0025.	
89 C3	mov bx, ax	Mueve el contenido de ax a bx.	
01 C3	add bx, ax	Suma el registro ax a bx.	
89 D9	mov cx, bx	Mueve el contenido de bx a cx.	
29 C0	sub ax,ax	Resta ax a ax.	
90	Nop	No hagas nada (<i>No opera</i>)	

La instrucción cambia dependiendo del tipo de direccionamiento.

 Verificar en el DEBUG que el siguiente código, basado en instrucciones de máquina, corresponda al código simbólico expuesto anteriormente, sacar las conclusiones correspondientes según los resultados observados.

```

C:/DEBUG ←
-E CS: 100 B8 23 01 05 25 00
-E CS: 106 89 C3 01 C3 89 D9
-E CS: 10C 29 C0 90 90
.
.
.
-U 100 10E
  
```

Diferencias entre comando, instrucción y directiva

COMANDO	INSTRUCCIÓN	DIRECTIVA
Se ejecuta sobre un intérprete.	Se ejecuta sobre el procesador.	Se ejecuta sobre compilador o ensamblador. Le dice al ensamblador cómo ensamblar (<i>lo dirige</i>).
No ocupa memoria.	Sí ocupa memoria	No genera memoria.
Tiempo de ejecución: "Tiempo Real"	Tiempo de ejecución: Tiempo de la corrida.	Tiempo de Ejecución: Tiempo de compilación o Tiempo de ensamblado.

SET DE INSTRUCCIONES

Operaciones lógicas y aritméticas

Lógicas: AND, NEG, NOT, OR, TEST, XOR

Aritméticas: ADC, ADD, DIV, IDIV, MUL, IMUL, SUB

♣ Instrucción AND

Operación: Realiza una operación lógica AND sobre los bits de cada operando. Los operandos pueden ser de longitud byte, palabra o palabra doble, mismas que AND compara bit a bit. Si ambos bits son uno, el bit en el primer operando es puesto en uno, de otra forma el bit es puesto en cero. Afecta las banderas: CF(0), OF(0), PF, SF, ZF.

Código Fuente:

```
and destino, fuente
and {reg/mem}, {reg/mem/inmediato}
    Primer operando      Segundo operando
```

Código Objeto: Tres formas

- Reg/mem con registro /001000 dw / mod reg / m /
- ...

♣ Instrucción NEG

Operación: Convierte un numero binario de positivo a negativo y viceversa (complemento a dos).

Código Fuente:

```
NEG {reg/mem}
NEG destino
```

Ejemplo:

Si AX = 1234_h

Neg AX → AX = EDCC

♣ Instrucción NOT (Negación Lógica)

Operación: Cambia los bits en cero por bits en uno y viceversa.

Código Fuente:

```
NOT {reg/mem},
NOT destino
```

♣ Instrucción OR (Disyunción Lógica)

Operación: Disyunción Lógica.

Código Fuente:

```
OR {reg/mem}, {reg/mem/inmediato},
OR destino, fuente
```

♣ **Instrucción TEST (Examina bits, AND lógica)**

Operación: Examina un campo para una configuración específica de bits tal como AND pero no cambia el operando destino. TEST utiliza un AND lógico para establecer banderas que puede probar con JE, JNE, JZ,... etc. (saltos condicionales).

Código Fuente:

TEST destino, fuente

TEST {reg/mem}, {reg/mem/inmediato}

Primer parámetro Segundo parámetro

Banderas afectadas:

Pone en cero OF(0) y afecta PF, SF, ZF.

♣ **Instrucción XOR (Disyunción OR exclusiva)**

Operación: Si los bits del primer y segundo operando tienen el mismo valor, el bit en el primer operando se pone en cero, si los operandos son diferentes el bit en el primer operando se pone en uno.

Código Fuente:

XOR {reg/mem}, {reg/mem/inmediato}

XOR destino, fuente

♣ **Instrucción ADC (Suma con acarreo)**

Operación: Por lo general es usado en suma de múltiples palabras binarias para acarrear un bit 1 al siguiente paso de la aritmética.

ADC suma el contenido CF (0/1) al primer operando y después suma el segundo operando al primero, al igual que ADD.

Código Fuente:

ADC {reg/mem},{reg/mem/inmediato}

Ejemplo:

mov ax, 8000_h ; 1000 0000 0000 0000 en binario

add ax, 8000_h

mov bx, 0000

adc bx, 0001

1000 0000 0000 0000	→ CF = 1 ; OF = 1
<u>1000 0000 0000 0000</u>	
1 0000 0000 0000 0000	

♣ Instrucción ADD

Suma números binarios.

Código fuente:

ADD {reg/mem}, {reg/mem/ inmediato}

ADD destino, fuente

- **Código de ejemplo 1**

title Demostración de mov

model small

stack 256

dataseg

exCode db 0

speed db 99

codeseg

start:

mov ax,@data

mov ds,ax

mov ax,1

mov bx,2

mov cx,3

mov dx,4

mov ah,[speed]

mov si,offset speed

exit:

mov ah,04ch

mov al,[exCode]

int 21h

end Start

- **Código de ejemplo 2**

title demostracion de and or y xor

model small

stack 256

dataseg

exCode db 0

sourceWord dw 0abh

wordmask dw 0cfh

codeseg

Start:

mov ax,@data

mov ds,ax

mov ax,[sourceWord]

```

mov bx,ax
mov cx,ax
mov dx,ax
and ax,[wordMask]
or bx,[wordMask]
xor cx,[wordMask]
xor dx,dx

```

Exit:

```

mov ah,04ch
mov al,[exCode]
int 21h
end Start

```

- **Código de ejemplo 3**

title "demostración de add, sub, inc, dec"

```

model small
stack 256
dataseg
exCode db 0
count dw 1
codeseg

```

Start:

```

mov ax,@data
mov ds,ax
mov ax,4
mov bx,2
add ax,bx
mov cx,8
add cx,[count]
add[count],cx
inc [count]
dec [count]
inc ax
dec cx


```

Exit:

```

mov ah,4ch
mov al,[exCode]
int 21h
end start

```

 Realizar un programa que realice la siguiente suma empleando la instrucción ADC

```

  A F C D E 1 2 3 4 A B C D E 1 2 3 C F F
+ 3 2 1 0 0 1 1 1 1 4 4 4 4 3 3 3 F 0 1

```


♣ **Instrucción DIV (División entera, sin signo)**

Toma el divisor y el dividendo sin signo. DIV trata el bit de más a la izquierda (el bit más significativo) como un bit de dato, no como bit de signo. La división entre cero provoca una interrupción de división entre cero.

TAMAÑO (numerador)	DIVIDENDO (Operando 1)	DIVISOR (Operando 2)	Cociente	Residuo
16 bits	AX	8 bits reg/mem	AL	AH
32 bits	DX:AX Parte alta: parte baja	16 bits reg/mem	AX	DX
64 bits	EDX:EAX	32 bits reg/mem	EAX	EDX

Ejemplo:

- a) DIV Bh
- b) DIV CX
- c) DIV ECX

Código Fuente:

DIV {mem/reg}
Denominador

♣ **Instrucción IDIV (División entera, con signo)**

Toma el divisor y el dividendo con signo. IDIV trata el bit de más a la izquierda como el signo (0 positivo, 1 negativo). La división entre cero provoca una interrupción de división entre cero. (Se utiliza la tabla de DIV).

Código Fuente:

IDIV {reg/mem}
IDIV fuente

♣ **Instrucción MUL (Multiplicación entera, sin signo)**

Multiplica un factor sin signo por otro factor sin signo, MUL trata el MSB (el bit más significativo) como un bit de dato y no como signo negativo. Sólo multiplica enteros positivos.

TAMAÑO (factor)	multiplicando (factor 1)	multiplicador (factor 2)	producto
8 bits	AL	8 bits reg/mem	AX
16 bits	AX	16 bits reg/mem	DX:AX
32 bits	EAX	32 bits reg/mem	EDX:EAX

Banderas afectadas: CF y OF porque hay números que afectan a la mantisa.

Código Fuente:

MUL {reg/mem}

Ejemplo:

5*2 = A

XXXX: 100 mov al, 5

XXXX: 103 mov cl, 2

XXXX: 106 mul cl

AX= 000A

Operación de 16 bits

A125 X 25

mov ax, a125

mov bx, 0025

mul bx ; AX * BX = DX : AX

♣ Instrucción IMUL (Multiplicación entera, con signo)

Multiplica un factor con signo por otro factor con signo; IMUL trata el bit más significativo como el signo (0 = positivo, 1 = negativo). (Se utiliza la tabla de MUL).

Ejemplo:

title "demostración de mul, div, imul, idiv"

model small

stack 256

dataseg

exCode db 0

opByte db 8

opWord dw 100

sourceByte db 64

sourceWord dw 4000

codeseg

Start:

mov ax, @data

mov ds, ax

mov al, [opByte]

mul [sourceByte]

mov ax, [opWord]

mul [sourceWord]

mov ax, [opWord]

mul ax

mov ax, [opWord]

div [sourceByte]

```

mov ax,[opWord]
mov dx,[opWord]
div [sourceWord]
mov al,[sourceByte]
cbw
mov ax,[sourceWord]
cwd

```

Exit:

```

mov ah,04ch
mov al,[exCode]
int 21h
end start

```

♣ El operador PTR (pointer)

El operador PTR se emplea cuando no hay algún parámetro que determine la longitud de una instrucción. El operador PTR es una directiva que dimensiona la cantidad de memoria que se va a utilizar.

Ejemplo:

```

A 100
      apuntador
100 mov ax, [11A] ;   ax=2314
103 add ax, [11C] ;   ax=2319
107 add ax, 25      ;   ax=233E

      (parámetro)
10A mov [11E], ax
10D mov Word PTR [120], 25
113 mov byte PTR [120], 30
118 NOP
119 NOP
11A DB 14 23
      Define un byte
11C DB 05 00
11E DB 00 00
120 DB 00 00 00

```

También sirve para tomar sólo una parte de un objeto de dimensión diferente:







```

Tabla DW 10 DUP (0) ; 10 palabras a 0
MOV AL, BYTE PTR tabla ; toma el byte del primer elemento de la tabla

```

 Implementar y ejecutar el programa anterior y sacar las conclusiones correspondientes.

Nota: Por cuestiones de arquitectura NO EXISTEN movimientos entre memoria, ya que no hay un bus que conecte memoria con memoria.

TIPOS DE DIRECCIONAMIENTO			
TIPO	INSTRUCCIÓN	FUENTE	DESTINO
Registro	mov ax, bx	BX 	AX
Inmediato: el dato está en el mismo código	mov bl, 3A _h	3A _h 	BL
Directo	mov [1234 _h], ax	AX 	[1234] _h
Indirecto por registros	mov [bx], ax contenido AX en lo que apunte BX. Bx=0300h	AX 	[0300] _h
Base más índice	mov [bx+si], ax si bx =0300h si = 0200h y ax =0633h	AX Hay un 0633h en la dirección 0500h 	0633 _h
Relativo por registros	mov[bx+4], ax	AX	[bx+4]
Destino	mov[ebx+4*esi],ax	AX 	[ebx+4*esi]

Ejemplos de direccionamiento:

INMEDIATO	DIRECTO	INDEXADO O INDIRECTO	
mov ax, 123 add ax, 120 sub ax, 14	Turbo Debugger: mov ax, valor1 add ax, valor2 sub ax, valor3	Turbo Debugger: valor1 dw, 123 valor2 dw, 210 valor3 dw 14	DEBUG: 0000 dw 123 0002 dw 210 0004 dw 14 mov bx, 0 ; inicializando bx mov ax, [bx] add bx, 2 add ax, [bx] ; suma a ax y el valor de la localidad de memoria a la que apunta bx add bx, 2 sub ax, [bx]

Nota: Diferencia entre la directiva DW y el comando E:


- DW: directiva que coloca el dato en memoria.
- E: comando que coloca el dato directo en memoria sin pasar por el ensamblador.


Diferencia entre nombre y etiqueta

- ♣ Etiqueta: hace referencia a instrucciones.
- ♣ Nombre: hace referencia a variables y constantes.

Programa para introducir cadena de caracteres

Turbo Debugger	DEBUG
<p>Cad1 db 'hola mundo' ; cadenas se definen como byte</p> <p>cad2 db ' ' ; cadenas se definen como byte</p> <p>lea bx, cad1 ; lea obtiene la dir real de cad1 y lo pasa a BX</p> <p>lea si, cad2</p> <p>mov ah, [bx]</p> <p>mov [si], ah</p> <p>add bx, 1</p> <p>add si, 1</p>	<p>200 db 'hola mundo'</p> <p>20a db ' ' ; cadenas se definen como byte</p> <p>214 mov bx, 200 ; poniendo apuntador en la dir 200</p> <p>xxx mov si, 20a</p> <p>mov ah, [bx]</p> <p>mov [si], ah</p> <p>add bx, 1</p> <p>add si, 1</p> <p>Nota: el texto en negritas se repite 10 veces</p>

 Ensamblar el programa anterior en DEBUG, y realizar los cambios correspondientes; convertir las letras mayúsculas a minúsculas y viceversa.

 Realizar el siguiente programa:

Siendo A = [1,2,3,4,5,6] y B = [A,B,C,D,E,F], obtener C = A + B considerando las siguientes condiciones:

- A = vive en la dirección 240
- B = vive en la dirección 100
- C = vivirá en la dirección 330 y de inicio deberá contener a B.

Se deberá realizar el programa en cuatro versiones las cuales serán:

1. Los datos de los vectores deben introducirse con el comando E.
2. Los datos deben introducirse desde programa con una directiva.
3. Los datos deben introducirse desde programa con una instrucción.
4. Los vectores se generarán.

Problema de análisis

Ejemplo 1: A partir del código que se presenta comentar cada instrucción, obtener qué operación efectúa el programa y el resultado de la operación en hexadecimal.

100	mov al, 1
102	mov bx, 1
105	mov cx, 5
108	mul bl
10A	inc bl
10C	loop 108
10E	mov [230], ax
111	neg ax
113	add ax, [230]
117	mov al, 1
119	mov bx, 1
11C	mov cx, 3
11F	mul bl
121	inc bl
123	loop 11F
125	add [230], al
129	nop

Nota: los datos son en hexadecimal.

Código Fuente:

INC

Inc {reg, mem}

{reg/mem}= {reg/mem} + 1

Inc destino

LOOP: funciona en conjunto con el registro CX, que contiene en número de repeticiones a realizar. LOOP hace un lazo que repite un número de CX veces.

Nota: CX se decrementa en uno automáticamente cuando se ejecuta la instrucción loop.

Código fuente:

Loop {dir}

Problema de síntesis

Ejemplo 2: Encontrar la inversa de la matriz de 2X2 que se presenta a continuación, mostrar mapa de memoria, código, comentar las instrucciones, indicar cómo se ordena la matriz resultante en memoria, poner los comandos para ver el código y el comando para poder ver los datos de la matriz original sobre el debug.

$$M = \begin{bmatrix} 5 & 4 \\ 4 & 4 \end{bmatrix}$$

$$\text{Nota: } Y = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \frac{1}{AD - CB} \begin{bmatrix} D & -B \\ -C & A \end{bmatrix}$$

 Realizar una versión del programa anterior diferente a la elaborada en clase.

Ejemplo 3: Evaluar

$$Y[n] = \sum_{n=0}^3 x[n] h[n-1] , \text{ donde:}$$

$$x[n] = [1 \ 2 \ 3 \ 4]$$

$$h[n] = [10 \ 11 \ 12 \ 13]$$

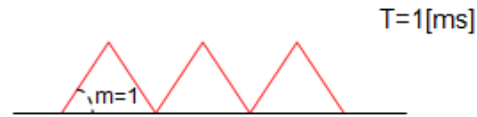
para

$$n < 0 \begin{cases} x[n] = 0 \\ h[n] = 0 \end{cases}$$

Ejemplo 4: Si $x = 2$, evaluar la siguiente ecuación.

$$y = x^2 + \frac{3}{2}x + 7$$

Ejemplo 5: Generador de Diente de Sierra, considere que las instrucciones tardan en ejecutarse; considere la siguiente señal:



La salida de la señal está en AX.

```
etq2 mov ax, 00
      mov cx, 49
etq1 add ax, 01
      loop et1
      jmp etq2
```

Ejemplo 6: Transpuesta de un matriz.

$$WILMA = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} \rightarrow WILMA^T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

```
E 200 1 2 3 4 1 2 3 4 1 2 . . .
A 100
100 mov cx, 4
103 mov si, 210
106 mov bx, 200
109 mov [230], bx
10D jmp 11B
10F mov bx, [230]
113 inc bx
114 mov cx, 4
117 mov [230], bx
11B mov al, [bx]
11D mov [si], al
11F add bx, 4
122 add si, 1
125 loop 11B
127 jmp 10F
129 nop
```

Nota: Encontrar la forma de concluir el programa.

📖 Realizar un programa para hacer las siguientes operaciones.

1. Suma de las siguientes matrices.

$$\text{MAXWELL} = \begin{bmatrix} 4 & A & B & C \\ 5 & 3 & F & 9 \\ 3 & 2 & 1 & 0 \\ 0 & 4 & 3 & B \end{bmatrix} + \begin{bmatrix} 3 & 2 & 1 & A \\ B & F & 1 & C \\ A & A & 1 & C \\ F & F & F & F \end{bmatrix}$$

2. De la matriz resultante obtener la transpuesta

$$\text{MAXWELL}^T = \begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & \tilde{N} \\ D & H & L & O \end{bmatrix}$$

3. De la matriz transpuesta:
 - ✓ sumar las columnas
 - ✓ sumar los renglones
 - ✓ sumar la diagonal principal. A+F+K+O
 - ✓ obtener la suma completa de la matriz. A+B+C+...
4. Multiplicar la matriz transpuesta por el siguiente vector.

$$\text{FARADAY}(4 \times 1) = \text{MAXWELL}^T * \begin{bmatrix} 1 \\ 2 \\ A \\ B \end{bmatrix}$$

5. “Flípear” FARADAY sobre la misma área de memoria.

$$\begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \end{bmatrix} \rightarrow \begin{bmatrix} X4 \\ X3 \\ X2 \\ X1 \end{bmatrix}$$

Hacer los correspondientes mapas de memoria

Realizar un programa que realice la siguiente suma de matrices, considerando que la suma debe elaborarse en un ciclo de 16 veces:

$$[\text{HILBERT}] = [\text{AMPERE}] + [\text{BIO - SAVAT}] + [\text{FOURIER}] + [\text{LAPLACE}] + [\text{NEWTON}] + [\text{ARISTÓTELES}] + [\text{EINSTEIN}] + [\text{PASCAL}] + [\text{LEIBNITZ}]$$

donde cada matriz es de 4 x 4.

Ejemplo 7: Obtener un programa general que obtenga la aproximación de n términos del $\sinh(x)$; resolver para cuando $x=1$ y 9 términos.

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!}$$

$$\sinh(x) = \sum_{n=0}^4 \frac{x^{(2n+1)}}{(2n+1)!}$$

```

mov byte PTR [216], 5
mov cx, [216]
mov byte PTR [200], 1
mov word PTR [214], 0
mov dl, 00
dir3: mov al, dl
      mov [216], cx
      mov bx, 02
      mul bl
      add al, 1
      mov cl, al
      mov al, 1
dir1:  mul byte PTR [200]
      loop dir1
      mov [210], ax
      mov al, dl
      mov bx, 02
      mul bl
      add al, 01
      mov cx, ax
      mov bx, 1
      mov al, 1
dir2:  mul bl
      inc bl
      loop dir2
      mov [212], ax
      mov ax, [210]
      mov bx, [212]
      div bl
      add [214], al
      mov cx, [216]
      inc dl
      loop dir3
      nop ó int 20

```

COMANDOS ÚTILES DEL DOS

type {filespec}

Despliega el archivo solicitado.

type {filespec} | more

Detiene el despliegue en pantalla del archivo solicitado para poder verlo por partes. También se puede usar more como en unix.

dir [filespec]

Muestra el árbol de archivos.

cd {path}

Cambia el directorio actual al path.

mkdir {name}

Crea un Nuevo directorio en el path actual llamado name.

rmdir {name}

Borrar el directorio con nombre name (debe estar vacío).

del {filespec}

Borra los archivos que correspondan a filespec usando expresiones regulares.

copy {fromfile} {tofile}

Copiar de un archivo a otro. Si el archivo nuevo no especifica una extensión, la nueva copia tendrá la misma que el viejo archivo.

ren {fromfile} {tofile}

Renombrar un archivo de fromfile a file. No crea un nuevo archivo.

TASM, TURBO DEBUGGER

Para crear un programa:

- ♣ Editor para crear el programa fuente. (Editor de texto plano es aquel que genera únicamente código ASCII).
- ♣ Ensamblador es un programa que “traduce” el programa fuente a un programa objeto (TASM).
- ♣ Enlazador o Linker que genere el programa ejecutable a partir del programa objeto (TLINK).

ENSAMBLADOR

- TASM Turbo ensamblador
- MASM Macro ensamblador de Microsoft

La extensión usada para que el ensamblador (TASM y MASM) reconozca los programas fuente en ensamblador es .ASM una vez traducido el programa fuente, el (MASM y TASM), crea un archivo con la extensión .OBJ, este archivo contiene un “formato intermedio” del programa, llamado así porque aún no es ejecutable pero tampoco es ya un programa en lenguaje fuente. El enlazador genera a partir de un archivo .OBJ o la combinación de varios de estos archivos, un programa ejecutable, cuya extensión es usualmente .EXE, aunque también puede ser .COM dependiendo de la forma en que fue ensamblado.

Nota: se compila un lenguaje de alto nivel, se ensambla un lenguaje de bajo nivel. DEBUG es un emulador del microprocesador que contiene un ensamblador y un intérprete.

1. **EL PASO DE ENSAMBLAJE** consiste en la traducción del código fuente en código objeto y la generación de un archivo intermedio **.OBJ** (objeto). Una de las tareas del ensamblador es calcular el desplazamiento de cada elemento en el segmento de datos y de cada instrucción en el segmento de código. El ensamblador crea un encabezado al frente del módulo .OBJ generado; parte de éste tiene información acerca de direcciones incompletas. El módulo .OBJ aún no está en forma ejecutable.
2. El **PASO DE ENLACE** implica convertir el módulo .OBJ en un módulo de código máquina .EXE (ejecutable).
3. El último **PASO ES CARGAR** el programa para su ejecución ya que el cargador conoce en dónde está el programa a punto de ser cargado, puede completar las direcciones indicadas en el encabezado que están incompletas. Esto se realiza con un ente de software llamado cargador.

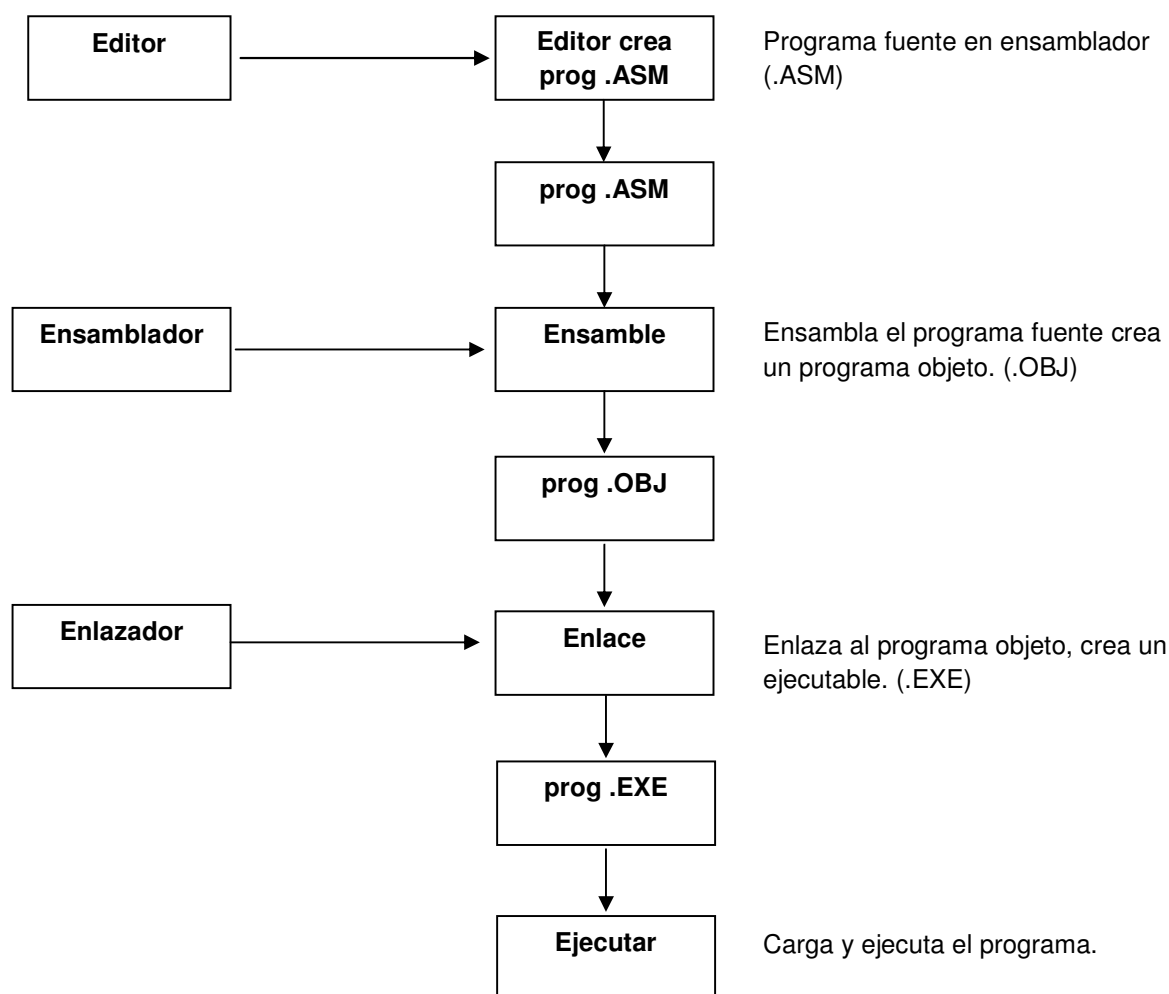


Tabla de símbolos

A cada una de las partes de una línea de código en ensamblador se le conoce como *token* (cada parte de la línea de una instrucción) por ejemplo:

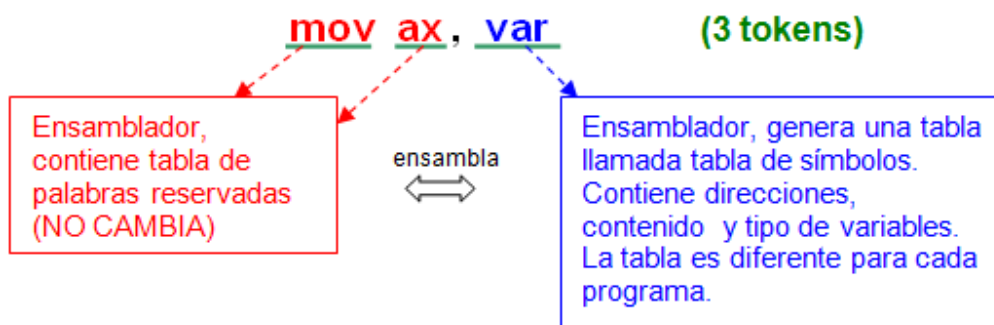
mov ax, var

Tenemos 3 tokens, la instrucción mov, el operando ax y el operando var.

Lo que hace el ensamblador para generar el código objeto es leer cada uno de los tokens y buscarlo en una tabla interna de “equivalencias”, conocida como *tabla de palabras reservadas* (la contiene el ensamblador) que es donde se encuentran todos los significados de los nemónicos que utilizamos como instrucciones.

Siguiendo este proceso, el ensamblador lee mov, lo busca en su tabla y al encontrarlo lo identifica como una instrucción del procesador, así mismo lee AX y lo reconoce como un registro del procesador, pero al momento de buscar el token VAR en la tabla de palabras reservadas no lo encuentra y entonces lo busca en la tabla de símbolos (la genera el ensamblador), que es una tabla donde se encuentran los nombres de las variables,

constantes y etiquetas, utilizadas en el programa donde se incluye su dirección en memoria y el tipo de datos que contiene.



Algunas veces el ensamblador se encuentra con algún token no definido en el programa, lo que hace en estos casos es dar una segunda pasada por el programa fuente para verificar todas las referencias a ese símbolo y colocarlo en la tabla de símbolos. Existen símbolos que no serán encontrados, ya que no pertenecen a ese segmento y el programa no sabe en qué momento entra en acción. El enlazador, el cual crea la estructura que necesita el cargador para que el segmento y el token sean definidos cuando se carga el programa y antes de que el mismo sea ejecutado.

Ensamblador de dos pasadas

Muchos ensambladores dan dos pasadas al programa fuente a fin de resolver referencias hacia delante (o posteriores) a direcciones que aún no se encuentran en el programa. Durante la pasada uno, el ensamblador lee todo el código fuente y construye una tabla de símbolos de nombres y etiquetas usadas en el programa, esto es, nombres de campos, datos y etiquetas del programa y sus localidades relativas (desplazamientos) dentro del segmento. La pasada uno, mientras que TASM lo hace en la pasada dos.

Durante la pasada dos, el ensamblador usa la tabla de símbolos que construyó en la pasada uno.

Ahora que “conoce” la longitud y posiciones relativas de cada campo de datos e instrucción puede completar el código objeto por cada instrucción. Después produce, si se solicita, los diferentes archivos:

- ♣ OBJETO (.obj)
Archivo.obj
- ♣ DE LISTADO (.lst)
archivo.lst: contiene las direcciones resueltas y los errores sintácticos
- ♣ MAP
archivo.map: contiene la dirección inicial y final y la longitud de cada segmento

Un problema potencial en la pasada uno es una referencia hacia adelante: una instrucción de salto en el segmento de código puede referenciar a una etiqueta, pero el ensamblador aún no ha encontrado su definición. El MASM construye el código objeto con base en lo que supone es la longitud de cada instrucción generada en lenguaje máquina. Si existen diferencias entre la pasada uno y la pasada dos con respecto a la longitud de una instrucción, MASM envía un mensaje de error “*Phase error between passes*”.

Tales errores son relativamente raros.

El enlazador realiza las siguientes funciones:

- ♣ Si se pide, combina más de un módulo ensamblado de forma separada en un programa ejecutable, como dos o más programas en ensamblador o en programa ensamblador con un programa en C.
- ♣ Genera un módulo .EXE y lo inicializa con instrucciones especiales para facilitar su subsecuente carga para ejecución. (Esto en el espacio reservado de la 0 a la 100).

Una vez que se han enlazado uno o más módulos .OBJ en un módulo .EXE, puede ejecutar el módulo .EXE cualquier número de veces.

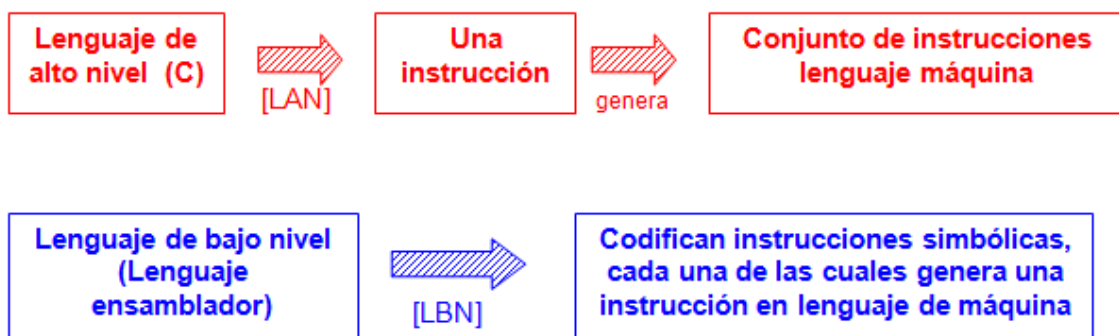
Siempre que necesite realizar un cambio al programa, debe corregir el programa fuente, ensamblarlo en otro módulo .OBJ y enlazar el módulo .OBJ en módulo .EXE.

Se puede generar un archivo. MAP que indica la ubicación relativa y el tamaño de cada segmento y cualquier error que LINK haya encontrado.

Mapa de Enlace: aparece en un archivo .MAP, se puede abrir con cualquier edito de texto plano.

START	STOP	LENGTH	NAME	CLASS
00000h	0003Fh	0003Fh	stacksg	stack
00040h	00045h	00006h	datasg	data
00050h	00063h	00014h	codesg	code

DIFERENCIA ENTRE LENGUAJE DE ALTO NIVEL Y LENGUAJE DE BAJO NIVEL

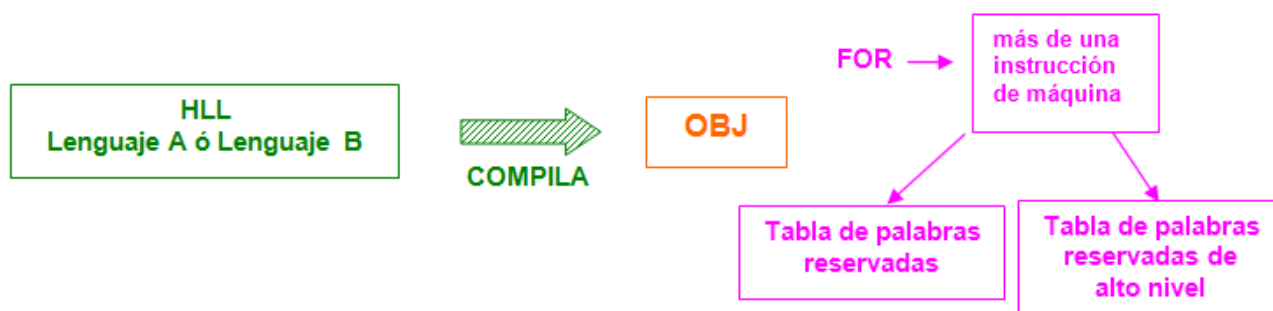


LAN: Utiliza un compilador para traducir el código fuente a código máquina → objeto.

LBN: Utiliza un ensamblador para realizar la traducción de lenguaje ensamblador a código máquina.

Un programa enlazador para ambos niveles completa el proceso al convertir el código objeto en lenguaje ejecutable de máquina.

Diferencia entre compilador y ensamblador



Lenguaje de Alto Nivel más completo por la capa extra de SOFTWARE.



PROGRAMACIÓN

Palabras reservadas

INSTRUCCIONES	Mov, add, imul, ... etc.
DIRECTIVAS (actúan sobre el ensamblador)	end o segment
OPERACIONES	Far, size, type, ... etc.
SIMBOLOS DEFINIDOS	@data ...

Nota: Datos que debe contener el código fuente.

- ♣ Nombre de la persona que lo elaboró.
- ♣ Fecha del término de la programación.
- ♣ Describir las funcionalidades implementadas.
- ♣ Versión de compilador o ensamblador, según sea el caso, que fue utilizado.
- ♣ Documentar las modificaciones subsecuentes.
- ♣ Agregar los comandos necesarios para el ensamblado (compilado si es que es un lenguaje de alto nivel), y el enlazado, así como las librerías y referencias necesarias para poder obtener el ejecutable.

Identificadores

Un identificador es un nombre que se aplica a elementos en el programa.

- ♣ Nombre: apunta a la dirección de un elemento de tipo dato o variable.
- ♣ Etiqueta: apunta a la dirección de una instrucción.

Miguel Israel Barragán Ocampo

Nota: El primer elemento de este identificador NO debe ser "." o un número. La longitud máxima de un identificador es de 31 caracteres.

Instrucciones

Instrucciones tales como mov, add, nop,... (todas las del set de instrucciones).

Directivas

Directivas que indican al ensamblador que realice una acción específica. Actúan sobre el ensamblador, lo dirigen.

Formato general de programación

Identificador operación [operando (s)] ; comentario
Puede o no llevar []

Máximo 132 caracteres por línea.

Ejemplo:

Id	Operación	Operando(s)	Comentario
Directiva	db	1	; nombre, op
Instrucción	mov	ax, 0	; op dos operandos

DIRECTIVAS PARA LISTA: page y title

Permiten controlar la manera en que un programa ensambla y lista estos enunciados, actúa sólo durante el ensamblado de un programa y no genera código ejecutable de máquina.

- ♣ Page: designa el número máximo de líneas para listar en una página y el número máximo de caracteres de una línea.

Ejemplo: **Page [longitud] [ancho]**

El número de página puede variar desde 10 hasta 255, el número de caracteres por línea desde 60 hasta 132; en caso de omisión los valores serán page 50, 80.

- ♣ Title: Se puede emplear para hacer que un título para un programa se imprima en la línea dos de cada página, en el archivo .lst

Ejemplo: *Title asmordenar* Programa en ensamblador para ordenar los nombres de los clientes.

SEGMENT: Un programa ensamblado en formato .EXE consiste en uno o más segmentos. Esta directiva define los segmentos.

Formato para dar de alta un segmento:

nombre segment [opciones] ; Inicia el segmento

nombre ends ; fin del segmento

Nota: En TASM todos los números son decimales.

OPCIONES Alineación, Combinar, Clase.

Nombre del segmento segment alineación combinar 'clase'

- ♠ **Tipo alineación:** La entrada alineación indica el límite en el que inicia el segmento. Para el requerimiento típico, PARA, alinea el segmento con el límite de un párrafo de manera que la dirección inicial es divisible entre la 16 y 40_h. Por default es PARA.

Ejemplo:

nombre segment para stack 'stack'

- ♠ **Tipo combinar:** La entrada combinar indica si se combina el segmento con otros segmentos cuando son enlazados (TLINK) después de ensamblar (TASM). Los tipos combinar son stack, common, public y la expresión AT.

Ejemplo:

nombre segment para stack 'stack'

public y common combinan de forma separada programas ensamblados.

- ♠ **Tipo clase:** Sirve para agrupar segmentos cuando se enlazan.

'code' para segmentos de código

'data' para segmentos de datos

'stack' para segmentos de pila

Directiva PROC

El segmento de código contiene el código ejecutable de un programa. También tiene uno o más procedimientos definidos como PROC. Un procedimiento que tiene sólo un procedimiento puede aparecer como sigue: (Un programa puede tener varios procedimientos)

Nombre	Operación	Operando	Comentario
nomsegmento	segment	Para	
nomproc	proc	FAR	Un procedimiento cuerpo del segmento de código.
nomproc	Endp		
nomsegmento	ends		

Nota: endp = fin de procedimiento

ends = fin de segmento

end = fin

NEAR: cuando un procedimiento está en el mismo segmento de programa accedido por un desplazamiento.

nombre proc (near)

PUBLIC: Cuando el procedimiento llamado es externo al segmento que hace la llamada y debe usar CALL para introducirlo.

FAR: Para un programa .EXE el procedimiento principal que es el punto de entrada para la ejecución.

Estructura de un programa .EXE

```
Page 60, 132
Title P04ASM1 Estructura de un programa .EXE
stacksg segment para stack 'stack'
...
stacksg ends
;...
datasg segment para 'data'
...
datasg ends
;...
codesg segment para 'code'
Begin proc far
assume ss:stacksg, ds:datasg, cs:codesg
    mov ax, @datasg
    mov ds, ax
    mov ax, 4C00h
    int 21h
begin endp
codesg ends
end Begin
```

Directivas simplificadas de segmentos

Para usar las abreviaturas hay que inicializar el modelo de memoria antes de definir algún segmento.

Formato general:

.model modelo de memoria

El modelo puede ser:

- ♣ TINY
- ♣ SMALL
- ♣ MEDIUM
- ♣ COMPACT
- ♣ LARGE

MODELO	NÚMERO DE SEGMENTOS DE CÓDIGO	NÚMERO DE SEGMENTOS DE DATOS
TINY (programas .com)	1	0
SMALL	1	1
MEDIUM	más de 1	1
COMPACT	1	más de 1
LARGE	más de 1	más de 1

Para utilizar cualquiera de estos modelos para un programa autónomo (esto es, un programa que no esté enlazado con algún otro).

El modelo **TINY** está destinado para uso exclusivo de programas .com, los cuales tienen sus datos, código y pila en un segmento.

El modelo **SMALL** exige que el código quepa en un segmento de 64K y los datos en otro de 64K.

La **directiva .MODEL** genera de forma automática el enunciado `assume` necesario.

Los formatos generales (incluyendo el punto inicial) para las directivas que define los segmentos de la pila de datos y código son:

.stack [tamaño]

.data

.code [nombre]

Para inicializar la dirección del segmento de datos en el DS son:

Mov ax, @data

Mov ds, ax

 Investigar qué implica el @ en:

Mov ax, @data

Mov ds, ax

Definición de datos (variables)

Un elemento de datos puede contener un valor indefinido (no inicializado) o un valor definido o varios valores, pueden ser una cadena de caracteres o un valor numérico.

Formato general:

[nombre]	Dn	Expresión
----------	----	-----------

Directivas:

- ♣ DB (byte)
- ♣ DW (palabra)
- ♣ DD (palabra doble)
- ♣ DF (palabra larga)
- ♣ DQ (palabra cuádruple)
- ♣ DT (diez bytes)

Cada una indica de manera explícita la longitud de la variable definida.

Expresión	fld1	DB	?	; Elemento no inicializado
	fld2	DB	25	; Elemento inicializado
también	fld3	DB	11, 12, 13, 14, 15, 16,...	

para referirse a ellos:

mov ax, fld3+3 (movería el 14)

Duplicación

DW	10	DUP(?)	; 10 palabras no inicializadas
DB	5	DUP(14)	; 5 bytes con 0E 0E 0E
DB	3	DUP(4DUP(8))	; doce 8 (por velocidad se ocupa esta expresión y no "12 DUP 8")

Cadena de caracteres

Usadas para datos descriptivos como nombres de personas y título de páginas definidas como "PC" o 'PC'. El ensamblador traduce las cadenas de caracteres en código objeto en formato ASCII normal.

DB 'Cadena de caracteres'

Ejemplo:

FLD1DB	DB	?	; No se inicializa
FLD2DB	DB	32	; variable decimal
FLD3DB	DB	20 _h	; variable hexadecimal
FLD4DB	DB	01011001 _b	; variable binaria
FLD6DB	DB	'Personal Computer'	; Cadena de caracteres
FLD7DB	DB	'32654'	; Número como cadena
FLD1DW	DW	0FFF0 _h	; variable hexadecimal
FLD3DW	DW	FLD1DW	; variable de dirección
FLD4DW	DW	3,4,7,8,9	; arreglo de variables

Directiva equ (constantes)

No define elementos de datos. Define un valor que el ensamblador puede usar para sustituir en otras instrucciones.

TIMES EQU 10

El nombre TIMES puede ser cualquier nombre aceptable por el ensamblador; siempre que en una instrucción o en otra directiva aparezca la palabra TIMES el ensamblador la sustituye por el valor 10.

Ejemplo:

```
F1ELDA    DB    TIMES DUP(?)
F1ELDA    DB    10 DUP(?)
-COUNTR    EQU    05
    mov cx, countr
después de ensamblado queda:
    mov cx, 05
```

Interrupciones

El modelo de tres capas.

Una de las tareas más importantes de la programación de sistemas incluye el acceso al hardware de la PC. Sin embargo este acceso no ocurre de manera inmediata con el programa actuando directamente sobre el hardware, que sucede, por ejemplo, cuando se accede al procesador en la tarjeta de video. En lugar de ello, el programa puede usar la ROM-BIOS y el DOS para negociar el acceso al hardware. La ROM-BIOS y el DOS son interfaces de software, las cuales fueron creadas específicamente para la administración del hardware.

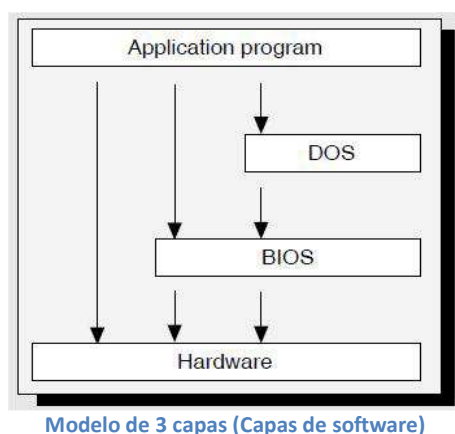
Ventajas de las interfaces del DOS y el BIOS

La mayor ventaja de usar el DOS o el BIOS es que un programa no tiene que comunicarse con el hardware por sí mismo, sino que se llaman las rutinas de la ROM-BIOS que ejecutan la tarea requerida. Después de que la

tarea ha sido completada, la ROM-BIOS regresa el estado de la información al programa como vaya siendo necesitada. Esto ahorra al programador mucho trabajo, ya que el llamado de estas “rutinas” es más rápido que acceder directamente al hardware.

Existe otra ventaja para usar estas interfaces. Las “rutinas” de las interfaces ROM-BIOS y del DOS mantienen al programa aislado de las propiedades físicas particulares del hardware. Esto es muy importante ya que si se quiere programar para todos los tipos de hardware que existen se deberían implementar “rutinas” individuales para cada uno y evidentemente eso requiere mucho tiempo.

Las rutinas de la ROM-BIOS usadas para la salida de video se adaptan a la tarjeta de video residente, así el programa puede llamar a estas “rutinas” sin tener que adaptarse al tipo de tarjeta de video local.



La ROM-BIOS ofrece “rutinas” para acceder a los siguientes dispositivos:

Tarjetas de Video, RAM (Memoria extendida), Discos duros, Puertos Seriales, Puertos Paralelos, Teclado, reloj de tiempo real operado por batería.

Como se puede observar, la ROM-BIOS puede ser vista como una capa que envuelve al hardware. Aun cuando se puede omitir la ROM-BIOS y directamente acceder al hardware, generalmente se deben usar las rutinas del ROM-BIOS pues éstas se encuentran estandarizadas y pueden encontrarse en cualquier máquina. La ROM-BIOS, como su nombre lo indica, es una memoria ROM que está en la tarjeta madre de la máquina. Esta memoria está disponible al momento en que se enciende la máquina.

Interfaz del DOS

Junto con el BIOS, el DOS provee “rutinas” para poder acceder al hardware. Sin embargo, ya que el DOS ve el hardware como dispositivos lógicos y no como dispositivos físicos, las “rutinas” del DOS manejan el hardware de manera diferente. Por ejemplo, la ROM-BIOS maneja los drives como un grupo de tracks y sectores, el DOS maneja estos drives como un grupo de archivos y directorios. Si se quisiera ver los primeros 1000 caracteres de un archivo, primero se debe dar a la ROM-BIOS la ubicación del archivo en el drive. Con las “rutinas” del DOS, simplemente se le instruye al DOS a abrir el archivo en el drive A:, C:, o cualquiera de los dispositivos, y desplegar los primeros 1000 caracteres de este archivo.

El acceso ocurre con frecuencia a través de las “rutinas” del BIOS usadas por el DOS. Sin embargo, algunas veces el DOS también accede el hardware de manera directa, pero no es algo a atender cuando se usa el DOS.

¿Qué “rutinas” debería usar?

Se tienen las opciones de manejar directamente el hardware por programación por las “rutinas” del BIOS y el DOS. Muchas tareas no están soportadas o implementadas por las “rutinas” del DOS o el BIOS. Por ejemplo, si se quiere que la tarjeta de video dibuje círculos o líneas no se encontrarán las “rutinas” apropiadas en el DOS o en el BIOS. Para este caso se debe usar el hardware directamente por programa o en su caso comprar una librería comercial que contenga el código de este programa.

Escogiendo entre el BIOS y el DOS

Cuando se puedan usar las “rutinas” del BIOS o el DOS, habrá que basarse en la situación actual para tomar una decisión. Usar las “rutinas” del DOS si se quiere trabajar con archivos. Si se quiere formatear un diskette, se deberán usar las “rutinas” apropiadas del BIOS. Las “rutinas” del BIOS proveen un mejor control de la pantalla.

Alentando el acceso

Sin embargo, en algunos casos, las “rutinas” del BIOS y del DOS están en desventaja debido a la velocidad lenta de ejecución. Conforme el número de capas de software, con las cuales se debe interactuar antes de que el acceso al hardware ocurra se incrementa el programa, se torna más largo. Si el hardware debe acceder a un programa que lee un archivo a través del BIOS y el DOS, la velocidad de transferencia de datos puede decrecer hasta en un máximo de 80%.

Este problema es causado por la manera en que las capas son manejadas. Antes de que la llamada pueda ser pasada al siguiente nivel, los parámetros deben ser convertidos, la información debe ser cargada desde las tablas internas, y el contenido del buffer debe ser copiado. Al tiempo que se necesita para esto se le llama *overhead*.

Como resultado de esto, cuando se requiere el máximo de la velocidad de ejecución y la programación directa sobre el hardware es relativamente simple, los programadores con frecuencia usan el acceso directo en lugar del BIOS o el DOS.

Las interrupciones son importantes para controlar el hardware, y actúan como la principal forma de comunicación entre un programa y las “rutinas” del BIOS y el DOS.

Interrupciones por Software

Las interrupciones por software llaman a un programa, con una instrucción especial del lenguaje ensamblador, para ejecutar una “rutina” del DOS, BIOS. La ejecución del programa en realidad no se interrumpe; el procesador ve la “rutina” llamada como una subrutina. Después de que la subrutina se ejecuta, el procesador continúa con el programa que hizo la llamada.

Para llamar una “rutina” del DOS o el BIOS usando una interrupción por software, sólo se necesita el número de interrupción desde la cual la rutina puede encontrarse. El que hace la llamada ni siquiera necesita saber la dirección de la “rutina” en la memoria. Estas rutinas están estandarizadas. Por lo tanto, sin importar la versión

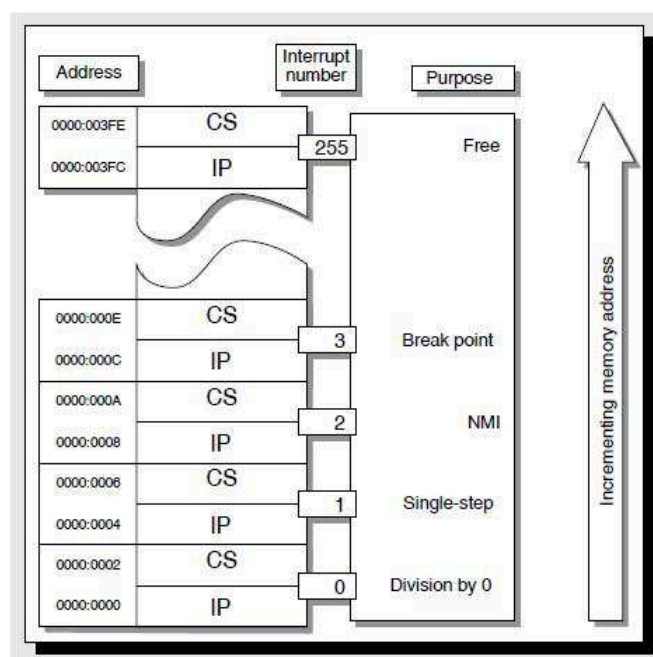
del DOS que se tenga, se tiene la certeza de que haciendo la llamada a la interrupción 21h se puede acceder a las funciones del DOS. El procesador llama al manejador de interrupción usando la tabla de vectores de interrupción, de la cual el procesador toma las direcciones de la función deseada. El procesador usa el número de la interrupción como un índice a esta tabla. La tabla se configura durante el arranque (bootup) del sistema por lo que varios vectores de interrupción apuntan a la ROM-BIOS. Lo siguiente ilustra la ventaja de usar interrupciones. Un fabricante de máquinas que quiere producir PC's compatibles con IBM, no puede copiar la ROM-BIOS entera de la de IBM. Sin embargo, el fabricante tiene permitido implementar las mismas funciones en su ROM-BIOS, aun cuando las "rutinas" de la BIOS están codificadas de manera diferente internamente. Así, las "rutinas" del BIOS son llamadas usando la misma interrupción que IBM usa y espera parámetros en el mismo procesador de registros. Pero las rutinas que proveen las funciones están organizadas de manera diferente que las rutinas de IBM.

Tabla de Vectores de Interrupción

Actualmente el 80X86 tiene 256 interrupciones posibles numeradas de la 0 a la 255. Cada interrupción tiene asociada una "rutina" de interrupción para manejar una condición en particular. Para organizar las 256 interrupciones, la dirección de inicio de la interrupción correspondiente está arreglada en una tabla de vector de interrupción.

Cuando una interrupción ocurre, el procesador automáticamente obtiene la dirección de inicio de la rutina de interrupción de la tabla de vectores de servicio. La dirección de inicio de cada rutina de interrupción esta especificada en la tabla en términos de dirección de desplazamiento y la dirección del segmento. Las dos direcciones son de 16 bits (2 bytes) de longitud. Por lo tanto la entrada de la tabla ocupa 4 bytes. La longitud total de la tabla es 256x4 o 1024 bytes (1kB). Debido a que la tabla del vector de interrupción reside en RAM, cualquier programa puede cambiarla. Sin embargo esta tabla la usan la mayor parte del tiempo los programas residentes y los drivers de los dispositivos.

(Cómo ver la versión del BIOS?, debug, -d f000:fff0 l 10, entregará la fecha del BIOS.)



La siguiente tabla muestra las direcciones de varios vectores de interrupción, así como la función que realizan. Esta tabla aplica para todas las máquinas. Muchas de estos vectores de interrupción son sólo montadas en memoria cuando el hardware correspondiente también ha sido instalado. Por ejemplo esto aplica para la interrupción 33h (mouse driver functions) y la interrupción 5Ch (network functions). El termino reservado indica que la interrupción es llamada por cierto componente del sistema (usualmente el DOS), pero el uso de la interrupción nunca fue documentada. En otras palabras, sabemos quién lo usa, pero no sabemos para qué.

Summary Of Interrupts		
No.*	Address*	Purpose
00	000 - 003	Processor: Division by zero
01	004 - 007	Processor: Single step
02	008 - 00B	Processor: NMI (Error in RAM chip)
03	00C - 00F	Processor: Breakpoint reached
04	010 - 013	Processor: Numeric overflow
05	014 - 017	Hardcopy
06	018 - 01B	Unknown instruction (80286 only)
07	01D - 01F	Reserved
08	020 - 023	IRQ0: Timer (Call 18.2 times/sec.)
09	024 - 027	IRQ1: Keyboard
0A	028 - 02B	IRQ2: 2nd 8259 (AT only)
0B	02C - 02F	IRQ3: Serial port 2
0C	030 - 033	IRQ4: Serial port 1
0D	034 - 037	IRQ5: Hard drive

Summary Of Interrupts		
No.*	Address*	Purpose
0E	038 - 03B	IRQ6: Diskette
0F	03C - 03F	IRQ7: Printer
10	040 - 043	BIOS: Video functions
11	044 - 047	BIOS: Determine configuration
12	048 - 04B	BIOS: Determine RAM memory size
13	04C - 04F	BIOS: Diskette/hard drive functions
14	050 - 053	BIOS: Access to serial port
15	054 - 057	BIOS: Cassettes/extended function
16	058 - 05B	BIOS: Keyboard inquiry
17	05C - 05F	BIOS: Access to parallel printer
18	060 - 063	Call ROM BASIC
19	064 - 067	BIOS: Boot system (Ctrl+Alt+Del)
1A	068 - 06B	BIOS: Prompt time/date
1B	06C - 06F	Break key (not Ctrl-C) pressed
1C	070 - 073	Called after each INT 08
1D	074 - 077	Address of video parameter table
1E	078 - 07B	Address of diskette parameter table
1F	07C - 07F	Address of character bit pattern
20	080 - 083	DOS: Quit program
21	084 - 087	DOS: Call DOS function
22	088 - 08B	Address of DOS quit program routine
23	08C - 08F	Address of DOS Ctrl-Break routine
24	090 - 093	Address of DOS error routine
25	094 - 097	DOS: Read diskette/hard drive
26	098 - 09B	DOS: Write diskette/hard drive
27	09C - 09F	DOS: Quit program, stay resident
28	0A0 - 0A3	DOS: DOS is unoccupied
29-2E	0A4 - 0BB	DOS: Reserved
2F	0BC - 0BF	DOS: Multiplexer
30-32	0C0 - 0CB	DOS: Reserved
33	0CC - 0CF	Mouse driver functions
34-40	0D0 - 0FF	DOS: Reserved
41	104 - 107	Address of hard drive table 1
42-45	108 - 117	Reserved
46	118 - 11B	Address of hard drive table 2
47-49	11C - 127	Can be used by programs

Summary Of Interrupts		
No.*	Address*	Purpose
4A	128 - 12B	Alarm time reached (AT only)
4B-5B	12C - 16F	Free: can be used by programs
5C	170 - 173	NETBIOS functions
5D-66	174 - 19B	Free: can be used by programs
67	19C - 19F	EMS memory manager functions
68-6F	1A0 - 1BF	Free: can be used by programs
70	1C0 - 1C3	IRQ08: Realtime clock (AT only)
71	1C4 - 1C7	IRQ09: (AT only)
72	1C8 - 1CB	IRQ10: (AT only)
73	1CC - 1CF	IRQ11: (AT only)
74	1D0 - 1D3	IRQ12: (AT only)
75	1D4 - 1D7	IRQ13: 80287 NMI (AT only)
76	1D8 - 1DB	IRQ14: Hard drive (AT only)
77	1DC - 1DF	IRQ15: (AT only)
78-7F	1E0 - 1FF	Reserved
80-F0	200 - 3C3	Used within the BASIC interpreter
F1-FF	3C4 - 3CF	Reserved
*= All addresses and numbers in hexadecimal notation		

Interacción con el Sistema (USO)

Veamos cómo el DOS y el BIOS, y los diferentes niveles de hardware se comunican para dar a los programas fácil acceso al hardware de la máquina.

Usaremos el teclado como un ejemplo, ya que las interrupciones por hardware y las rutinas del DOS y el BIOS están involucradas. Vamos a seguir el camino de un carácter ingresado desde teclado hasta el programa que lee el carácter ingresado y lo despliega en la pantalla.

Hardware -> Teclado.

El hardware del teclado lo constituye el procesador del teclado. Está conectado al procesador de la PC por medio de un cable. El procesador del teclado monitorea el teclado y reporta cada una de las teclas que son presionadas o liberadas al sistema. El procesador del teclado asigna un número en lugar de un carácter para cada tecla. Las teclas de control tales como Ctrl o Shift, son tratadas como cualquier otra tecla.

Cuando el usuario presiona una tecla, el procesador del teclado pasa el número de la tecla al procesador como un "make code" (código para cuando se presiona la tecla, existe el break code que para cuando se libera la tecla). Cuando el usuario libera la tecla, el procesador pasa un "break code". Aunque ambos usan números entre 0 y 127 para esa tecla, el "break code" incluye el bit 7. Para iniciar la transferencia, el controlador del teclado primero manda una señal de interrupción al controlador de interrupciones, el cual llega a la línea

IRQ2. Si las interrupciones por hardware están habilitadas y no existe una prioridad más alta habilitada de solicitud de interrupción el procesador entonces ejecuta la interrupción 09h.

Manejador del teclado del BIOS

La interrupción 09H es una rutina de la BIOS llamada el manejador del teclado. El procesador del teclado pasa el código de la tecla al puerto 60H usando el cable del teclado. Después llama al manejador de interrupción. De aquí, el manejador del BIOS lee el número de la tecla que fue presionada o liberada. El resto del sistema no puede usar el número de la tecla debido a que los teclados generan diferentes números. Así, el manejador de teclado debe convertir el código en un carácter del conjunto de caracteres ASCII en una forma que el sistema pueda entender.

Cuando se presiona una tecla, este código de tecla es pasado al CPU como un byte. Cuando se libera la tecla, el procesador pasa el código del CPU sumándole un 128. Esto es lo mismo que poner en 1 el bit 7 del byte. El teclado instruye al controlador de interrupciones 8259 al CPU que debe activar la interrupción 09H. Si el CPU responde, alcanzamos el siguiente nivel ya que la rutina del BIOS es controlada a través de la interrupción 09H. Mientras esta rutina está siendo llamada, el procesador del teclado manda el código de la tecla al puerto 60H de la tarjeta del circuito principal usando un protocolo de transmisión asíncrona. La rutina del BIOS checa este puerto y obtiene el número de la tecla oprimida o liberada. Esta rutina entonces genera un ASCII con este código de tecla.

Esta tarea es más complicada de lo que parece debido a que la rutina del BIOS debe probar si es una tecla de control tal como shift o Alt. Dependiendo de la tecla o la combinación de teclas, ya sea un código normal o extendido de ASCII puede ser requerido.

Buffer del Teclado

Una vez que el BIOS determina el código correcto, este código es pasado al buffer de teclado del BIOS de 16 byte, el cual está localizado en el área más baja de la memoria RAM. Si está llena, la rutina hace sonar un bip que informa al usuario que hay un overflow o desbordamiento en el buffer del teclado. El procesador regresa a la tarea que estaba en progreso antes de la llamada a la interrupción 09H.

Interrupción de teclado del BIOS

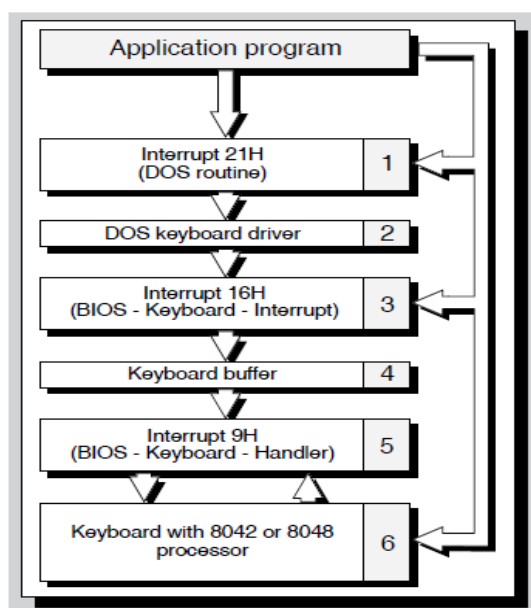
El siguiente nivel, la interrupción 16H del BIOS, lee el carácter en el buffer del teclado y lo pone disponible al programa. Esta interrupción incluye tres rutinas del BIOS para leer caracteres así como el estado del teclado (ejemplo, cuáles teclas de control fueron presionadas), desde el buffer del teclado. Estas rutinas pueden ser llamadas con la instrucción INT.

Nivel de DOS.

El driver del dispositivo del teclado representa el nivel del DOS. Estas rutinas del Dos leen el carácter desde el teclado y guardan el carácter en un buffer usando las funciones del BIOS desde la interrupción 16H. En algunos casos, las rutinas del DOS pueden limpiar el buffer del teclado del BIOS. Si el sistema usa el driver extendido de teclado ANSI.SYS, este teclado puede traducir ciertos códigos (tecla de función F1 por ejemplo) en otros códigos o cadenas. Por ejemplo es posible programar la tecla F10 para que despliegue el comando

DIR en la pantalla. Aunque teóricamente se puede llamar a las funciones de los drivers desde los programas, el DOS usualmente direcciona estas funciones.

El DOS es el nivel más alto al que se puede ir. Se encontrarán las funciones de acceso a teclado en la int 21h del DOS. Estas funciones llaman las funciones del driver, transmiten los resultados y ejecutan otras tareas. Por ejemplo los caracteres y cadenas pueden ser leídas y desplegadas directamente en la pantalla hasta que se presione enter. Estas cadenas son llamadas por un programa y completan un largo proceso.



Acceso a teclado usando el modelo de tres capas

Interrupciones útiles

Status del teclado

INT 21H AH = 0BH

Descripción: La función de esta rutina es detectar si se ha pulsado una tecla.

Uso: Entrada: AH = 0BH
Salida: AL = FF si carácter disponible
AL = 0 si carácter no disponible
Registros afectados: AL

Entrada de un carácter desde teclado

INT 21H AH = 8H

Descripción: La función de esta rutina es esperar un carácter del teclado sin escribirlo por pantalla y almacenarlo en el registro AL en forma de código ASCII.

Uso: Entrada: AH = 8H
Salida: AL = carácter ASCII de la tecla pulsada
Registros afectados: AL

Leer una cadena desde teclado

INT 21H AH = 0AH

Descripción: La función de esta rutina es la de obtener una línea de datos del teclado (que finaliza al pulsar el retorno de carro) y almacenarlos en un área de memoria. Los caracteres son mostrados en la pantalla al ser tecleados.

Uso: Entrada: AH = 0AH
DS contiene la dirección del segmento de memoria en el cual se almacenan los datos introducidos.
DX contiene la dirección del offset de la zona de memoria del segmento anterior en la que se almacenan los datos.
En el primer byte del área debe indicarse el máximo número de caracteres a introducir sin superar 255.
Salida: Ninguna en registro
En el 2do byte del área se almacena el número de caracteres tecleados sin contar el retorno de carro.
Registros afectados: Ninguno

Imprimir un carácter por pantalla INT 21H AH = 2H

Descripción: La función de esta rutina es visualizar un carácter.

Uso: Entrada: AH = 2H
DL contiene el código ASCII del carácter a visualizar.
Salida: Ninguna
Registros afectados: Ninguno

Imprimir un string a la pantalla INT 21H AH = 9H

Descripción: Su función es la de sacar una cadena de caracteres ASCII por pantalla.

Uso: Entrada: AH = 9H
DS contiene el valor de la dirección del segmento del comienzo de la cadena de caracteres a sacar.
DX contiene el offset de dicha cadena en el segmento anterior.
El último byte de la cadena de caracteres debe ser el carácter \$, que no se muestra en pantalla.
Salida: Ninguna
Registros afectados: AX

Posicionar el cursor INT 10H AH = 02H

Entrada: DH = fila (0-24)
DL = columna (0-79)
BH = número de página

Escribir un carácter en pantalla, donde está el cursor INT 10H AH = 0AH

Entradas: BH = número de página
AL = carácter a escribir
Cx = # de veces a escribir el carácter

Leer carácter y atributo de la posición actual del cursor INT 10H AH = 08H

Entradas: BH = número de página
Salidas: AL = carácter leído
AH = atributo del carácter leído

Escribir carácter y atributo en la posición actual del cursor INT 10H AH = 09H

Entradas: BH = número de página
BL = atributo del carácter
CX = número de caracteres a escribir
AL = carácter a escribir

Obtener el tiempo del sistema INT 21h AH=2Ch

Entradas: ninguna
Salidas: CH=hora
 CL = minutes
 DH = segundos
 DL = centésimas de segundo

Atributos:

Color	I R G B	Color	I R G B
Negro	0 0 0 0	Gris	1 0 0 0
Azul	0 0 0 1	Azul Claro	1 0 0 1
Verde	0 0 1 0	Verde Claro	1 0 1 0
Cian	0 0 1 1	Cian Claro	1 0 1 1
Rojo	0 1 0 0	Rojo Claro	1 1 0 0
Magenta	0 1 0 1	Magenta Claro	1 1 0 1
Café	0 1 1 0	Amarillo	1 1 1 0
Blanco	0 1 1 1	Blanco Brillante	1 1 1 1

Atributo:	Fondo	Frente
BL R G B	I R G B	
Número de Bit:	7 6 5 4	3 2 1 0

Bit 7: (BL) Establece intermitencia
 Bit 6-4: determina el fondo de la Pantalla
 Bit 3: (I) Establece la intensidad Alta
 Bits 2-0 Determinan el frente o Primer plano (para el Carácter que será Desplegado).

Tabla de scancodes (en hexadecimal):

Key	Down	Up	Key	Down	Up	Key	Down	Up	Key	Down	Up
Esc	1	81	[{	1A	9A	. <	33	B3	center	4C	CC
1 !	2	82] }	1B	9B	. >	34	B4	right	4D	CD
2 @	3	83	Enter	1C	9C	/ ?	35	B5	+	4E	CE
3 #	4	84	Ctrl	1D	9D	R shift	36	B6	end	4F	CF
4 \$	5	85	A	1E	9E	* PrtSc	37	B7	down	50	D0
5 %	6	86	S	1F	9F	alt	38	B8	pgdn	51	D1
6 ^	7	87	D	20	A0	space	39	B9	ins	52	D2
7 &	8	88	F	21	A1	CAPS	3A	BA	del	53	D3
8 *	9	89	G	22	A2	F1	3B	BB	/	E0 35	B5
9 (0A	8A	H	23	A3	F2	3C	BC	enter	E0 1C	9C
0)	0B	8B	J	24	A4	F3	3D	BD	F11	57	D7
- _	0C	8C	K	25	A5	F4	3E	BE	F12	58	D8
= +	0D	8D	L	26	A6	F5	3F	BF	ins	E0 52	D2
Bksp	0E	8E	; :	27	A7	F6	40	C0	del	E0 53	D3
Tab	0F	8F	' "	28	A8	F7	41	C1	home	E0 47	C7
Q	10	90	~	29	A9	F8	42	C2	end	E0 4F	CF
W	11	91	L shift	2A	AA	F9	43	C3	pgup	E0 49	C9
E	12	92	\	2B	AB	F10	44	C4	pgdn	E0 51	D1
R	13	93	Z	2C	AC	NUM	45	C5	left	E0 4B	CB
T	14	94	X	2D	AD	SCRL	46	C6	right	E0 4D	CD
Y	15	95	C	2E	AE	home	47	C7	up	E0 48	C8
U	16	96	V	2F	AF	up	48	C8	down	E0 50	D0
I	17	97	B	30	B0	pgup	49	C9	R alt	E0 38	B8
O	18	98	N	31	B1	-	4A	CA	R ctrl	E0 1D	9D
P	19	99	M	32	B2	left	4B	CB			

Clasificación de Interrupciones

- ♣ POR SOFTWARE (internas)
- ♣ POR HARDWARE (externas)

Interrupciones Externas: Son provocadas por dispositivos externos del microprocesador.

Interrupciones Internas: Ocurren como resultado de la ejecución de una instrucción INT o como la división entre cero.



Coprocador matemático

El FPU (Float Point Unit) tiene 8 registros de 80 bits idénticos y 3 registros de 16 bits (la Palabra de control, La palabra de estado y la palabra de etiqueta), todos ellos accesibles al programador. También tiene un registro de banderas el cual no es accesible de manera directa.

Registros de 80-bits

Los registros de 80-bit están señalados en la mayor parte de la literatura como un stack de 8 registros. Cuando el FPU se inicia, todos los compartimientos se encuentran vacíos y el compartimiento 0 (ST(0) o simplemente ST) estará en la posición inicial.

Notar que estos números de los registros son para uso interno estricto del FPU. Son utilizados para identificar cada compartimiento como una unidad física similar a EAX, EBX, etc.,.... que son registros en el CPU.

Regla #1: Un registro de 80-bit del FPU debe estar libre para poder cargar un valor en él.

Sólo se puede realizar pop en el compartimiento tope del FPU.

Cualquier valor cargado al FPU debe inicialmente estar referido a ST(0) debido a que solamente puede ser cargado al registro tope. Si al FPU se carga un valor mientras el primero está todavía ahí, ese segundo valor sería ahora referido como ST(0) debido a que ahora se ha convertido en el nuevo tope. Como consecuencia, el primer valor ahora se referenciará como ST(1). Si otro valor es cargado, el primer valor tendría que ser referido ahora como ST(2). Después de realizar un pop, ese mismo primer valor será referido como ST(1).

Regla #2: El programador debe constantemente rastrear la ubicación relativa de los valores que existen en los registros en la medida que se realicen operaciones de carga y pop en los registros.

Cuando a un registro se le realiza un pop, el valor que contenía en ese registro ya no puede ser usado dentro de FPU. Si dicho valor se usará más tarde, deberá ser almacenado en memoria antes de realizar el pop y poder cargarlo cuando se requiera.

Registros de 16-bits

Los registros de 16-bits que están disponibles para el programador son la palabra de control (control Word), la palabra de estado (status Word), y la palabra de etiqueta (tag Word).

Palabra de Control

El registro de 16-bit Palabra de Control se usa por el programador para poder seleccionar entre varios modos de computo disponible en el FPU, y para definir cuál de las excepciones debe ser manejada por el FPU o por un manejador de excepciones escrito por el programador.

La palabra de control está dividida en varios campos como se muestra en la siguiente figura 1.2.

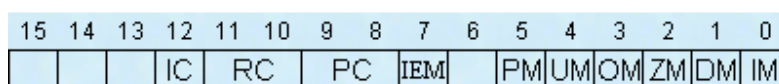


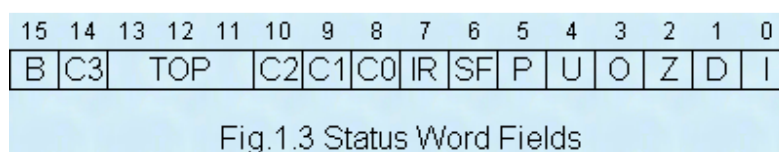
Fig.1.2 Control Word Fields

Las instrucciones para acceder el registro de palabra de control son FSTCW y FLDCW.

Palabra de estado

El registro de palabra de estado es 16-bit indica la condición general del FPU. Su contenido puede cambiar después de cada instrucción ejecutada. Parte de él no puede ser cambiada directamente por el programador. Sin embargo puede accederse de manera indirecta en cualquier momento para observar su contenido.

El registro de estado de palabra está dividido en varios campos como se muestra en la siguiente Figura 1.3.

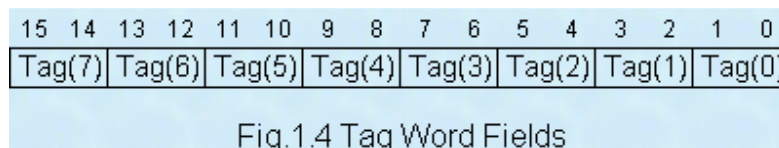


Ver la instrucción FSTSW para detalles acerca de acceder el registro de palabra de estado.

Tag Word

El registro "Tag Word" de 16-bit es manejado por el FPU para mantener alguna información en el contenido de cada uno de los registros de 80-bit.

La palabra etiqueta está dividida en 8 campos de 2 bits cada uno como se muestra en la figura 1.4.



Los números Tag corresponden a la numeración interna para los registros de 80 bits. El significado de cada uno de los pares de bits es como sigue:

- 00 = El registro contiene un valor diferente de cero válido.
- 01 = El registro contiene un valor igual a 0
- 10 = El registro contiene un valor especial (NAN, infinito, o no normal)
- 11 = El registro está vacío.

Cuando el FPU está inicializado, los registros de 80-bit están vacíos y la palabra etiqueta por tanto tiene un valor de 1111111111111111b (FFFFh).

Registro interno de banderas

EL FPU cuenta con un registro de banderas de excepción el cual no es accesible al programador. Todas estas banderas se limpian antes de cada instrucción y son puestas en 1 conforme se encuentren excepciones. Estas son las banderas que disparan una respuesta desde el FPU o una interrupción para el manejador de excepción del programador. Es posible que muchas de las banderas puedan ser modificadas con una instrucción. Por

ejemplo usando un número no normal como un operando debiera prender la bandera Denormal Flag. El resultado de la operación con él podrá entonces poner en 1 las banderas Underflow y Precision.

Para poder ver los registros asociados al coprocesador numérico en el TD hay que ir al menú View → Numeric Processor, esto desplegará los registros.

ST(0)-ST(7)

Todas las instrucciones relacionadas con el coprocesador matemático inician con F.

FLD	Loads A Float
FMUL	Multiply 2 Floats
FLDPI	Loads PI
FSTP	Stores a Float and Pops the Stack

NOTA: los ángulos son en radianes

Ejemplo

page 60,132

TITLE Calculando el área de un círculo

;

.model small

.stack 64

;

.data

otro db 'aqui'

data9 dd 2.45E+5

otro1 dt 12365.589 ; declara flotante en 10 bytes

area dq 10

Radio dd 2.0

;

.code

areas proc far

mov ax,@data

mov ds,ax

mov es,ax

fld radio; carga el radio como flotante radio a st(0)

fmul st,st(0); radio al cuadrado, también se puede cargar nuevamente y hacer fmul

fldpi ; pi a st

fmul ; multiplicar st=st*st(1)

fstp area ; guardar el resultado en la var area

fwait ; esperar coprocesador a que termine la tarea

mov ah,2ch

int 21h

mov ax,4c00h

int 21h

areas endp

end areas

Ejemplo

```
.386
.model small
.stack 800h
.data
Numero1 dd 25 ; Numero en formato entero
Numero2 dd 1.25 ; Numero en formato real
; (¡Ojo! este número sólo puede ser
; accedido correctamente por el copro,
; a no ser que nos hagamos una rutina
; conversora de formato de FPU a entero
; cosa que no merece la pena, porque
; es más fácil cargarlo y convertirlo
; con el copro, y más rápido)

Resul dd ? ; Variable para el resultado
Desca dd ? ; Variable para corrección de pila
```

.CODE

Start:

```
fild Dword Ptr ds:[Numero1] ;Cargar en el copro la
;variable Numero1 indicando
;que es un entero
fld Dword Ptr ds:[Numero2] ;Idem pero es un real
fadd St(0),St(1) ;Sumarlos
;St(0)=St(0)+St(1)
fstp Dword Ptr ds:[Resul] ;Descargar el resultado.
fstp Dword Ptr ds:[Desca] ;Descartar el otro valor.

mov eax,4c00h
int 21h
End Start
```

Set de instrucciones corto:

FBLD	Loads BCD Number
FBSTP	Stores And Pops a BCD Number
FILD	Loads An Integer
FIST	Stores an Integer
FISTP	Stores an Integer and Pops The Stack
FLD	Loads A Float
FSTP	Stores a Float and Pops the Stack
FST	Stores A Float
FXCH	Exchanges two stack elements
FABS	Computes Absolute Value

FADD	Adds 2 Floats
FIADD	Adds 2 Integers
FADDP	Adds real Numbers and Pops the stack
FCHS	Change Sign of Number
FDIV	Divides 2 Floats
FIDIV	Divides 2 Integers
FDIVP	Divides 2 Floats and Pops the stack
FDIVR	Divides 2 Floats but Reverses the dividend and divisor
FIDIVR	Same as above with Integers
FDIVRP	Divide real numbers, reverse order and pop stack
FMUL	Multiply 2 Floats
FIMUL	Multiply 2 Integers
FMULP	Multiply 2 Floats and Pop Stack
FPREM	Computes Partial Remainder
FPREM1	Computes Partial Remainder using IEEE Format
FRNDINT	Rounds the Operand to an Integer
FSCALE	Scales by a power of 2
FSUB	Subtracts real numbers
FISUB	Subtracts integers
FSUBP	Subtracts real numbers & pops stack
FSUBR	Subtracts real number in reverse order
FISUBR	Subtracts integers in reverse order
FSUBRP	Subtracts real numbers in reverse order and pops the stack
FSQRT	Computes the square root
EXTRACT	Extracts the exponents and significant from real number
F2XM1	Computes the value $2^x - 1$
FCOS	Computes the Cosine
FPATAN	Computes partial arctangent
FPTAN	Computes partial tangent
FSIN	Computes Sine
FSINCOS	Computes Sine and Cosine
FYL2X	Computes the expression $y * \log_2(x)$
FYL2XP1	Computes the expression $y * \log_2(x+1)$

Constants loading into stack

FLD1	Loads a 1.0
FLDL2E	Loads $\log_2(e)$
FLDL2T	Loads $\log_2(10)$
FLDLG2	Loads $\log_{10}(2)$

FLDPI	Loads PI
FLDZ	Loads 0
FCOM	Compares real numbers
FCOMP	Compares real numbers & pops the stack
FCOMPP	Compares real numbers & pops the stack twice
FICOM	Compares integers
FICOMP	Compares integers and pops the stack
FTST	Compares top of stack to 0
FUCOM	Performs an Unordered compare
FUCOMP	Performs an Unordered compare and pops the stack
FUCOMPP	Performs an unordered compare and pops the stack 2x
FXAM	Set condition bits for value at top of stack
FCLEX	Clears all unmasked floating point exceptions
FNCLEX	Clears all exceptions
FDECSTP	Decrements the stack pointer
FFREE	Clears a stack element, making it seem as if it was popped
FINCSTP	Increments the stack pointer
FINIT	Initializes 387 and checks for exceptions
FNINIT	Initializes 387 w/o checking for exceptions
FLDCW	Loads the Control word
FLDENV	Loads the 387 Environment
FNOP	Equivalent to NOP
FRSTOR	Restores the state of the 387 with a given memory area
FSAVE	Saves the state of the 387 to memory and checks for exceptions
FNSAVE	Saves the state of the 387 to memory w/o checking for exceptions
FSTCW	Stores the control word and checks for exceptions
FNSTCW	Stores the Control word w/o checking for exceptions
FSTENV	Stores the environment and checks for exceptions
FNSTENV	Stores the environment w/o checking for exceptions
FSTSW	Stores the status word and checks for exceptions
FNSTSW	Stores the status word w/o checking for exceptions
FSTSW AX	Stores the status word into AX and checks for exceptions
FNSTSW AX	Stores the status word into AX w/o checking for exceptions
WAIT	Suspends the CPU until the 387 is finished with operation.

Saltos, ciclos y procedimientos

Los saltos incondicionales en un programa escrito en lenguaje ensamblador están dados por la instrucción jmp. Un **salto** es alterar el flujo de la ejecución de un programa enviando el control o la dirección indicada.

Un **ciclo**, conocido también como iteración, es la repetición de un proceso un cierto número de veces hasta que alguna condición se cumpla. En estos ciclos se utilizan los saltos “condicionales” basados en el estado de las banderas.

Por ejemplo la instrucción **jnz** que salta solamente si el resultado de una operación es diferente de cero y la instrucción **jz** que salta si el resultado de la operación es cero.

Por último tenemos los **procedimientos o rutinas**, que son una serie de pasos que se usarán repetidamente en el programa y en lugar de escribir todo el conjunto de pasos únicamente se les llama por medio de la instrucción **CALL**.

Realmente lo que sucede con el uso de la instrucción **CALL**, es que se guarda en la pila el registro IP y se carga la dirección del procedimiento en el mismo registro, conociendo qué IP contiene la localización de la siguiente instrucción que ejecutará la UCP, entonces podemos observar que se desvía el flujo del programa hacia la dirección especificada en este registro. Al momento en que se llega a la palabra **RET** se saca de la pila el valor del IP con lo que se devuelve el control al punto del programa donde se invocó al procedimiento.

Llamada a procedimientos

Es posible llamar a un procedimiento que se encuentre ubicado en otro segmento, para esto el contenido de CS (que nos indica qué segmento se está utilizando) es empujado en la pila.

JMP (Salto incondicional)

Salta a una dirección designada bajo cualquier condición una dirección de **jmp** puede ser corta (-128 a 127 bytes), cercana (dentro de 32 K) o lejana (a otro segmento).

Un **jmp** corto reemplaza el IP con el desplazamiento de la dirección de destino. Un salto lejano (como **jmp far PTR etiqueta**) reemplaza el CS:IP con una nueva dirección de segmento.

Código fuente:

JMP {reg/mem}

Reglas sobre distancias para **jmp**, **loop** y **call**

Instrucción	Corta (mismo segmento) -128 a 127	Cercana (mismo segmento) -32768 a 32767	Lejana (a otro segmento)
JMP	Sí	Sí	Sí
LOOP	Sí	No	No
CALL	N/A	Sí	Sí

Nota: **JMP** vacía el resultado de la instrucción previamente procesada, por lo que un programa con muchas operaciones de salto puede perder velocidad de procesamiento.

Existen dos variaciones de la instrucción LOOP, ambas también decrementan el CX en 1 LOOPE/LOOPZ (repite el ciclo mientras sea igual o repite el ciclo mientras sea cero). Continúa el ciclo mientras el valor de CX no es cero o la condición de cero no está establecida.

Código Fuente:

LOOPE/LOOPZ etiqueta

Ejemplo

Algoritmo:

CX = CX – 1

If (CX != 0) and (ZF = 1) then

Jump

Else

No jump, continue

; el loop se realiza hasta que el resultado solo cabe en AL solamente

; o 5 veces. El resultado será mayor a 255

; En el tercer loop (100+100+100),

; Entonces el loop saldrá.

MOV AX, 0

MOV CX, 5

label1:

ADD AX, 100

CMP AH, 0

LOOPE label1

Loops

Instrucción	Operación y condición de salto	Instrucción opuesta
LOOP	Decrementa cx, salta a la etiqueta si cx no es cero.	DEC CX and JCXZ
LOOPE	Decrementa cx, salta a la etiqueta si cx no es cero e igual (zf = 1).	LOOPNE
LOOPNE	Decrementa cx, salta a la etiqueta si cx no es cero y no igual (zf = 0).	LOOPE
LOOPNZ	Decrementa cx, salta a etiqueta si cx no es cero y zf=0.	LOOPZ
LOOPZ	Decrementa cx, salta a etiqueta si cx no es cero y zf=1.	LOOPNZ
JCXZ	Salta a etiqueta si cx es cero.	OR CX, CX y JNZ

LA INSTRUCCIÓN CMP

Utilizada para comparar dos campos de datos, uno o ambos de los cuales están contenidos en un registro.

Formato General:

[etiqueta] CMP {reg/mem}, {reg/mem}

El resultado de una operación CMP afecta las banderas:

AF, CF, OF, PF/SF Y ZF

El código siguiente prueba el registro BX por un valor cero.

CMP BX, 00 ; compara BX con cero

JZ B50 ; salta si es cero o B50

Nota: Siempre comparamos y después saltamos. En el turbo debugger vemos la etiqueta si no su dirección.

Para el DEBUG

100 mov bx, 00

103 CMP bx, 00

JMP 200

*IP = 200

Instrucciones de salto incondicional

Formato General:

Jnnn dirección corta (-127 a 128)

Ejemplo que hace lo mismo que un LOOP

A 20 MOV ...

...


DEC CX

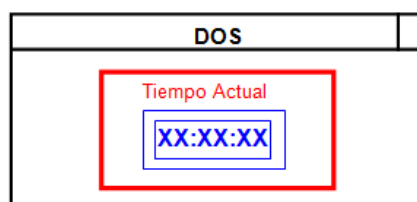
CMP CX, 00

JNZ A20

Salto con base en datos sin signo

Símbolo	Descripción	Banderas
JE/JZ	Salta si es igual o si es cero	ZF
JNE/JNZ	Salta si no es igual o si no es cero	ZF
JA/JNBE	Bifurca si es mayor o salta si no es menor o igual	CF,ZF
JA/JNB	Salta si es mayor o igual o si no es menor	CF
JB/JNAE	Salta si es menor o si no es mayor o igual	CF
JBE/JNA	Salta si no es menor o igual o salta si no es mayor	CF/AF

 Hacer un programa en el Turbo Debugger que realice un marco en pantalla y dentro de éste despliegue un reloj empleando interrupciones.



Desplegar código ASCII en pantalla dentro de un marco y colocarlo debajo del reloj anteriormente hecho.

Salto con base en datos con signo

Símbolo	Descripción	Banderas
JE/JZ	Salta si es igual o si es cero	ZF
JNE/JNZ	Salta si no es igual o salta si no es cero	ZF
JG/JNLE	Salta si es mayor o salta si no es menor o igual	ZF,SF,OF
JNL/JGE	Salta si es mayor o igual o salta si no es menor	SF,OF
JL/JNGE	Salta si es menor o salta si no es mayor o igual	SF,OF
JLE/JNG	Salta si es menor o igual o salta si no es mayor	ZF,SF,OF

Pruebas aritméticas especiales

Símbolo	Descripción	Banderas
JS	Salta si el signo es negativo	SF
JNS	Salta si el signo es positivo	SF
JC	Salta si hay acarreo (=JB)	CF
JNC	Salta si no hay acarreo	CF
JO	Salta si hay desbordamiento	OF
JNO	Salta si no hay desbordamiento	OF
JP/JPE	Salta si hay paridad o salta si la paridad es par	
JNP/JPD	Salta si no hay paridad o salta si la paridad es impar	

Llamadas a procedimientos

Beneficios:

- ♣ Reduce código.
- ♣ Fortalece la mejor organización del programa.
- ♣ Facilita la depuración.
- ♣ Ayuda en el mantenimiento progresivo de programas (rápida modificación).
- ♣ Su uso implica más tiempo, que en algunos casos es crítico, puede tardarse hasta 1% más de tiempo.

OPERACIONES CALL Y RET

La instrucción CALL transfiere el control a un procedimiento llamado, y la instrucción RET regresa del procedimiento llamado al procedimiento original que hizo la llamada RET, ésta debe ser la última instrucción en un procedimiento.

Formatos Generales:

[etiqueta] CALL procedimiento
[etiqueta] RET [inmediato]

CALL: Hace una llamada a un procedimiento, antes de saltar guarda el IP, termina el proceso de pipeline.

RET: Recupera el IP y salta al IP inmediato en la pila, siempre regresa al procedimiento que hace la llamada.

El código objeto particular que CALL y RET generan depende de si la operación implica un procedimiento NEAR (cercano) o un procedimiento FAR (lejano).

Una llamada (CALL) a un procedimiento dentro del mismo segmento es cercana y realiza lo siguiente:

- ♣ Disminuye en SP en 2 (una palabra).
- ♣ Introduce el IP (que contiene el desplazamiento de la instrucción que sigue al CALL) en la pila.
- ♣ Inserta la dirección del desplazamiento del procedimiento llamado en el IP (esta operación vacía el resultado de la instrucción previamente procesada). Un RET que regresa desde un procedimiento cercano realiza lo siguiente.
- ♣ Saca el antiguo valor de IP de la pila y lo envía al IP actual (la cual también vacía el resultado de la instrucción previamente procesada).
- ♣ Incrementa el SP en 2.

Ahora el CS:IP apunta a la instrucción que sigue al CALL original en la llamada del procedimiento, en donde se reanuda la ejecución.

MANEJO DE LA PILA

PUSH: SP se decrementa en 2

POP: SP se incrementa en 2

Sintaxis para la instrucción **PUSH:**

PUSH REG

PUSH SREG

PUSH memory

PUSH immediate

REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memoria: [BX], [BX+SI+7], 16 bit variable, etc...

inmediato: 5, -24, 3Fh, 10001101b, etc...

Sintaxis para la instrucción **POP:**

POP REG

POP SREG

POP memory

REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, (excepto CS).

memoria: [BX], [BX+SI+7], 16 bit variable, etc...

Nota: las instrucciones PUSH y POP trabajan sólo con 16 bits.

VARIABLES

- ♣ **Locales:** Estas variables se encuentran en el stack, ya no se puede hacer referencia a ellas, se extinguen cuando termina el procedimiento.
- ♣ **Globales:** Están contenidas en memoria principal razón por la cual no se extinguen al terminar el procedimiento.

Calcular las siguientes expresiones con los procedimientos que se solicitan (para cuando Z=2)

$$X = Z^3$$

$$Y = 2x^2 + 2X + 5!$$

Procedimientos:

Nombre del Procedimiento	Operación que realiza	Parámetros
POT	potencia	- base - potencia - resultado
SUMA	suma	- sumandos - resultado
MULTI	multiplicación	- multiplicando - resultado
FACT	factorial	- factorial - resultado

Page 60, 132

Title Ejemplo CALL y RET

```
.model small
.stack 64
.data
X dw 00
Y dw 00
Z dw 02
TMP dw 00
.code
```

```
Principal proc far
mov ax, @data
mov ds, ax
mov es, ax
mov dx, 03
push z
push dx
push tmp
call pot
```

```

    pop x

Principal endp
    POT proc far
        mov sp, bp
    mov ax, 01
        mov bx, [sp] ; bx=z
        mov cx, [sp+2] ;[dx]=potencia
    et1 mul bl
        loop et1
        add sp, 4
        mov [sp], ax
        ret
    POT endp

```

MACROS

Una macro es un grupo de instrucciones que ejecutan una tarea, así como un procedimiento. La diferencia es que un procedimiento es accedido por medio de una instrucción CALL, mientras una macro y todas las instrucciones definidas en el macro son insertadas en el programa en el punto de uso.

Las instrucciones de una macro son ubicadas en el programa por el ensamblador en el punto en que son invocadas.

Son útiles para:

- ♣ Simplificar y reducir la cantidad de codificación repetida. (Simplifica el procedimiento más no el código).
- ♣ Reducir errores causados por la codificación repetitiva.
- ♣ Linealizar un programa en lenguaje ensamblador para hacerlo más legible.

Diferencias entre un procedimiento y una macro

PROCEDIMIENTOS	MACROS
<ul style="list-style-type: none"> - Nacen para reducir espacio y la repetitividad. - Facilitan la corrección. - Simplifica el código. - Ocupa menos espacio. - Rompe el pipeline. - Este ralentiza el proceso hasta 1% en tiempo en comparación de un macro. 	<ul style="list-style-type: none"> - Se resuelve en tiempo de ensamblado. - Simplifica la repetición de código. - Ocupa más espacio que los procedimientos. - No rompe el pipeline. - Este es más rápido (1% con respecto de una llamada a procedimiento). - Se inserta antes de cualquier definición de segmento.

Una definición de macro aparece antes que cualquier definición de segmento, por lógica de ensamblado, como lo emplea en el código necesita estar definida antes.

Ejemplo:

Macro que inicializa los registros de segmento para un .EXE

INITR macro ; Define una macro

Miguel Israel Barragán Ocampo

```
mov ax, @data  
mov ds, ax  
mov es, ax  
endm ; fin de macro
```

Programa que despliega el mensaje 'PRUEBA DE MACRO'

Page 60, 132

Title P22 MACR1 (EXE)

```
INITZ macro ; Define una macro
mov ax, @data
mov ds, ax
mov es, ax
endm ; fin de macro
.model small
.stack 64
.data
message DB 'Prueba de Macro', '$', 09, 0A
.code
BEGIN PROC FAR
INITZ
Mov ah, 09h
Lea dx, message
Int 21h
Mov ax, 4000h
Int 21h
endp
end BEGIN
```

Nota: Opción 9 Int 21h muestra caracteres en la pantalla.

'\$' Fin de cadena

dx→ dirección de inicio de la cadena.

 Ver qué es lo que realiza el programa anterior y sacar conclusiones.

USO DE PARÁMETROS EN MACROS

Para hacer una macro flexible, puede definir nombres en ella como argumentos mudos (ficticias).

Ejemplo:

```
PROMT      MACRO    message ; parámetro que le vamos a pasar
mov ah, 09h
lea dx, messge
int 21h
endm
```

TEMPORAL

Mover macro A, B ; parámetros. Existen pero no aparecen se sustituyen por las variables que estamos llamando por eso son mudas.

```
    push ax
    mov ax, b
    mov a, ax
    pop ax
endm
mover var1, var2
    push ax
    mov ax, var2
    mov var1, ax
    pop ax
endm
```

VARIABLES LOCALES

Una variable local es aquella que aparece en el macro pero que no está disponible fuera de él, para definirla usamos la directiva LOCAL.

Ejemplo:

```
READ      MACRO  A ; leer teclado
           local  Read 1
           push dx
Read1 :
           mov ah, 6
           mov dl, 0ffh
           int 21h
           JE Read 1
```

Nota: Opción 06 Int 21_h, recibe datos desde teclado.

```
FACTORIAL    MACRO  num
              local  et1
              xor ax, ax
              mov al, 01
              mov cl, num
et1:         mul cl
              loop et1 *
FACTORIAL endm
```

Nota: Variable Local: ya que como necesita insertarlo de nuevo la dirección a donde salta varia y si no se define como variable local tendría un error de referencia.

Definición de una estructura en ensamblador

Una estructura es un tipo de variable que contiene otras variables, llamadas campos.

La palabra reservada STRUC inicia la estructura, seguida en la misma línea por cualquier nombre que se quiera. La palabra reservada ends termina la estructura, también se puede poner el nombre de la estructura después del ends; por ejemplo la siguiente estructura contiene tres campos que representan una fecha:

Miguel Israel Barragán Ocampo

```
Struc Date
dia    db 1
mes    db ?
año    dw 2006
ends Date
```

Se pueden insertar campos de cualquier tipo dentro de una estructura.

Cuando se defina una estructura como esta, recuérdese los siguientes puntos:

- Una estructura no es una variable. Una estructura es el esquema para una variable.
- Las estructuras se pueden declarar en cualquier lugar. La directiva struc no se localiza en el segmento de datos del programa. Aunque ciertamente podría estar ahí.
- Una estructura le indica al Turbo ensamblador acerca del diseño de las variables, que se planean declarar después o que ya existen en la memoria.
- Aun cuando se utilizan directivas tales como db y dw para definir los tipos de los campos de la estructura, la estructura no reserva espacio en el segmento de datos.

DECLARANDO VARIABLES ESTRUCTURADAS

Para usar el diseño de una estructura, se debe reservar espacio en memoria para los campos de la estructura. El resultado es una variable que tiene el diseño de la estructura. Tal declaración comienza con una etiqueta, seguida del nombre de la estructura, y terminando con una lista de valores por default entre <>. Deje los < y > vacíos si se quiere que tome los valores por default dados en la definición de la estructura: ejemplo

```
DATASEG
Birthday    Date <> ;1-0-2006
Today       Date <5,10> ; 5-10-2006
Dayinayout  date <11,12,1912> ; 11-12-1912
```

Para hacer referencia al campo de la estructura:

```
Mov [today.day],5 ; cambia el día a 5
Mov ax,[today.year] ; obtiene el año y lo mueve a ax
Inc [earthday.day]
Cmp [today.month],8
```

Estructura de círculo

```
circulo struc
posX    dd ? ; Posición X izquierda del círculo * 65536.
posY    dd ? ; Posición Y superior del círculo * 65536.
incX    dd ? ; Incremento X en 1/18,2 segundo (* 65536)
          ; Este número tiene signo.
incY    dd ? ; Incremento Y en 1/18,2 segundo (* 65536).
          ; Este número tiene signo.
radio   dd ? ; Radio del círculo.
```

Miguel Israel Barragán Ocampo

```
color db ? ; Color del círculo.
circulo ends
```

Estructura paleta (de colores)

```
paleta struc ; Definición de paleta de 18 bits de VGA.
rojo db ? ; Vale entre 00h y 3Fh.
verde db ? ; Vale entre 00h y 3Fh.
azul db ? ; Vale entre 00h y 3Fh.
paleta ends
```

ejemplo:

```
title Demostración de Estructuras
        IDEAL
        MODEL small
        STACK 256

struc Fecha
dia db 1 ; día valor de default 1
mes db ? ; mes sin valor por default
year dw 2006 ; año con valor por default 2006
ENDS Fecha

STRUC CiudadEstado
ciudad db '#####', 0 ; 20 caracteres
estado db '###', 0 ; dos caracteres
ENDS CiudadEstado

DATASEG
exCode db 0
        hoy Fecha <>
        cumple Fecha <8,8,1954>
        diaDeLaTierra Fecha <1,1,2006>
        añoNuevo Fecha <,,2007>
direccion CiudadEstado <>
capital CiudadEstado <'Mexico','D.F.'>
estado1 CiudadEstado <'Orizaba','Ver'>
estado2 CiudadEstado <'Taxco','Gue'>
EstadoDefault CiudadEstado <,'Chi'>
CiudadDefault CiudadEstado <'Campeche',>
CODESEG

Start:
        mov ax,@data ;DS=Data
        mov ds,ax

Exit:   mov ah, 04ch
        mov al, [exCode]
        int 21hEND Start ; fin de programa / punto de entrada
```

Programas .com

Title Estructura para archivos .COM

MODEL tiny

;----- insertar INCLUDE "nombre del archivo" y directivas

;----- insertar EQU y = aquí

DATASEG

exCode DB 0

;----- Declarar otras variable con DB, DW, etc,... aquí

CODESEG

ORG 100h ; Dirección estándar de inicio para programas .COM (origen)

Star:

;----- Insertar programa, llamadas a subrutinas etc, aquí

Exit:

Mov ah,4ch ; función del DOS: Salir de programa

Mov al, [exCode] ; regresa el valor del código de error

Int 21h ; llama al DOS , termino de programa

END Star; fin de programa / punto de entrada → después de la directiva END ya no toma en
; cuenta el ensamblador lo que sigue.

Programas .EXE

Title Estructura para archivos .EXE

MODEL small

Stack 256

;----- insertar INCLUDE "nombre del archivo" y directivas

;----- insertar EQU y = aquí

DATASEG

exCode DB 0

;----- Declarar otras variable con DB, DW, etc,... aquí

; ---- Especificar cualquier variable EXTRN aquí

CODESEG

; especificar cualquier procedimiento externo aquí

Miguel Israel Barragán Ocampo

Stara:

```
Mov ax, @data ;Inicializa DS en la dirección del
```

```
Mov ds,ax ;segmento de datos
```

```
Mov es,ax ; hace es=ds
```

;---- Insertar Programa, llamadas a subrutinas, etc,... aquí

Exit:

```
Mov ah,4ch ; función del DOS: Salir de programa
```

```
Mov al, [exCode] ; regresa el valor del código de error
```

```
Int 21h ; llama al DOS , termino de programa
```

END Stara; fin de programa / punto de entrada → después de la directiva END ya no toma en
;cuenta el ensamblador lo que sigue.

Para ligar y obtener un .com es necesario poner le modificador /t

Tlink /t nombre_archivo

En el final de programa, el programa no se detiene cede el control a otro programa, lo que hace la opción 4ch de la interrupción 21h es recargar el COMMAND.COM

Los programas .COM ocupan el segmento completo a la hora de ejecución a diferencia de los .EXE, usualmente un programa .EXE equivalente (en función) a un .COM ocupa menos espacio.

Generar Librerías

Para generar librerías en asm se tiene el siguiente esquema:

Para el programa principal que hace las llamadas a los procedimientos contenidos en la librería.

page 60,132

title Haciendo ALGO

;declaración de macros

cuadro macro A,B,C,D ;renglon, columna, ancho,largo

```
mov ax,A
```

```
push ax
```

```
mov ax,B
```

```
push ax
```

```
...
```

```
call dib_cuadros ;llamada a procedimiento de una librería
```

endm

movcursor macro A,B,C

```
mov ah,02
```

```
...
```

```
int 10h
```

endm

;declaración de modelo de memoria

.model tiny

```
;declaración de segmento de datos
.data
;declaración de variables
    ten1 equ 10
    ten2 = 10
    texto db 'AQUI ESTOY'; este es sólo para que sepas en dónde andas
    dots   db 58
    ...
;declaración de segmento de código
.code
; agregando las referencias de las librerías
; ---- de strings.obj y strio.obj
    extrn StrLength:proc, StrRead:proc ;tb pueden ser directivas nombre y etiquetas
    extrn StrWrite2:proc, NewLine:proc ; proc es el tipo de elemento externo a usar
; ---- de grafico.obj
inicio:
;main proc
    mov ax,@data
    mov ds,ax
; limpiamos pantalla usando procedimiento en librería
    call clrscr
;llamada a macro definida al inicio que hace uso de proc en librería
    cuadro 2,4,71,4
    ...
@repeat:
    in al,60h          ; leyendo puerto 60h
    dec al             ; if puerto 60h <> 1 (esc) sigue saltando
    jnz @repeat        ; =)))
;regresando a sistema operativo
    mov ah,4ch
    int 21h
    end inicio
```

Para los procedimientos a ser llamados y contenidos en la librería

```
; -----
; DIBUJA RECTANGULOS EN MODO TEXTO int x, int y, ancho, largo
; -----
;se declara el modelo de memoria
.model small
;se declara el segmento de código
.code
;se declara como público el proc dib_cuadros
;para poder ser llamado desde una librería o un archivo .obj
public dib_cuadros
public MoveLeft, MoveRight
```

```
;declaración del procedimiento
dib_cuadros proc
mov bp,sp
...
pop bx
pop bx
pop bx
push ax
;declaración del regreso de llamada a proc
ret
; declaración del segmento de datos
.data
;declaración de las variables y constantes
X db 0; coordenada de inicio en X
Y db 0; coordenada de inicio en Y
ancho db 0; ancho de cuadro
largo db 0;largo de cuadro
...
;fin del procedimiento
;poner el nombre del proc después de endp no al revés
endp dib_cuadros

;-----
; MoveLeft Mueve bloques de byte a la izquierda (abajo) en la memoria
;-----
; Entrada:
; si= dirección de la cadena fuente (s1)
; di= dirección de la cadena destino (s2)
; bx= índice s1 (i1)
; dx= índice s2 (i2)
; cx = número de bytes a mover (count)
; Salida:
; bytes contados desde s1[i1] movidos a la ubicación destino
; comenzando desde s2[i2]
; Registros:
; ninguno
;-----
```

```
PROC MoveLeft
    jcxz @@99 ; salir si count = 0
    push cx ; salvar registros modificados
    push si
    push di
    add si, bx ; índice a fuente de cadena
    add di, dx ; índice en cadena de destino
    cld ; autoincremento de si y di
    rep movsb ; mover mientras cx != 0
    pop di ; restablece los registros
```

```

        pop si
        pop cx
@@99:
        ret
ENDP  MoveLeft
; -----
; MoveRight Mueve bloque de bytes a la derecha (hacia arriba ) en memoria
; -----
; Entrada:
; (el mismo que MoveLeft)
; Salida:
; (el mismo que MoveLeft)
; Registros:
; ninguno
; -----
PROC MoveRight
    jcxz @@99
    push cx
    push di
    push si
    add si,bx
    add di,dx
    add si,cx ;Ajusta al último byte fuente
    dec si
    add di,cx; ajusta al último byte de destino
    dec di
    std     ;autodecrementa a si y di
    rep movsb ; mueve mientras cx !=0
    pop si
    pop di
    pop cx
@@99:
    ret
ENDP  MoveRight
end

```

para ensamblar:

tasm /zi principal.asm

tasm /zi proced.asm

esto genera principal.obj y proced.obj

para generar la librería:

una vez que se ha ensamblado el archivo, en donde están nuestros procedimientos a llamar, en este caso proced.obj se procede al ejecutar el siguiente comando

tlb /E oceotl -+ proced

Miguel Israel Barragán Ocampo

Esto genera librería oceotl.lib que contiene ya los procedimientos de proced; si el modulo no existe mostrará un warning de que proced no está en la librería, esto es normal. Para ver el contenido de la librería se ejecuta el siguiente comando:

tlib oceotl, uno

Esto genera un archivo de salida uno.lst; en éste se encuentra los procedimientos de la librería y a qué modulo pertenecen.

Ejemplo:

```
Publics by module
grafico size = 60
  CLRSCR
```

```
grafico2      size = 373
  DIB_CUADROS
```

Aquí hay dos módulos (archivos objeto con procedimientos) el grafico y el grafico2, en cada uno de ellos hay un procedimiento CLRSCR y DIB_CUADROS, estos son los procedimientos que se pueden mandar llamar desde la librería. Este .lst no indica cómo usar los procedimientos y como no tienen firma, es responsabilidad de quién lo programe el que documente cómo es que se debe usar el procedimiento.

Para hacer uso de las librerías (ligar-linker)

Ya una vez que se tiene la librería generada es decir que exista el archivo nombre.lib en nuestro caso oceotl.lib, se utiliza el siguiente comando.

Tlink <objetos> ,,, oceotl

Esto generará un ejecutable que hace uso de los obj's y de los procedimientos que hay en la librería oceotl y que son usados en los obj's.

Si se requiere usar más de una librería:

Tlink <objetos> ,,, oceotl mta mta2 , etc

Para ensamblar y ligar módulos separados

Se ensamblan los módulos

Ejemplo:

```
Tasm uno.asm
Tasm dos.asm
Tasm tres.asm
```

Esto generará los archivos uno.obj, dos.obj, tres.obj

Para generar el ejecutable será

Tlink uno dos tres → uno.exe que se considerará como el principal

Para generar un ejecutable con nombre diferente del principal

Tlink uno dos tres, números → números.exe

El formato completo para el ligado es el siguiente:

Tlink <opciones> <objetos> archivo.exe (nombre resultante), mapfile(nombre),libfiles, deffile, restfiles

Los dos últimos deffile, restfiles se utilizan cuando se programa para el ambiente Windows.

Las etiquetas y nombres declaradas como public pueden ser usadas en cualquier programa "host".

Aunque los enteros proveen una representación exacta de los valores numéricos, sufren de dos grandes desventajas: la inhabilidad de representar valores fraccionales y un rango dinámico limitado. La aritmética de punto flotante resuelve estos dos problemas al costo de perder exactitud y en algunos procesadores velocidad. Para muchas aplicaciones, los beneficios del uso del punto flotante compensan sus desventajas. Sin embargo, para usar apropiadamente la aritmética de punto flotante en cualquier programa, se debe aprender cómo opera la aritmética de punto flotante.

Formatos de Punto Flotante de la IEEE

Cuando Intel planeó introducir el coprocesador de punto flotante para su nuevo procesador 8086, fueron lo suficientemente inteligentes como para comprender que los ingenieros eléctricos y los físicos de estado sólido quienes diseñan los chips, eran, tal vez, no la mejor gente para realizar el análisis numérico para escoger la mejor representación posible binaria para el formato del punto flotante. Por ello Intel contrató los mejores analistas numéricos que pudieron encontrar para diseñar un formato de punto flotante para su FPU 8087. Esta persona después contrató a otros dos expertos en el campo y los tres (Kahnem, Coonan y Stone) diseñaron el formato para punto flotante de Intel. Ellos realizaron tan buen trabajo diseñando el estándar KCS de punto flotante que la IEEE lo adoptó como su formato estándar.

Para manejar un amplio rango de desempeño y de exactitud, Intel introdujo tres formatos de punto flotante posibles: precisión simple, doble precisión y la precisión extendida. La precisión simple y la doble corresponden a la precisión float y double en C. o real y doble de FORTRAN. La precisión extendida contiene 16 bits extra que los cálculos pueden usar antes del redondeo hacia abajo a un valor de precisión doble cuando se guarda el resultado. Bajo este esquema, un número puede ser expresado mediante un exponente y una mantisa. Por ejemplo el número 10.75 puede ser expresado como:

10.75 x 10 0

1.075 x 10 1

mantisa exponente

En general, un número en punto flotante puede ser representado como $\pm d_0.d_1d_2d_3\dots d_k \times b_{\text{exp}}$ donde $d_0.d_1d_2d_3\dots d_k$ se conoce como la mantisa, b es la base y exp es el exponente. ¿Qué se necesita para representar un número en punto flotante?

- ♣ el signo del número.
- ♣ el signo del exponente.
- ♣ Dígitos para el exponente.
- ♣ Dígitos para la mantisa.
- ♣

1.00 x 10⁻¹ normalizado
0.01 x 10² no normalizado

La representación de un número en precisión doble en el formato IEEE-754 consta de las siguientes partes:

- ### Ejemplo

Necesitamos obtener el signo, el exponente y la fracción.

Primero, escribimos el número (sin signo) usando notación binaria. El resultado es 1110110.101

Ahora, movamos el punto decimal a la izquierda, dejando sólo un 1 a su izquierda.

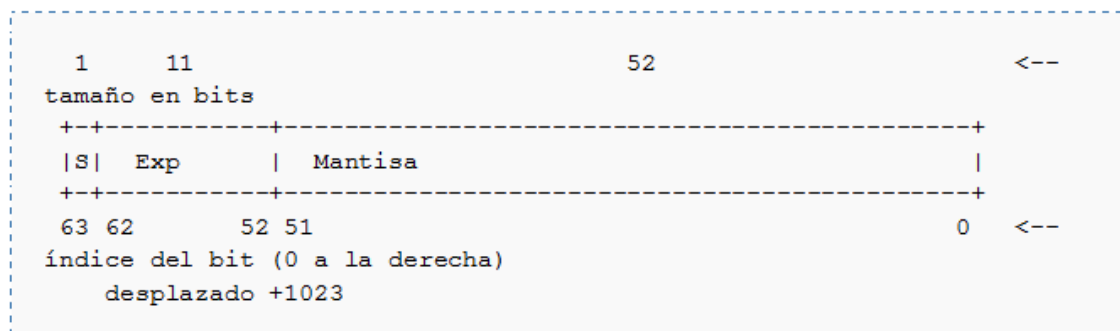
La mantisa es la parte a la derecha del punto decimal, rellena con ceros a la derecha hasta que obtengamos todos los 23 bits. Es decir 11011010100000000000000.

Poniendo todo junto:



Precisión doble 64-bits

La precisión doble es esencialmente lo mismo exceptuando que los campos son de mayor tamaño (más bits por campo):



Los NaN s y los infinitos son representados con todos los bits de los Exp siendo 1 (2047 en decimal). Para los números normalizados, el exponente es desplazado +1023 (así nuestro ejemplo anterior es Exp - 1023) Para números desnormalizados el exponente es -1022 (el mínimo exponente para un número normalizado— no es 1024 porque los números normalizados tienen un bit a 1 delante del punto binario y los números desnormalizados no). Como antes, ambos infinitos y los ceros contienen signo.

Comparación de números en punto flotante

La comparación de números en punto flotante se realiza generalmente usando instrucciones de punto flotante. Sin embargo esta representación (IEEE 754) hace la comparación de determinados subconjuntos posible byte-por-byte, si comparten el mismo orden de bytes y el mismo signo, y además los NaN s son excluidos.

Por ejemplo, para dos números positivos a y b, $a < b$ es cierto siempre que los enteros binarios sin signo con los mismos patrones de bits y el mismo orden de bytes que a y b son también ordenados de forma $a < b$. En otras palabras, dos números positivos (que se sabe que no son NaN s) puede ser comparados con una comparación entre enteros binarios sin signo entre los mismos grupos de bits, teniendo como base que los números tienen el mismo orden de bytes (esta ordenación, por tanto, no puede ser utilizada a través de una unión en el lenguaje de programación C. Este es un ejemplo de ordenación lexicográfica.

Redondeo de números en punto flotante

El estándar de la IEEE tiene cuatro formas diferentes de redondeo:

- ♣ **Unbiased**, que redondea al número más cercano, si el número cae en medio, este es redondeado al valor más cercano con un valor par (cero) en su bit menos significativo. Este modo es el requerido como por defecto.
- ♣ **Hacia el cero**
- ♣ **Hacia el infinito positivo**
- ♣ **Hacia el infinito negativo**

Para cambiar entre estos modos hay que cambiar la palabra de control el campo RC.

Programa que obtiene la cadena de un valor flotante.

;MIGUEL ISRAEL BARRAGAN OCAMPO

page 60,132

TITLE Obtiene la cadena de un valor flotante para imprimir en pantalla

```
;-----
.model small
.486
.stack 64
;-----
.data
otro db 'aqui'
      db 0AAh
numFlot1 dt -96385.123456789 ;declaración del número flotante en 80 bits
      db 0AAh
Entero dq 00h ; pondré la parte entera del flotante
Decimal dt 00h ; aquí guardaré el decimal que quede para después transformarlo
Digito dq 00h ; iré sacando los dígitos decimales del flotante
NumDec dw 05h ; cantidad de decimales a manejar
NumDigEnEntero dw 00h
NumInter dq 00h ; guardo el entero que se restará para obtener
; los dígitos de la parte entera del flotante
      db 0AAh
temp dw 00h
      db 0AAh
ten dt 10.0
uno dt 1.0
menosuno dt -1.0
      db 0AAh
oldcw dw ? ; para guardar el contenido de la palabra de control
newcw dw ? ;para la nueva palabra
negativo db '-'
cadflip db 100 dup (?) ; cadena donde queda el entero con flip
cadena db 100 dup (?)
.code
Float2Str proc
    mov ax,@data
    mov ds,ax
    mov es,ax
; configurando la palabra de control de manera que redondee hacia abajo
; por default redondea hacia el más cercano bits 10 y 11 en 0, 0
; para que redondee hacia abajo debe ser 1,0
    fstcw oldcw ; obtiene el estado de la palabra de control actual
    fwait; esperar a que lea la palabra
    mov ax,oldcw
    or ax,0800h ; poniendo en RC un 2 bit 10 y 11
    ;sabiendo que el procesador inicia en 0,0
```

Miguel Israel Barragán Ocampo

```

mov newcw,ax
fldcw [newcw] ; cargando la nueva palabra de control con RC = 2
; ----- comenzando la conversión, se asume que el flotante esta en st
fld numFlot1
fld st ; lo duplica
; fextract
; fscale
fstp Entero ; obteniendo el entero del flotante
fild Entero ; carga el entero para restarlo con el # flotante original
fsub ; realiza la resta, sólo queda el decimal del flotante
fstp Decimal ; lo guardo en Decimal
; ----- Proceso el entero
; verifico que no sea cero
fild Entero
fldz ;carga un cero
fcomp ; comparo
fstsw ax ; carga a ax el valor de las banderas del FPU
fwait
sahf ; carga ax al registro de banderas
je es_cero ; quiere decir que la parte entera es cero
; ----verificando el primer bit para diferenciar el signo
xor ax,ax
lea si,Entero
mov al,[si+7]
and al,80h
cmp al,80h
jne es_positivo
lea bx,cadena
mov al,negativo
mov [bx],al ; poniendo el signo de negativo
lea bx,cadflip
fld menosuno ; para pasarlo a positivo ya que sabemos que es (-)
; cambiando el signo tb de parte decimal de flotante
fld menosuno
fld Decimal
fmul
fstp Decimal
fmul
jmp sigue_obt_digitos
es_positivo:
lea bx,cadflip
sigue_obt_digitos:
lea di,NumInter
lea si,Digito
obten_dig_enteros:
fld ten
fdiv ; divide el número entre 10

```

```

fld st
mov ax,oldcw
;para que redondee hacia abajo
or ax,0400h; poniendo un 1
;or ax,0800h ; poniendo en RC un 2 bit 10 y 11
;sabiendo que el procesador inicia en 0,0
mov newcw,ax
fldcw [newcw] ; cargando la nueva palabra de control con RC = 2
fistp NumInter
fild NumInter
fsub
fld ten
fmul
;para que redondee al más cercano
; OJO!!! esto es necesario porque de otra forma obtiene mal los dígitos
;sabiendo que el procesador inicia en 0,0
mov ax,oldcw
mov newcw,ax
fldcw [newcw] ; cargando la nueva palabra de control con RC = 2
fistp Digito
xor ax,ax
mov al,[si]
add al,30h
mov [bx],al
inc bx
inc NumDigEnEntero
fild NumInter
cmp word ptr [di],00
jne obten_dig_enteros
cmp word ptr [di+2],00
jne obten_dig_enteros
cmp word ptr [di+4],00
jne obten_dig_enteros
cmp word ptr [di+6],00
jne obten_dig_enteros
lea bx,cadflip
add bx,NumDigEnEntero
dec bx
lea si, cadena
inc si
mov cx,NumDigEnEntero
flip:
mov al,[bx]
mov [si], al
dec bx
inc si
loop flip

```

```

; agregando el punto
mov byte ptr [si], '.' ; agregando el punto
jmp obt_decimales
; ===== hasta aquí la conversión de la parte entera =====
es_cero:
    lea si, cadena
    mov byte ptr [si], 30h ; el cero
    inc bx
    mov byte ptr [si], '.' ; agregando el punto
    ; continuar con los decimales

obt_decimales:
    mov ax, oldcw
    ; para que redondee hacia abajo
    or ax, 0400h ; poniendo un 1
    mov newcw, ax
    fldcw [newcw] ; cargando la nueva palabra de control con RC = 2
    ffree st ; quitando el valor del st
    lea bx, Digito
    mov cx, NumDec ; número de decimales a obtener del número
    fld Decimal ; cargando número

obteniendo:
    fld ten
    fmul
    fld st
    fistp Digito
    inc si
    mov al, [bx]
    add al, 30h
    mov [si], al
    fild Digito
    fsub
    loop obteniendo
    ffree st
    inc si
    mov byte ptr [si], '$' ; agregando el fin de cadena
    lea dx, cadena
    mov ah, 09h
    int 21h
    ; recuperando la palabra de control original
    mov ax, oldcw
    mov newcw, ax
    fldcw [newcw]
    mov ax, 4c00h
    int 21h
Float2Str endp
end Float2Str

```

PROGRAMACIÓN ORIENTADA A OBJETOS

De la manera más sencilla, un objeto es una estructura que relaciona datos y código colectivamente conocidos como los miembros del objeto. El objeto encapsula a sus miembros en un paquete muy útil.

Puntos clave de un objeto:

- ♣ El código en un objeto usualmente ejecuta algunas operaciones sobre los datos del objeto. Esto no es un requerimiento sin embargo es usualmente el caso. El código de un objeto consiste de subrutinas, llamadas métodos, que se escriben de la misma manera que las subrutinas convencionales.

Modo de video VGA

Modos alfanuméricos: En los modos alfanuméricos (modos de texto), la pantalla está dividida en una serie de celdillas donde sólo cabe un carácter (tanto un carácter alfabético como un número de ahí el término alfanumérico), proporcionándonos resoluciones generalmente reducidas, como por ejemplo 80x25, que quiere decir que tenemos la pantalla limitada a 80 caracteres por línea horizontal y con 25 líneas en el total que se visualiza. Este tipo de limitaciones es el que nos queremos saltar pasando a programar en los modos gráficos, en contraposición a los de texto. Cualquier forma gráfica en estos modos tiene que ser creada a base de combinaciones de símbolos ASCII.

Pixel: Cuando ejecutamos un programa en modo gráfico, mirando a la pantalla podemos darnos cuenta de que el dibujo ahí representado está formado por una serie de unidades mínimas (unos pequeños puntos cuadrados), cada una de ellas con un color, y con la acumulación de las cuales se forman las figuras gráficas que vemos representadas en el monitor.

La palabra pixel viene de la abreviación del término inglés PEL picture element (elemento del dibujo), y representa la unidad mínima capaz de ser representada por el monitor en esa resolución.

Resolución: Acabamos de nombrar la palabra resolución en la definición de pixel, y es que en la pantalla caben un número determinado de píxeles (o puntos) a lo ancho y a lo alto de ella. Así, podemos ver que, en un determinado juego, en la pantalla caben 320 píxeles a lo ancho y 200 a lo alto mientras en Windows pueden caben por ejemplo 640x480 píxeles en la misma pantalla.

La resolución de un modo de vídeo, especificado en el formato (ej: 640x480) nos indica la cantidad de píxeles que caben en la pantalla tanto a lo ancho (640 en este caso) como a lo alto (480), lo que quiere decir que en la pantalla hay un total de 640x480 píxeles = 307.200 píxeles (en nuestro ejemplo). Queda claro pues que a mayor resolución, como el tamaño del monitor es siempre el mismo, los puntos serán más pequeños y conseguiremos una mayor calidad gráfica, mientras que a resoluciones menores (ej: 320x200) es más fácil notar el tamaño de los puntos y la calidad representada será menor.

Coordenadas de pantalla: cuando necesitemos localizar en pantalla una posición específica (por ejemplo: un punto), lo haremos a partir de sus coordenadas, que consisten en dos números que indican una posición respecto a las coordenadas <0,0> (parte superior izquierda de la pantalla). Supongamos que en el modo 320x200 queremos localizar el centro de la pantalla. Este será, obviamente, el punto (160,100), puesto que

ambos números nos indican la posición horizontal primero y luego vertical considerando la pantalla como un eje XY. El último punto de la pantalla tendría en este caso las coordenadas (319,199).

Modos gráficos: Cada modo gráfico se caracteriza por su resolución y su número de colores. Así, tenemos desde el modo gráfico 320x200x256 (256 colores) hasta, por ejemplo, el 1024x768x16M (con 16 Millones de colores). Nuestra tarjeta gráfica es capaz de inicializar todos estos distintos modos, y podremos elegir el más conveniente a nuestros propósitos para usarlo en nuestro programa. Queda claro entonces que, cuanto más nueva y potente sea nuestra tarjeta gráfica (que va instalada dentro del ordenador), de más modos gráficos dispone, de tal manera que sería imposible inicializar el modo 800x600x256 colores en una tarjeta gráfica que no sea SVGA (Super VGA). Cuando nos dispongamos a crear un programa, tendremos que decidir primero en qué modo gráfico vamos a realizarlo, debido a que cada modo se programa de una manera y, como es obvio, a menor resolución y menor número de colores, el programa resultante será más rápido. Con respecto al término modo gráfico, que no sea un modo de texto no quiere decir que no podamos representar información textual, pues construiremos los caracteres a base de píxeles, ya que cualquier carácter es en sí mismo un símbolo gráfico.

Paleta: Llamaremos paleta (por ahora) al conjunto de colores que posee un modo gráfico (2, 16, 256, 32.000 ó 16 millones, por ejemplo).

Si pretendemos usar los distintos modos gráficos de que disponemos nos vemos en la necesidad de crear nuestro propio set de funciones (`putpixel()`, `getpixel()`, `line()`, etc...) para poder trabajar en el modo gráfico que deseemos. El problema es que cada modo de video (de todos los que vimos la anterior entrega, como 320x200, 640x480, etc...) se programa de una manera; es decir: cuando desarrollemos nuestra función `putpixel()` para poner un punto en pantalla, según para qué modo gráfico vayamos a realizar el programa o juego habrá que realizar en ella una serie de acciones u otras. Esto implica que cada modo de video de los (en principio) 19 estándar existentes se gestiona de una forma diferente, y tendremos que crear una librería especializada para cada uno de ellos.

El lenguaje ensamblador (`assembler`) es la verdadera base de un programa profesional. Aunque los compiladores actuales realizan buenas optimizaciones de código (lo hacen más reducido y por tanto más rápido), no hay compilador capaz de superar una bien optimizada rutina `assembler`, si bien en el mercado actual nos encontramos con juegos generados casi totalmente en C puro con las rutinas más críticas en `assembler`, lo que proporciona mayor facilidad de programación junto con mayor velocidad.

EL MODO DE VIDEO 320x200x256

Como única desventaja, 320x200 puede ser una resolución algo baja para aplicaciones con muchos datos de texto o para programas que necesiten más área de trabajo que 320 píxeles horizontales por 200 verticales. Esta resolución puede resultar reducida para bases de datos, juegos con mucha área de visión, etc..., pero lo más seguro es que, para empezar, este modo gráfico (conocido como 13h) cubra nuestras necesidades para la mayoría de programas y juegos.

Para programar gráficos en este modo de video principalmente hemos de saber hacer tres cosas:

1. Saber inicializar el modo de video 320x200 ó 13h.
2. Comprender el modo de direccionamiento lineal del 13h y su organización interna.
3. Realizar rutinas gráficas específicas para dicho modo.

Veamos primero como inicializar cualquier modo de video (entre ellos el 13h) utilizando los servicios disponibles en la ROMBIOS del PC.

Supongamos el ejemplo concreto de inicializar el modo de video 320x200 a 256 colores. Si miramos abajo (funciones más importantes de la int 10h) veremos que la interrupción 10h (gestión de la tarjeta gráfica) posee un servicio para inicializar modos de video (servicio 0). Cargamos los registros tal y como los pide la interrupción (AH = servicio, AL = Modo), y efectuamos la llamada:

```
mov ah, 0 /* AH=0: Init VideoMode */
mov al, 13h /* modo: 13h */
int 10h /* llamada a int 10h */
```

Algunos servicios de la int 10h

PARÁMETROS	SERVICIO	FUNCIÓN QUE REALIZA
AH = 00h AL = Modo	Set Video Mode	Inicializa el modo de video especificado en AL, según los valores de la tabla 2.
AH = 0Ch CX = Coord. X DX = Coord. Y AL = Color BH = Página	Write Pixel	Dibuja el pixel (CX,DX) en pantalla con el color AL. El parámetro página debe ser 0 en 13h ya que sólo hay 1 página de video y esa es la 0.
AH = 0Dh CX = Coord. X DX = Coord. Y BH = Página Devuelve: AL = Color	Read Pixel	Devuelve en AL el color del pixel de la posición (CX,DX).
AH = 0Fh Devuelve: AL = Modo	Get Video Mode	Devuelve en AL el modo de video actual.
AX = 1001h BH = Color	Set Border Color	Cambia el color del borde de la pantalla al color BH (por defecto es 0, equivalente al negro).

En la tabla disponemos del listado de los modos de video que pueden ser inicializados en una VGA estándar mediante el servicio 0 de la interrupción 10h. Podemos llamar a este servicio dentro de una función C, como puede verse en el siguiente código:

```
void SetVideoMode ( char modo )
{
asm {
```

```
mov ah, 0
mov al, [modo]
int 10h
}
}
```

MODO	TIPO	RESOLUCIÓN	COLORES	VRAM	SISTEMA
00h	Texto	40x25	16	B800h	CGA/MCGA/EGA/VGA
02h	Texto	80x25	16 grises	B800h	CGA/MCGA/EGA/VGA
03h	Texto	80x25	16	B800h	CGA/MCGA/EGA/VGA
04h	Gráfico	320x200	4	B800h	CGA/MCGA/EGA/VGA
06h	Gráfico	640x200	2	B800h	CGA/MCGA/EGA/VGA
07h	Texto	80x25	mono	B000h	MDA/Herc/EGA/VGA
0Dh	Gráfico	320x200	16	A000h	EGA/VGA
0Eh	Gráfico	640x200	16	A000h	EGA/VGA
10h	Gráfico	640x350	16	A000h	EGA/VGA (256Kb)
12h	Gráfico	640x480	16	A000h	VGA
13h	Gráfico	320x200	256	A000h	MCGA/VGA

Organización interna del modo 13h

Ya sabemos inicializar el modo de video 320x200x256 (y cualquier otro modo de video) mediante una simple llamada a la interrupción 10h. Ahora hemos de comprender como gestiona el ordenador esos píxeles que escribimos en pantalla para empezar a trabajar en 13h. Es algo muy sencillo y visual y requiere tan sólo conocimientos sobre la memoria del PC.

En cierta manera, de toda la memoria del PC los primeros 1024 KiloBytes (1 MegaByte) están divididos en segmentos de 64kb (65.536 bytes) a los que se accede mediante un segmento y un desplazamiento u offset.

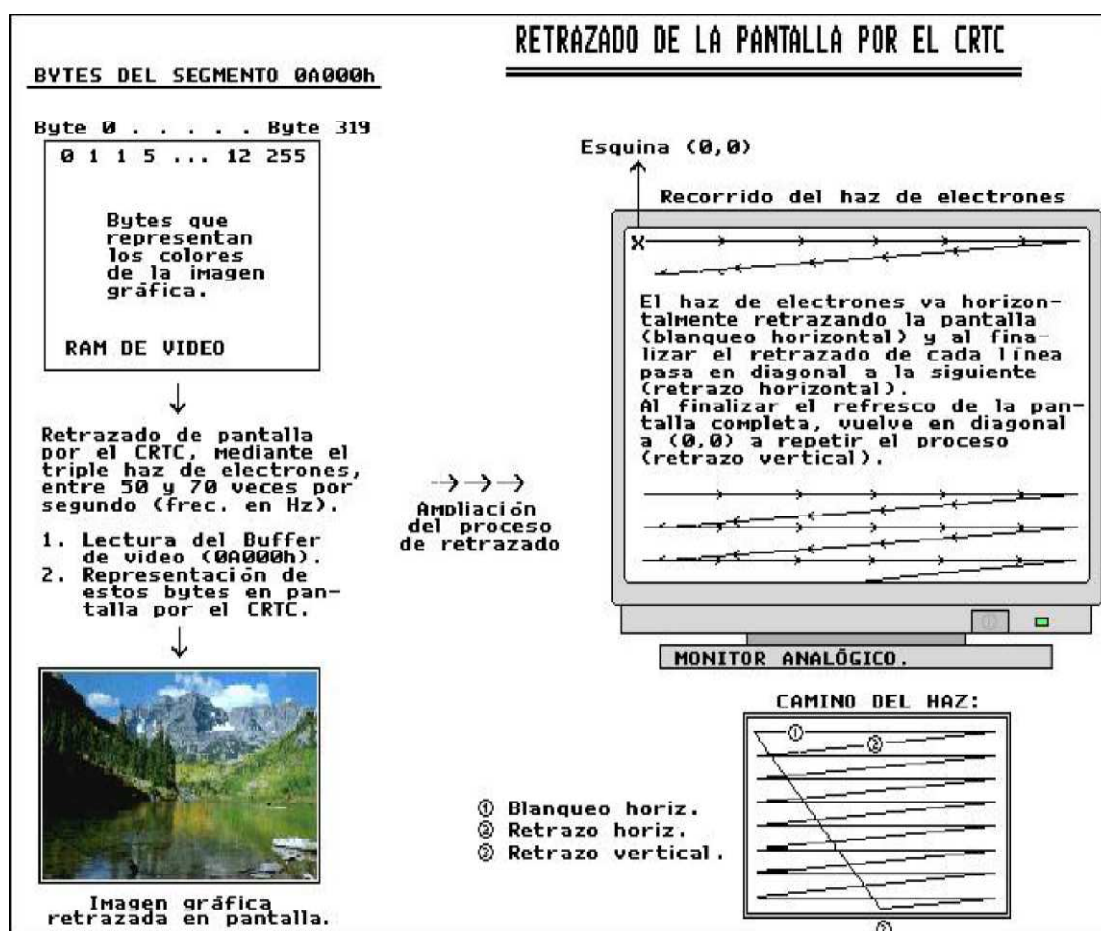
El segmento de memoria (64Kb, 65.536 bytes) es el segmento de VideoRAM, es decir, es donde la VGA guarda los datos de las imágenes gráficas que dibuja en el monitor. Si escribimos algún valor en este segmento, la próxima vez que la tarjeta gráfica redibuje la pantalla (lo hace entre 50 y 70 veces por segundo) el valor que hemos escrito aparecerá en pantalla en forma de punto. Pero veamos qué es lo que hace la tarjeta gráfica con esta VideoMemoria.

El retrazado de pantalla

Aproximadamente entre 50 y 70 veces por segundo (según el modo de vídeo), la tarjeta gráfica lee de su memoria todos los valores que contiene y transforma esta información digital (unos y ceros, o sea: números) en los puntos que vemos en pantalla. Tras nuestro monitor hay un haz de electrones que "bombardea" la pantalla con electrones que producen las diferentes tonalidades. Estas partículas tienden a apagarse, por lo que es necesario redibujar (retrazar) la pantalla el número de veces necesario por segundo para que la imagen no desaparezca.

En este proceso, llamado retrazado de pantalla, el haz de electrones se desplaza hasta la esquina (0,0) del monitor (incluyendo el borde) y comienza a leer bytes de la VideoRAM (segmento 0A000h), transformándolos en píxeles y trazándolos en pantalla. Al llegar al final de una línea horizontal, el haz vuelve en a la siguiente línea (retrazado horizontal) y continúa con el proceso hasta llegar a la esquina inferior derecha, donde vuelve

en diagonal a (0,0) para repetir el proceso. En la figura 1 puede verse el proceso con más claridad. El refresco de la pantalla consiste en un continuo retrazado actualizando la pantalla para ofrecernos la imagen contenida en la videomemoria (que en realidad constituye RAM de la tarjeta a la que se nos permite acceder tras el proceso de autoarranque del encendido).



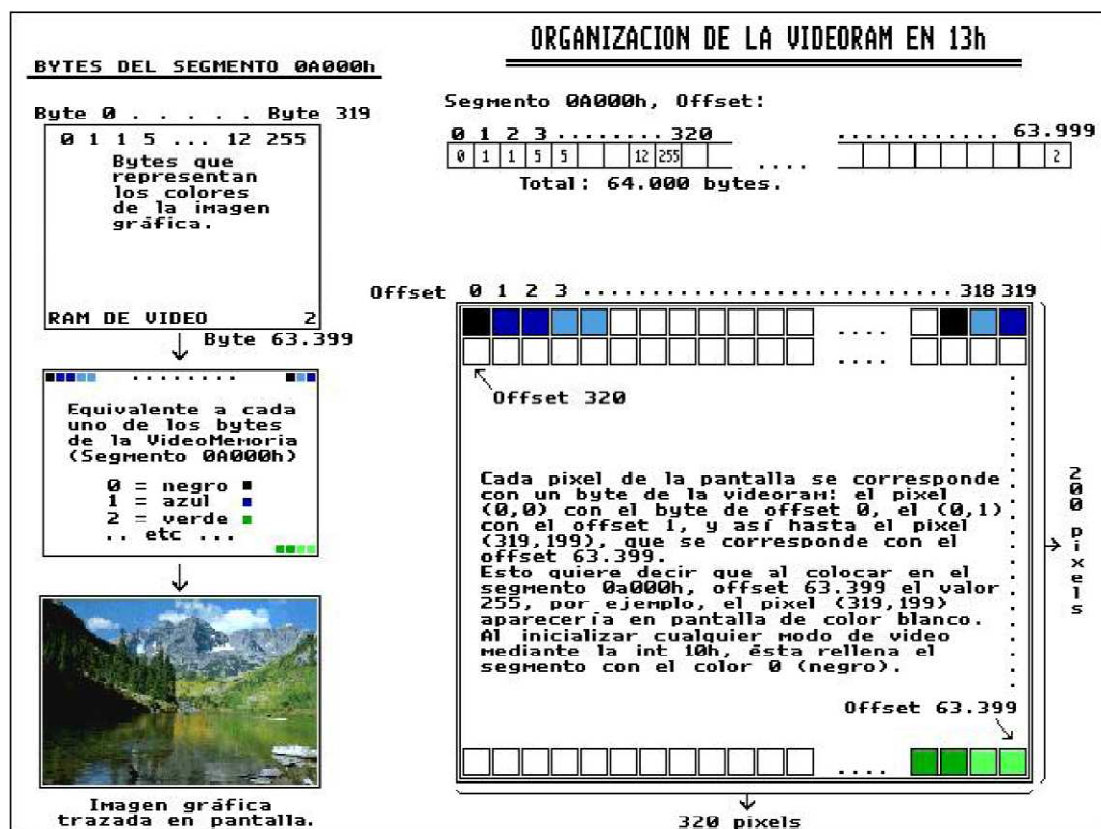
Direccionamiento lineal

Visto de esta manera, sabiendo que la tarjeta transforma los bytes del segmento 0a000h (0xA000 en hexadecimal de C) en píxeles, si averiguamos qué representa cada byte, cuando tratemos de dibujar un pixel bastaría con escribir en este segmento el color que queremos para que la tarjeta gráfica lo represente durante el próximo retrazado. Nada más sencillo en este modo gráfico, modo de 8 bits por pixel.

Esto quiere decir que cada número del 0 al 255 se corresponde con un color. Por defecto, el 0 es el negro, el 1 el azul, y así hasta llegar al 255. Entre el 0 y el 255 disponemos de gamas de azules, verdes, amarillos, etc..., que componen la paleta por defecto de la VGA.

Que este modo gráfico sea de un byte por pixel significa que al escribir un byte en este segmento de memoria, su equivalente en pantalla será un pixel, que aparecerá automáticamente en cuanto el haz de electrones pase por esa posición al refrescar la imagen.

En la figura 2 tenemos una representación de cómo está organizada la VideoRAM en el modo 13h.



Como puede verse, al byte 0 le corresponde el pixel (0,0) (el primero de la pantalla); al byte 1 le corresponde el pixel (1,0), al byte número 320 le correspondería el pixel (0,1), (primer pixel de la línea 1, porque hay 320 píxeles de resolución horizontal) y así hasta el byte 63.999 del segmento, que corresponde a la posición (319,199). Depende del offset en que coloquemos el byte, el punto aparecerá en distinta posición en el monitor (cada byte es un pixel individual en la pantalla).

El segmento de la VideoRAM se comporta en este modo de video como si fuera una larga línea de píxeles de manera que al llegar al final de una línea horizontal de pantalla, el siguiente byte de la VideoMemoria es el que continúa en la siguiente línea de pantalla. De ahí el término direccionamiento lineal: es como si la pantalla fuera un array de C o PASCAL unidimensional desde 0 a 64.000 donde cada 320 bytes estamos situados en una nueva línea de pantalla (el byte 320 es el primer pixel de la segunda línea). Así, durante el retrazado la tarjeta únicamente tiene que dedicarse a leer bytes (todos ellos consecutivos) y representarlos en pantalla. También podríamos comparar la VideoRam con una gran pantalla de una sola línea de ancho (de 320x200=64.000

píxeles de ancho), y cuando la tarjeta traslada esos colores al monitor, cada 320 bytes salta a una nueva línea. Así obtenemos en pantalla una imagen de 320x200 píxeles.

Por otra parte, hay que hacer notar que los 256 colores de que disponemos pueden ser adaptados a nuestras necesidades. La paleta por defecto contiene unos 32 tonos de azul, 32 de rojos, grises, etc... Pero si estamos dibujando (por ejemplo) una selva, probablemente necesitaremos más tonos de verdes de los 32 que hay por defecto. La manera de cambiar la paleta (es decir, que cada número 0255 corresponda a una tonalidad o color) la abordaremos en un próximo artículo, de manera que podríamos (en este ejemplo en concreto) hacer que los primeros 128 colores sean tonos de verdes y los restantes 128 tonos de azules (para el cielo de nuestra selva). Esto quiere decir que el color 0 no tiene porqué ser el negro (como en la paleta por defecto), sino que podemos adaptar cada color (desde el 0 al 255) a la tonalidad (mezcla de rojo, verde y azul) que deseemos.

CÁLCULO DEL OFFSET PARA UN PUNTO

Hemos visto pues que cada byte de la VideoRAM (segmento 0A000h) corresponde a un pixel concreto en pantalla, y que es posible dibujar y leer píxeles tan sólo escribiendo valores en este segmento.

Supongamos que queremos poner un pixel blanco en la posición (0,0) de la pantalla. El pixel (0,0) corresponde a la dirección 0A000:0000h. Si queremos poner este punto con el color blanco (15 en la paleta por defecto), bastaría con

```
asm {  
  mov ax, 0xA000  
  mov es, ax  
  xor di, di  
  mov es:[di], 15  
}
```

Por otra parte, los corchetes de `mov es:[di], 15` significan acceso a memoria, y en esta orden forzamos al micro a escribir el valor 15 en la dirección ES:DI, que apunta a 0A000:0000 gracias las órdenes `mov` anteriores. Esta línea podría haber sido reemplazada por el par de órdenes "`mov al, 15`" y "`stosb`" pero es aconsejable (por cuestiones de velocidad) guardar las instrucciones de cadena (`stosb/stosw/stosd`) para su uso con el prefijo `rep` (bloques completos), porque en la mayoría de los casos resulta más rápido la orden "`mov`" (hasta 3 ciclos más rápido en un 486).

El acceso directo con assembler a la VideoRAM es la manera más directa posible de representar gráficos en pantalla, único camino a seguir al desarrollar aplicaciones y juegos de tiempo crítico (aquellos donde es necesaria mucha velocidad de representación) al igual que hacen los videojuegos de las compañías profesionales, aunque sí es preciso hacer notar que la

VideoRAM es bastante más lenta que la RAM normal, pero esto es algo que solucionaremos más adelante mediante el uso de "pantallas virtuales" (doble buffer).

OFFSET DE UN PIXEL (X,Y)

Resulta bastante intuitivo ver que el offset 0000h corresponde al pixel (0,0), y que el offset 0001h corresponde al pixel (1,0), pero, si queremos modificar o leer el pixel (160, 100), por ejemplo, ¿qué offset tendremos que mover en DI para el acceso a memoria?

De nuevo la solución nos la proporciona la figura 1. Cada 320 bytes de memoria estamos dentro de una nueva línea de pantalla. Esto quiere decir que el byte 320 corresponde al pixel (0,1), el 640 al (0,2), y así hasta el pixel (0,199), que corresponde al offset $199 \times 320 = 63.380$. Analicemos este último valor.

Si el pixel (0,199) está situado en el offset 63.380, el pixel (1,199) será el siguiente $((199 \times 320) + 1 = 63.380 + 1 = 63.381)$.

De ello se deduce que el offset de cualquier pixel (x,y) será:

$\text{offset} = (320 \times y) + x$; Por lo tanto, el pixel (160,100) estará situado en el segmento 0A000h, en el offset $(320 \times 100) + 160 = 32160$. Si escribimos el valor 1 (azul en la paleta por defecto) en esta posición de memoria, en el centro de la pantalla aparecerá un pequeño pixel de color azul.

Así de sencillo es el cálculo del offset correspondiente a cualquier pixel (x,y). Únicamente hemos de saltarnos 320 bytes por cada línea desde el principio del segmento (pixel (0,0) = offset 0) hasta llegar al offset deseado.

Podemos ver la misma función con ensamblador.

Dibujo de píxeles en VideoRam.

Pseudocódigo:

$\text{OFFSET} = (320 \times y) + x$

ES:DI = A000:OFFSET

ES:[DI] = Color

*/

void PutPixel(int x, int y, char color)

```
{
    unsigned int offset_pixel;
    offset_pixel = (320*y)+x;
    asm {
        mov ax, 0xA000
        mov es, ax
        mov di, [offset_pixel]
        mov al, [color]
        mov es:[di], al
    }
}
```

OPTIMIZACIONES A PUTPIXEL

Vamos a suponer que ya han sido asimilados todos los conocimientos explicados hasta ahora. Esto quiere decir comprender cada uno de los pasos que ejecuta PutPixel() y la organización lineal de la VideoMemoria en 320x200x256.

En ese caso podemos pasar a optimizar nuestro PutPixel() para incrementar su eficiencia. Optimizar una rutina significa eliminar operaciones innecesarias y sustituir las operaciones lentas por otras más rápidas con el objetivo de conseguir otra rutina que realice el mismo trabajo pero de manera más rápida, y la velocidad es fundamental en el mundo de los gráficos.

Por una parte ya sabemos: que la orden "mov es:[di], al" es más rápida y eficiente que "stosb", a menos que se trate de escrituras masivas de datos con el prefijo REP.

Por otra parte tenemos la optimización de las multiplicaciones. Cualquier programador de ensamblador sabe que la multiplicación es una de las operaciones más lentas para el microprocesador (entre 139 y 13 ciclos de reloj, según los operandos y el tipo de procesador), y una rutina tan crítica como el cálculo del offset de un punto (que puede ser repetida miles de veces por segundo) es importantísimo que esté lo más optimizada posible. Para ello, vamos a reescribir nuestra rutina PutPixel() usando assembler, eliminando la multiplicación aprovechándonos de un pequeño truco matemático.

MULTIPLICAR MEDIANTE DESPLAZAMIENTOS

Recordemos que los bits de cualquier byte pueden ser desplazados a izquierda y derecha mediante las instrucciones asm shl y shr (Shift Logical Left y Shift Logical Right), o usando los operadores de C << y >>. Veamos el siguiente ejemplo para comprender su utilidad, desplazando los bits de AX 1 y 2 veces a la izquierda (shl 1 y shl 2):

```
AX = 00010110b (22 decimal)
SHL AX, 1 = 0101100b (44 decimal)
SHL AX, 2 = 1011000b (88 decimal)
```

Desplazar un número una posición a la izquierda es equivalente a multiplicarlo por 2. Desplazarlo 2 veces es equivalente a multiplicarlo por 4, y así sucesivamente con las potencias de 2 (desplazarlo 4 veces es el equivalente a multiplicar por 2 elevado a 4: 16). Exactamente lo mismo ocurre con SHR, que equivale a dividir por 2, 4, 8, 16, 32, 64, 128, 256, 512, etc... según el número de bits que desplazemos. Además, estas operaciones de multiplicación y división mediante desplazamientos son casi instantáneas (no olvidemos que son simples operaciones lógicas de bits, realizándose en 1 ó 2 ciclos de reloj), resultando mucho más rápidas que la multiplicación normal (hasta 139 ciclos).

Pero por desgracia, 320 no es una potencia de 2, por lo que no podemos multiplicar directamente usando desplazamientos... excepto aprovechándonos de la propiedad distributiva del producto según la cual una multiplicación puede dividirse en sumas de multiplicaciones (por ejemplo: $x*4 = x*2 + x*2$, ya que $2+2=4$).

La multiplicación por 320 es el equivalente a multiplicar por 256 más multiplicar por 64 ($256+64=320$):
 $num*320 = (num*256) + (num*64)$

Ahora sí, 256 y 64 son potencias de 2 y podemos utilizar desplazamientos para multiplicar, como en la función PutPixel() $offset = (y<<8) + (y<<6) + x$;

Píxeles en VideoRam mediante shifts

```
/*PutPixel()
```

```
; Dibujo de píxeles en VideoRam
```

```
multiplicando con desplazamientos.
```

```
Pseudocódigo:
```

```
OFFSET = (y*256)+(y*64)+x
```

```
ES:DI = A000:OFFSET
```

```
ES:[DI] = Color
```

```
*/
```

```
void PutPixel( int x, int y, char color )
```

```
{
```

```
    unsigned int offset_pixel;
```

```
    offset_pixel = (y<<8)+(y<<6)+x;
```

```
    asm {
```

```
        mov ax, 0xA000
```

```
        mov es, ax
```

```
        mov di, [offset_pixel]
```

```
        mov al, [color]
```

```
        mov es:[di], al
```

```
    }
```

```
}
```

Esta rutina es mucho más rápida que la primera que desarrollamos y por tanto podremos poner más píxeles en pantalla en menos tiempo. La optimización de código es otro punto importante de la programación gráfica, ya que arañar algún ciclo de reloj puede parecer poco importante, pero si es una rutina que se ejecuta miles de veces por segundo, el incremento de velocidad puede ser notable.

La única desventaja de los desplazamientos radica en que las instrucciones "SHL reg, n" y "SHR reg, n", requieren un procesador 286 o superior, por lo que habremos de compilar nuestros programas con la opción de generar código 286 y ya no funcionarán en los antiguos 8086 (que por otra parte prácticamente han desaparecido).

USO DE TABLAS PRECALCULADAS

Como alternativa para programadores más expertos, también podemos precalcular tablas con las multiplicaciones de $y \cdot 320$ y de manera que el cálculo del offset se limitaría a una simple línea como esta:

```
offset = tabla[y] + x;
```

siendo `tabla[]` un array de 320 elementos conteniendo desde 0 a 320 todos los valores de $320 \cdot y$ precalculados al inicio del programa (0, 320, 640, etc... hasta 63.680, que sería lo que contendría `tabla[199]`).

The TwoBytes structure defines two byte fields, LoByte and hiByte. The union ByteWord also defines two fields. First is asBytes, of the previously defined TwoBytes structure. Next is asWord, a single 16-bit word. Variables of type ByteWord make it easy to refer to locations as both word and double-byte values without the danger of forgetting that words are stored in byte reversed order –a problem with the LABEL method. To use the nested union, first declare a variable, in this case assigned the value of 0FF00_h.

```
DATASEG
data      ByteWord      <,0FF00h>
```

You can now refer to data as a TwoBytes structure or as a 16-bit word. A short example demonstrates how to load the same memory locations into either byte or word registers. Because the TwoBytes structure is nested inside the union, two periods are required to “get to” the byte fields. Notice how the field names reduce the danger of accidentally loading the wrong byte of a word into an 8-bit register:

```
CODESEG
mov      al, [data.asBytes.LoByte]    ;      Load LSB into al
mov      ah, [data.asBytes.hiByte]    ;      Load MSB into ah
mov      ax, [data.asWord]            ;      Same result
```

Declaring RECORD Types

RECORD is a directive that lets you give names to bit fields in bytes and words. You simply specify the width of each field –in other words, the number of bits the field occupies. Turbo Assembler then calculates the position of the field for you. For example, this RECORD defines signedByte as an 8-bit value with two fields:

```
RECORD    signedByte      sign:1,      value:7
```

After the RECORD directive comes the record’s name, followed by a series of named fields. Each field name ends with a colon and the width of the field in bits. The sign field in this example is 1 bit long. The value field is 7 bits long. Separate multiple fields with commas. If the total number of bits is less or equal to 8, Turbo Assembler assumes the record is a byte; otherwise, it assumes the record is a word. You can’t construct records larger than a word, although you can create multifield structures containing multiple bit fields, which would accomplish the same thing. You don’t have to specify exactly 8 or 16 bits, although most programmers do, inserting dummy fields to flesh out a bit record to account for every bit, whether used or not.

Creating variables of a RECORD type is similar to creating variables of structures and unions. In fact, the three forms appear identical, leading to much confusion over the differences between structures and records. A few samples will clear the air:

```
DATASEG
v1      signedByte      <>          ;      default values
v2      signedByte      <1>         ;      sign = 1, value = default
v3      signedByte      <,5>        ;      sign – default, value = 5
v4      signedByte      <1,127>     ;      sign = 1, value = 127
```



```
v5    signedByte    <3,300>    ;    sign = 1, value = 44
```

A record variable declaration has three parts: a label, the RECORD name, and two angle brackets with optional values inside. The first sample declares v1 as a variable of type signedByte. Because no values are specified in brackets, the default values for all bit fields are used. (In this case, the defaults are 0. In a moment, you'll see how to set other defaults). The second sample sets the sign bit of v2 to 1, leaving the value field equal to the default. The third line sets value to 5, letting the sign field assume the default value. The fourth line assigns values to both fields in the variable, setting sign to 1 and value to 127. The fifth line shows what happens when you try to use out-of-range values such as 3 and 300. In this case, the actual values inserted into the record equal the attempted values modulo (division remainder) 2^n , where n equals the number of bits in the field.

Setting Default Bit-Field Values

Normally, the default field values in RECORD variables are 0. To change this, add to the field width an equal sign and the default value you want. For example, to create a RECORD with an MSD default of 1 and a second field defaulting to 5, you can write:

```
RECORD    minusByte    msign:1 = 1,    mvalue:7 = 5
```

Declaring a variable of this type with empty angle brackets sets the msign field to 1 and the mvalue field to 5. Specifying replacement values in brackets as explained before overrides these new defaults. Notice that different field names are used here. Even though the names are contained in the RECORD definition, Turbo Assembler considers these names to be global –active at all places in the program or module. Therefore, you must use unique field names among all your RECORD definitions in one module.

Using RECORD Variables

After declaring a RECORD type and a few variables of that type, you can use several different methods to read and write bit-field values in those variables. To demonstrate how to do this, we first need a new RECORD type:

```
RECORD person sex:1, married:1, children:4, xxx:1, age:7, school:2
```

RECORDs like this one can pack a lot of information into a small space. In this example, only 16 bits are needed to store five facts about a person –with field sex equal to 0 for male and 1 for female, married equal to 0 if false or 1 if true, children ranging from 0 to 15, a 1-bit dummy field xxx reserved for future use, an age field ranging from 0 to 127, and school from 0 to 3, representing four levels of a person's schooling. Figure 6.4 illustrates how these fields are packed into one 16-bit word. As with all 16-bit values, the two 8-bit bytes of this variable are stored in memory in reverse order, with bits 0-7 (LSB) at a lower address than bits 8-15 (MSB).

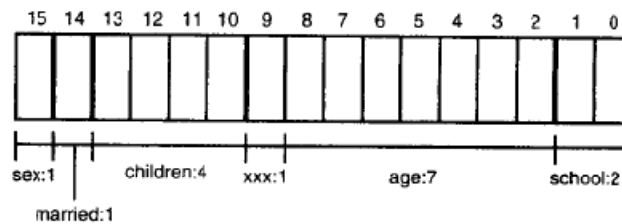
What's in a Field Name?

Turbo Assembler converts bit-field names into the number of right shifts required to move the field to the rightmost position in the byte or word. The value is equal to the byte or word bit position of the least significant digit for this field. Referring to the person record, then, sex = 15, married = 14, children = 10, xxx = 9, age = 2, and school = 0. (See figure 6.4). You can use these field name constants as simple EQU equates.

Normally, though, you'll use the values to shift bit fields into the rightmost position in a register, making it easy to process individual field values. The process works in reverse, too. If the children bit-field value is already in the rightmost position of ax, shifting ax left by the value of children moves the bit-field value into its proper position, ready to be packed into the record.

Figure 6.4.

A record packed with six bit fields stores a lot of information in a small space.



Using field names instead of manually counting bits saves time and helps prevent bugs. For example, to increment the `age` field, you can shift the appropriate bit-field value to the rightmost position in a word register, increment the register, and then shift the result back into position. Before doing this, however, you must strip out other bits from the variable. To help with this step, Turbo Assembler provides an operator called `MASK`, which takes the name of a bit field and generates an appropriate mask with bits equal to 1 in all positions for this field. A good way to organize your masks is to use names similar to the associated fields:

```
maskSex      = MASK sex
maskMarried  = MASK married
maskChildren = MASK children
maskAge      = MASK age
maskSchool   = MASK school
```

Each new identifier—for example, `maskSex` and `maskMarried`—is assigned a mask for each bit field (except for `xxx`, which we'll just ignore). The names make the purpose of the various symbols easy to remember, although you can use whatever names you like. You don't have to preface the identifiers with "mask." With the bit-field names and masks, it's easy to isolate and process bit-field information without having to calculate the positions of fields in records. An example explains how this works. First, declare a variable named `subject` of type `person`:

```
DATASEG
subject    person    <>
```

Then, to set single bit fields to 1, use `or` to combine the mask with the record's current value:

```
CODESEG
or  [subject], maskSex      ; Set sex field = 1
or  [subject], maskMarried  ; Set married field = 1
```

To reset single-bit fields to 0, use the `NOT` operator along with the bit mask, toggling all bits in the mask. The following shows two ways to proceed:

```
and [subject], NOT maskSex ; Change sex field to 0
mov ax, [subject]          ; Load subject into ax
and ax, NOT maskMarried    ; Change married field to 0
mov [subject], ax          ; Store result back in memory
```

Extracting Bit Fields

For bit fields of more than 1 bit, the process is similar but requires additional steps to isolate the values. There are several possible methods you could use, but these steps always work:

1. Copy the original variable into a register
2. AND the register with the field mask
3. Shift the register right by the field-name constant

After copying the variable into a register (either 8 or 16 bits wide, depending on the variable's size), step 2 isolates the field's bits, stripping other fields out of the record, thus setting all other bits but those in the desired field to 0. Step 3 then shifts the isolated field bits to the rightmost position in the register. To add a new member to our subject's family, use these steps:

```
mov ax, [subject]      ; Step 1--copy the variable
and ax, maskChildren   ; Step 2--isolate the bit field
mov cl, children        ; Prepare shift count
shr ax, cl              ; Step 3--shift field to right
inc ax                  ; Add 1 to number of children
```

The `mov` and `and` instructions copy the subject variable into `ax` and strip other fields out of the value, leaving only the bits that apply to `children`. After loading the shift count into `cl`, the `shr` instruction shifts the `children` field to the far right of `ax`, preparing for `inc` to increment this value. If the `children` field was already rightmost in the variable—making the shift count equal to 0—the shift instructions can be skipped. For example, you could write:

```
mov cl, children        ; Move shift count into cl
or cl, cl                ; Is count = 0?
jz @@10                 ; Jump if yes, cl = 0
shr ax, cl               ; Else shift ax, cl times
@@10:
inc ax                  ; Add 1 to number of children
```

A better approach is to use a conditional `IF` directive, which Chapter 8 explains in more detail. This lets the assembler, rather than the program, decide whether shifting is required. After completing steps 1 and 2 to copy and mask the record variable, the following instructions shift the result right only if the `children` constant is greater than 0:

```
IF children GT 0
    mov cl, children      ; Move nonzero count into cl
    shr ax, cl            ; Shift ax, cl times
ENDIF
inc ax                    ; Add 1 to number of children
```

If the expression in the conditional `IF` is true, then Turbo Assembler assembles the code up to the next `ENDIF` directive. If the expression is false, then the code is ignored. This method eliminates the unnecessary comparison, jump, and shift instructions of the previous technique.

Recombining Bit Fields

After extracting a bit field and processing its value, you now need a way to insert the result back into a record variable. Assuming the result is rightmost in a register, follow these four steps:

1. Shift the register left by field-name constant
2. AND the register with the field mask

3. AND the original value with NOT field mask
4. OR the register into the original value

Step 1 shifts the value into its correct position, again using the field name as the shift count but this time shifting left instead of right. Step 2 is an optional safety valve, which limits the new value to the field's width in bits. If you are positive that the new field value is within the proper range, you can skip this step. But any out-of-range values—accidentally giving our subject the burden of 45 children, for example—can change the values of other fields. For this reason, it's a good idea to mask the new value this way before combining the value back into the original variable. Step 3 complements step 2 by setting all bits of the field in the original value to 0—in a sense, punching a hole in the original value like a cookie cutter punching out a circle in dough. Step 4 then Ors the new value into this punched-out hole, completing the process.

To demonstrate these four steps in assembly language, the following code fragment moves the children field (now rightmost in register ax) back into the subject variable:

```
mov  cl, children          ; Move shift count into cl
shl  ax, cl                ; Step 1--shift into position
and  ax, maskChildren      ; Step 2--Limit value
and  [subject], NOT maskChildren ; Step 3--punch a hole
or   [subject], ax         ; Step 4--drop value into hole
```

As with the previous steps that extract a bit field, you can use a conditional IF directive to skip the shift if children = 0, indicating that this field is already rightmost in the variable. Also, you can eliminate the first and if the result cannot possibly be larger than 15—the maximum value that the 4-bit children field can express.

Putting the extraction and recombination steps together, here's another example that adds 10 to our subject's age field:

```
mov  ax, [subject]         ; Copy the variable into ax
and  ax, maskAge           ; Isolate the age field
mov  cl, age               ; Prepare shift count
shr  ax, cl                ; Shift age field to right
add  ax, 10                ; Age 10 to subject's age
shl  ax, cl                ; Shift age back into position
and  ax, maskAge           ; Limit age to maximum range
and  [subject], NOT maskAge ; Punch a hole in (zero) age field
or   [subject], ax         ; Drop new age value into hole
```

Many programmers avoid using RECORD bit fields, probably because they do not understand the techniques. This fact is evident from the many assembly language programs that declare fixed constants for shift values and masks, making the code much more difficult to modify. If you take the time to learn how to use RECORD and MASK, defining your packed records as described here, you'll be able to write programs that automatically adjust for new situations—a change to the number of bits in the school field or a newly found uses for the reserved xxx single-bit field. You can also change the default values assigned to fields without having to

hunt through a lot of cryptic statements, making changes to programs that don't need fixing! Just change your RECORD definitions, and you're done. The same advantages apply to STRUC and UNION, which help take much of the complexity out of working with complex data structures.

Efficient Logical Operations

The saying “There’s always room for improvement” is especially true in assembly language. One improvement that’s often missed is the replacement of word-based instructions for shorter, and potentially faster, byte-based instructions that perform identical jobs in certain situations.

For example, when testing a bit in a record, or when setting or exclusive-ORing bits, it’s possible to use a byte-based instruction even when operating on a 16-bit word value when the target bit is in the low-order portion of the word. An example will help clarify the problem and its solution. Consider the following bit-field record:

```
RECORD BitRec b0:1, b1:4, b2:3, b3:7
```

Logical and, or, test, and xor instructions can manipulate bits in `BitRec` record variables by referring to the `b0`, `b1`, `b2`, and `b3` labels. You can, for instance, set bit `b2` in the `ax` register with the instruction:

```
or ax, b2
```

When assembled, this generates a word-based instruction that takes three machine code bytes:

```
0D 07 00
```

That same instruction, however, is more efficiently coded as follows, which performs the identical job and has the same effect on processor flags:

```
or al, b2
```

When assembled, this instruction takes only two machine code bytes:

```
0C 07
```

Even though the variable is in the 16-bit register `ax`, an 8-bit instruction that refers to the 8-bit low-order byte register `al` has the identical effect.

Automating Efficient Logical Operations

To automate the selection of efficient logical instructions, Turbo Assembler 3.0 and later versions provide four pseudo instructions: `SETFLAG`, `MASKFLAG`, `TESTFLAG`, and `FLIPFLAG`. With them, the assembler can choose the most efficient forms of logical instructions automatically. For example, the assembler replaces this instruction:

```
SETFLAG ax, b2
```

with the more efficient:

```
or ax, 07
```

rather than the equivalent, but less efficient, instruction that might appear to be necessary:

```
or ax, b2
```

The following code snippet shows how to use the pseudo instructions. Comments show the assembled code. For example, in the first line, the SETFLAG instruction is encoded as a byte-based logical or instruction. The equivalent, but potentially inefficient, instruction follows on the second line. Notice that in the case of logical and, in this example, a byte-based instruction replacement is not possible:

```
SETFLAG ax, b2 ; or ax, 07 / 0C07
or ax, b2 ; or ax, 0007 / 0D0700
MASKFLAG ax, b2 ; and ax, 0007 / 250700
and ax, b2 ; and ax, 0007 / 250700
TESTFLAG ax, b2 ; test al, 07 / A007
test ax, b2 ; test ax, 00007 / A90700
FLIPFLAG ax, b2 ; xor al, 07 / 3407
xor ax, b2 ; xor ax, 0007 / 350700
```

Automating Record Field Operations

Turbo Assembler 3.0 and later versions provide two additional pseudo instructions, SETFIELD and GETFIELD that greatly simplify working with bit-field records. Before using them, you should be familiar with the discussions in this chapter on using record variables along with MASK values to set and retrieve bit values packed into bytes and words.

A few examples show how these new instructions can simplify the steps for inserting and extracting person record fields. So you don't have to flip pages, here is the record declaration again:

```
RECORD person sex:1, married:1, children:4, xxx:1, age:7, school:2
```

As you learned, it takes a combination of shift, rotate, and logical instructions to set and retrieve values in person record fields, but Turbo Assembler 3.0 and later versions can create the necessary instructions for you. For example, first prepare a register to hold a person record:

```
xor ax, ax ; Clear person record
```

That simply clears register ax to zero. To insert a value into the record's children field, first assign the value to a register (b1 here), and use SETFIELD as follows:

```
mov b1, 3 ; Move no. children to b1
SETFIELD children ax, b1 ; Set children field in ax
```


The second line inserts the value of `b1` into `ax` *without disturbing other bits in ax*. To do that, Turbo Assembler writes the following logical operations in place of the `SETFIELD` pseudo instruction:

```
rol  b1, 02
or   ah, b1
```

The first instruction rotates the `children` value left two positions, and the second instruction logically ORs that value into `ax`. The assembler also chooses a more efficient byte-form of the logical `or` rather than operating on the full 16-bit word.

You can use `SETFIELD` similarly to insert values into any record field—an age value, for example:

```
mov     b1, 43           ; Move age to b1
SETFIELD age ax, b1      ; Set age field in ax
```

This generates another set of rotate and logical operations to insert into `ax` an age value from `b1`, without disturbing other record fields.

To extract bit-field values from records, use `GETFIELD`. For example, the following instruction sets `b1` to the number of children in the person record held in register `ax`:

```
GETFIELD children b1, ax ; Get children into b1 (destroys bh!)
```

Assuming the preceding `SETFIELD` instructions were executed, this sets `b1` to 03. In place of the pseudo `GETFIELD` instruction, Turbo Assembler writes the following instructions:

```
mov  b1, ah
ror  b1, 02
and  b1, 0F
```

The first line moves the portion of the record that contains the desired bit-field value (`ah`) into `b1`. The second line rotates that value right two positions, moving it to the rightmost spot in `b1`. The third line applies the literal mask `0Fh` to isolate the desired value, which in this example, sets `b1` to 03.

Similarly, you can use `GETFIELD` to extract the age value from the record in `ax`:

```
GETFIELD age b1, ax      ; Get age into b1 (destroys bh!)
```

The assembler generates another set of logical operations that in this case set `b1` to 43, the age value packed in the record.

One danger with `GETFIELD` is that it always uses the full 16-bit target register, even though you specify only the low-order portion. In the preceding two `GETFIELD` examples, as the comments indicate, the most significant byte in `bh` is destroyed by the logical instructions that Turbo Assembler creates.

You may use other registers and memory references with `SETFIELD` and `GETFIELD`—you don't have to use `ax` and `b1` as demonstrated here. The full syntax for both pseudo instructions follow:

```
SETFIELD field_name destination_r/m, source_reg
GETFIELD field_name destination_reg, source_r/m
```

Use these rules to construct SETFIELD and GETFIELD instructions. Each requires a field name followed by destination and source specifications. The destination for SETFIELD may be a register or a memory reference. Its source must be a register. The destination for GETFIELD must be a register. Its source may be a register or memory reference.

Using Predefined Equates

Turbo Assembler knows a few predefined equates that you can use as default values for program variables. Table 6.1 lists these equates, all of which begin with two question marks.

Table 6.1. Predefined Equates.

<i>Symbol</i>	<i>Meaning</i>
??Date	Today's date in the DOS country-code style
??Filename	The module or program's disk-filename
??Time	The current time in the DOS country-code style
??Version	Turbo Assembler version number

Converting Numbers and Strings

In high-level languages, you can read and write numeric values directly. For example, to let someone enter a number and then display the result, assuming *n* is an integer, you might use these Pascal statements:

```
Write( 'Enter a value: ' );
ReadLn( n );
WriteLn( 'Value is: ', n );
```

Native assembly language lacks similar abilities. Instead, you have to read and write strings and then convert those strings to and from binary values for processing, storing on disk, and so on. Of course, high-level languages must do this internally, too!

Listing 6.3, BINASC.ASM, is a module that you can use to make this process easier to program. The module has routines that can convert 16-bit values to and from signed and unsigned decimal, hexadecimal, and binary ASCII strings. Assemble to BINASC.OBJ and store this code in your MTA.LIB file with the commands:

```
tasm /z binasc
tlib /E mta -+binasc
```

As with the modules in Chapter 5, ignore the warning that BINASC is not in the library. It won't be until you install it the first time. Also, be aware that BINASC uses two procedures from STRINGS; therefore, you won't be able to link programs to BINASC until at least both of these modules are installed in MTA.LIB.

title conversion a binario desde o hacia ASCII

ideal

Miguel Israel Barragán Ocampo

model small

; ----- equ's

ASCNull EQU 0 ; caracter de nulo en ASCII
dataseg

codeseg
; ---- de Strings.obj

EXTRN StrLength:proc, StrUpper:proc
PUBLIC HexDigit, ValCh, NumToAscii
PUBLIC BinToAscHex, SBinToAscDec, BinToAscDec, BinToAscBin
PUBLIC AscToBin

; -----
; HexDigit Convierte un valor de 4 bit a un dígito en ASCII
; -----
; Entrada:
; dl = valor limitado al rango de 0..15
; Salida:
; dl = dígito hexadecimal equivalente
; Registros:
; dl
; -----

PROC HexDigit
cmp dl,10 ; dl es < 10 (hex 'A')?
jb @@10 ; si sí, salta a @@10
add dl, 'A'-10 ;sino convierte a A,B, C, D, E, o F
ret ; regresa a quién llamo

@@10:
or dl,'0'; convierte dígitos 0 a 9
ret

endp HexDigit

; -----
; ValCh Convierte el caracter del dígito en ASCII a un valor binario
; -----

; entrada:
; dl = dígito en ASCII '0'..'9'; 'A'..'F'
; bx = base (2=binario, 10=decimal, 16=hexadecimal)
; salida:
; cf = 0; dx = valor equivalente binario
; cf = 1; mal caracter para esta base (dx no tiene nada)
; Registros:
; dx
; -----

PROC ValCh

```

    cmp dl,'9' ; checar para un dígito posiblemente hex
    jbe @@10 ; probablemente '0'..'9', salta
    sub dl, 7 ; ajusta dígito hex a 3A..3F
@@10:
    sub dl,'0' ; Convierte de ascii a decimal
    test dl, 0f0h ; checa los 4 msbs (pone cf = 0)
    jnz @@99 ; salta a @@99 si hay error (no dígito o A-F)
    xor dh,dh ; convierte el byte en dl a una palabra en dx
    cmp dx,bx ; compara con la base (cf = 1 si ok)
@@99:
    cmc ; complementa cf para poner 1/0 en bandera de error
    ret
endp ValCh

; -----
; NumToASCII Convierte un valor binario sin signo a ASCII
; -----
; Entrada:
;     ax=valor de 16 bits a convertir
;     bx=base del resultado
;     cx=número mínimo de dígitos para la salida
;     di=dirección que contendrá el resultado
;     Nota: se asume que la cadena es lo suficientemente grande para contener el resultado
;     Nota: crea resultado completo si cx es menor que el numero de
;           dígitos requeridos para especificar el resultado o cx=0
;     Nota: si cx = 0 y ax = 0 entonces la longitud de la cadena será cero
;           poner cx = 1 si se quiere que la cadena = '0' si ax = 0
;     Nota: asume que (2<=bx<=16)
; Salida:
;     ninguna
; Registros:
;     ax,cx
; -----
PROC NumToASCII
    push dx
    push di
    push si
; si = cuenta de digitos en el stack
    xor si,si ; si = 0
    jcxz @@20 ; si cx = 0, salta a poner cx =1
@@10:
    xor dx,dx ; para extender ax a 32 bit dx:ax
    div bx ; ax <- ax:dx div bx dx<- residuo
    call HexDigit ; convierte dl a un dígito ASCII
    push dx ; salva el dígito en el stack
    inc si ; cuenta los dígitos en el stack
    loop @@10 ; Salta en el mínimo de cuenta de dígitos

```

```

@@20:
    inc cx ; cx=1 en caso de que no se haya hecho
    or ax,ax ; es ax=0? (todos los dígitos hechos)
    jnz @@10 ; si ax <> 0, entonces continua la conversión
    mov cx,si ; poner en cx la cuenta de caracteres en el stack
    jcxz @@40 ; salta el siguiente loop si cx = 0000
    cld ; auto incrementa di para stosb
@@30:
    pop ax ; pop el siguiente dígito en al
    stosb ; guarda el dígito en una cadena:avanza di
    loop @@30 ; salta por los cx dígitos
@@40:
    mov [byte di], ASCNull ; guarda null al final de la cadena
    pop si
    pop di
    pop dx
    ret
endp NumToASCII
; -----
; BinToAscHex Convierte de valores binarios a cadenas en hex en ASCII
; -----
; Entrada:
;     ax= valor de 16 bits a convertir
;     cx = el valor mínimo de dígitos a salida
;     di = dirección de la cadena que contendrá el resultado
;     Nota: se asume que la cadena es lo suficientemente larga para contener el resultado
;     Nota: da resultado completo si cx es menor que el número de dígitos requeridos para
;           especificar el resultado
; Salida:
; ninguna
; Registros:
; ax,cx
; -----
PROC BinToAscHex
    push bx ; salva bx en el stack
    mov bx, 16 ; base = 16 (hex)
    call NumToAscii ; convierte ax en ASCII
    pop bx ; restablece bx
    ret
endp BinToAscHex
; -----
; BinToAscDec Convierte de valores binarios a cadenas ASCII decimales
; -----
; Entrada:
;     lo mismo que BinToAscHex
; salida:
;     ninguna

```

```
; Registros:
;     ax,cx (indirectamente)
; -----
PROC BinToAscDec
    push bx ; salva bx en el stack
    mov bx,10 ; pone base=10
    call NumToAscii ; convierte ax a ASCII
    pop bx ; restablece bx
    ret
endp BinToAscDec
; -----
; SBinToAscDec Convertir binarios con signo a cadenas ASCII decimales
; -----
; Entrada:
;     lo mismo que BinToAscHex (ax = valor de 16 bits con signo)
; Salida:
;     ninguno
; registros:
;     ax, cx
; -----
PROC SbinToAscDec
    push bx ; salvar bx y di
    push di
    cmp ax, 0 ; tiene signo ax < 0?
    jge @@10 ; salta si ax >= 0
    neg ax ; complemento a dos de ax
    mov [byte di], '-' ; insertar '-' en la cadena
    inc di ; avanzar el apuntador de la cadena
@@10:
    mov bx, 10 ; poner base = 10
    call NumToAscii ; convertir ax a ASCII
    pop di
    pop bx
    ret
endp SBinToAscDec
; -----
; BinToAscBin Convierte los valores binarios a cadenas binarias ASCII
; -----
; Entrada:
;     Lo mismo que BinToAscHex
; Salida:
;     Ninguna
; Registros:
;     ax, cx (indirectamente)
; -----
PROC BinToAscBin
    push bx
```

```

    mov bx,2 ; pone la base = 2
    call NumToAscii ; convertir ax a ASCII
    pop bx
    ret
endp BinToAscBin
; -----
; ChToBase Regresar el número de la base por un string
; -----
; Nota:
;     Subrutina Privada para AscToBin. no llamar directamente
; Entrada:
;     si = apuntador al null al final de la cadena
;     Nota: asume que la longitud de la cadena es >=1
; Salida:
;     bx=2 (binario) , 10(default), 16
;     si= dirección de el último caracter dígito probable en la cadena
; Registros :
; bx,dl,si
; -----
PROC ChToBase
    mov dl, [byte si-1]; obtiene el último caracter de la cadena
    mov bx,16 ; base=16
    cmp dl,'H' ; es una cadena en hexadecimal?
    je @@10 ; salta si es hex
    mov bx,2 ; base = 2
    cmp dl, 'B' ; es una cadena binaria
    je @@10
    mov bx, 10; base =10
    cmp dl, 'D'
    jne @@20 ; salta si no es decimal
@@10:
    dec si ; ajusta si al último dígito probable
@@20:
    ret
Endp ChToBase
; -----
; AscToNum Convierte caracteres ASCII a binario
; -----
; Nota:
;     Rutina privada para AscToBin, no llamar directamente
; Entrada:
;     ax=valor inicial 0
;     bx=base
;     di=dirección de cadena sin signo (cualquier formato)
;     si=dirección del último dígito probable en la cadena
; Salida:
;     cf=0: ax= variable sin signo

```

```
;      cf=1: caracter no correcto en la cadena (sin significado)
;      Registros:
;      ax, cx, dx, si
;-----
PROC AscToNum
    mov cx,1 ; inicializa el multiplicador
@@10:
    cmp si,di ; estamos al frente de la cadena?
    je @@99 ; salir si cf=0
    dec si ; hacer el siguiente caracter a la izq
    mov dl,[byte si] ;cargar caracter en dl
    call ValCh ; convertir dl al valor en dx
    jc @@99 ; salir si hay error (caracter sin significado)
    push cx ; salvar cx en el stack
    xchg ax,cx ; ax= multiplicador; cx=valor parcial
    mul dx ; dx:ax <- valor del dígito * multiplicador
    add cx,ax ; cx <- cx +ax (nuevo valor parcial)
    pop ax ;Restablecer el multiplicador a ax
    mul bx ; dx:ax <- multiplicador * base
    xchg ax,cx ; ax = valor parcial; cx = nuevo multiplicador
    jmp @@10; hace el siguiente dígito
@@99:
    ret
endp AscToNum
;-----
; AscToBin Convertir cadenas ASCII a valores binarios
;-----
; Entrada :
;      si = cadena ASCII para convertir a binario
;      'H' al final de la cadena = hexadecimal
;      'B' al final de la cadena = binario
;      'D' o dígito al final de la cadena = decimal
;      '-' EN S[0] indica un número negativo
; Nota: no se permiten espacios en blanco en la cadena
; Salida:
;      cf = 1; caracter sin significado
;      cf = 0 ; ax =valor de una cadena
;      Nota: los caracteres en una cadena son convertidos a mayúsculas
;      Nota: los caracteres nulos (null) ponen ax=0
; Registros:
; ax
;-----
PROC AscToBin
    push bx ; Salvando registros (algunos de estos son cambiados en subrutinas llamadas desde este procedimiento)
    push cx ;
    push dx
    push si
```

```

call StrUpper ; convierte cadena a mayúsculas
call StrLength ; pone cx la longitud de la cadena en di
xor ax, ax ; inicializa cf=0
jcxz @@99 ; salir si la longitud = 0 ax=0 cf=0
mov si,di ; direcciona la cadena en di con si
add si,cx ; avanza si al final null de la cadena
cmp [byte di], '-' ; checa por el signo menos
pushf ; salva el resultado de la comparación
jne @@10 ; Salta si el signo menos no es encontrado
inc di ; avanza di pasando el signo menos
@@10:
call ChToBase ; pone bx=numero base; si en el último dígito
call AscToNum ; convierte ASCII (base bx) al número
rcl bx,1 ; preserva cf por el shift de bx
popf; restaura banderas de el chequeo de el signo menos
jne @@20 ; salta si el signo de menos no se encontró
neg ax ; si no, forma el complemento a dos
dec di; restaura di a la cabeza de la cadena
@@20:
    rcr bx,1 ; restaura el resultado de cf de AscToNum
; push cx FALTA po ahí!!!
@@99:
    pop si
    pop dx
    pop cx
    pop bx
    ret
endp AscToBin
end ; fin del módulo

```

usando BINASC

```

title Despliega el equipo de la PC
    ideal
    model small
    stack 256
; ---- Equ's

```

```

EOS equ 0 ; fin de cadena
cr equ 13 ; retorno de carro ASCII
lf equ 10 ; line feed ASCII

```

; ----- Define los records de byte con los equipos para la info del equipo

RECORD Equip printers:2, x:1, game:1, ports:3, y:1, drives:2, mode:2, ram:2, z:1, disk:1

; ---- Definiendo máscara para aislar campos individuales por bit

```
; -----
; máscara and    campo
; -----
```

```
maskPrinters = MASK printers
maskGame     = MASK game
maskPorts    = MASK ports
maskDrives   = MASK drives
maskMode     = MASK mode
maskDisk     = MASK disk
```

```
        dataseg
exCode db 0
welcome db cr, lf, 'Determinacion del equipo'
        db cr, lf, '2006 version', cr, lf, lf, EOS
strPrinters db 'Numero de impresoras .....', EOS
strGame      db 'Puerto de juegos I/O .....', EOS
strPorts     db 'Numero de puertos RS232 .....', EOS
strDrivers   db 'drives de disco (menos 1) ...', EOS
strMode      db 'modo inicial de video .....', EOS
strDisk      db 'tiene drive (1=yes) .....', EOS
string       db 40 dup (?); cadena de trabajo
        codeseg
;----- de strio.obj y binasc.obj
```

```
extrn BinToAscDec:proc, StrWrite:proc, NewLine:proc
```

Start:

```
mov ax, @data
mov ds, ax
mov es, ax
mov di, offset welcome ; direccion del mensaje de bienvenida
call StrWrite ; Despliega el mensaje
int 11h ; determinación del equipo por BIOS
mov bx, ax ; salvar la información en bx
mov di, offset strPrinters ; dirección de la etiqueta del equipo
mov dx, maskPrinters ; asigna máscara and
mov cl, printers ; asigna la cuenta del shift
call ShowInfo ; despliega información y la etiqueta
mov di, offset StrGame ; siguiente equipo
mov dx, maskGame
mov cl, game
call ShowInfo
mov di, offset strPorts ; siguiente equipo
mov dx, maskPorts
```



```

mov cl,ports
call ShowInfo
mov di, offset strDrivers ; siguiente equipo
mov dx,maskDrives
mov cl, drives
call ShowInfo
mov di, offset strMode ; siguiente equipo
mov dx,maskMode
mov cl, mode
call ShowInfo
mov di, offset strDisk ; siguiente equipo
mov dx, maskdisk
mov cl, disk
call ShowInfo
Exit:
    mov ah, 04ch
    mov al, [exCode]
    int 21h
; -----
; ShowInfo Despliega la etiqueta y el valor del equipo
; -----
; Entrada :
;     bx=dato del equipo de la int 11h
;     cl=campo de bit shifted
;     dx=campo de bit máscara AND
;     di=dirección de la cadena de etiqueta
; Salida:
; etiqueta y datos desplegados
; registros:
;     ax,cx
; -----

PROC ShowInfo
    mov ax, bx ; asigna el valor del equipo a ax
    and ax,dx ; aisla el campo del bit en ax
    shr ax,cl ; desplaza el campo más a la derecha en ax
    call StrWrite ; Despliega la etiqueta en di
    mov di, offset string ;dirección de la cadena de trabajo
    mov cx, 1; solicita al menos 1 dígito
    call BinToAscDec ; convierte ax a cadena ASCIIZ
    call StrWrite ; Despliega cadena
    call NewLine ; comienza una nueva línea
    ret
endp ShowInfo
end Start ;

```

Convertidor de base
PROGRAMA DE 32 bits

```

; set some options for the assembler

.386P
Locals
jumps
;includelib kernel32.lib ; no hace falta incluirlos con import
;includelib user32.lib
.Model Flat ,StdCall
mb_ok equ 0 ;mb_ok gets the value "0"
hWnd equ 0
lpText equ offset text ;set a pointer to the text
lpCaption equ offset caption ;set a pointer to the caption
; declaration of all used API-functions

extrn ExitProcess : PROC ;procedure to shut down a process

extrn MessageBoxA : PROC ;procedure to show a MessageBox

; here begins our Data

.Data

text db "Hola Mundo",13,10 ; first row of the text(with word-wrap)
      db "Haciendo mi primer programa con API win en asm",0
      ; second row, terminated with "0"

caption db "HOLITAS",0 ;Captionstring, 0-terminated

; and here we start with our code

.Code

Main:

; lets greet the world :)

push mb_ok ;PUSH value for uType
push lpCaption ;PUSH Pointer to Caption
push lpText ;PUSH Pointer to Text
push hWnd ;PUSH Masterhandle
call MessageBoxA ;CALL MessageBoxA
xor ax,ax ; limpia registro

push ax ;código de salida

```

```
call ExitProcess ;End (exit) program
```

```
End Main ;End of code, Main is the entrypoint
```

```
;-----END-----tut.asm
```

```
;===== PARA REALIZAR EL MAKE.BAT
```

```
; tasm32 -ml -m5 -q -zn ex32win.asm ---->>> lo importante aqui es hacelo casesensitive global por los extrn
```

```
; tlink32 -v -Tpe -c -x -aa ex32win,,,import32
```

Import32 es una libreria fácil de conseguir, si no es que ya se tiene en la PC.

Para saber cuáles son los parámetros del API bajar win32.hlp

Example 10-1. Interrupt 9 (Keyboard) Handler

KbdInt

```

push    ax                ; Save registers
push    ds                ;
mov     ax, cs            ; Make sure DS = CS
mov     ds, ax            ;
in      al, 60h           ; Get scan code
        |                ;
        |                ; Process event
        |                ;
in      al, 61h           ; Send acknowledgment without
or      al, 10000000b      ; modifying the other bits.
out     61h, al           ;
and     al, 01111111b      ;
out     61h, al           ;
mov     al, 20h           ; Send End-of-Interrupt signal
out     20h, al           ;
pop     ds                ; Restore registers
pop     ax                ;
iret                    ; End of handler

```

10.4 Internal Speaker

10.4.1 The Speaker Interface

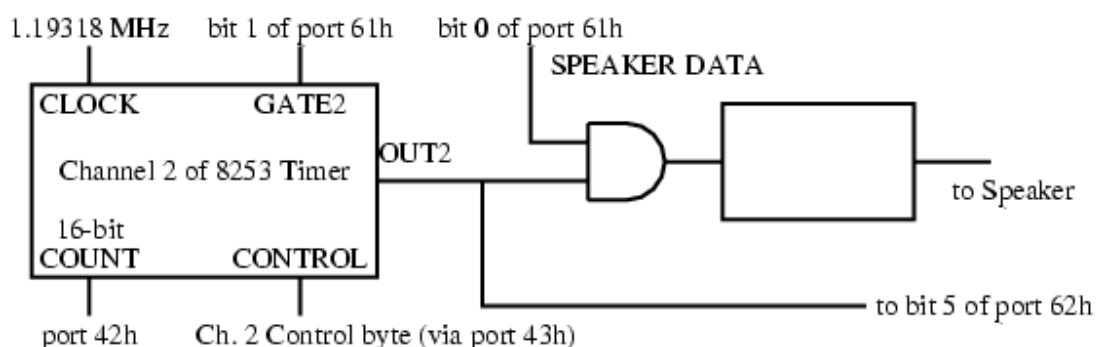
The PC has an internal speaker which is capable of generating beeps of different frequencies. You control the speaker by providing a frequency number which determines the pitch of the beep, then turning the speaker on for the duration of the beep.

The frequency number you provide is actually a counter value. The PC uses it to determine how long to wait between sending pulses to the speaker. A smaller frequency number will cause the pulses to be sent quicker, resulting in a higher pitch. The PC uses a base rate of 1,193,180 Hz (this frequency is generated by an oscillator chip). The frequency number tells the PC how many of these cycles to wait before sending another pulse. Thus, you can calculate the frequency number required to generate a specific frequency by the following formula:

The frequency number is a word value, so it can take values between 0 and 65,535 inclusive. This means you can generate any frequency between 18.21 Hz (frequency number = 65,535) and 1,193,180 Hz (frequency number = 1).

Figure 10-1 is a diagram of the hardware for driving the built-in speaker. OUT2 is the output of Channel 2 of the 8253-5 timer chip, GATE2 (= bit 1 of port 61h) is the enable/trigger control for the Channel 2 counter, and SPEAKER DATA (= bit 0 of port 61h) is a line that may be used independently to modulate the output waveform, e.g., to control the speaker volume.

Figure 10-1. Built-in speaker hardware



The count and load modes selected for Channel 2 during BIOS initialization are probably the best to use for tone production. In Mode 3, the counter output is a continuous symmetrical square wave as long as the GATE line of the channel is enabled; the other modes either produce outputs that are too asymmetrical or require retriggering for each count cycle.

The frequency count is loaded into the Channel 2 COUNT register at I/O port 42h. GATE2 (bit 1 of I/O port 61h) must be set to 1 to get an output on OUT2; the SPEAKER DATA line (bit 0 of I/O port 61h) must also be set to 1 to produce a tone. Note that the remaining bits of port 61h must not be changed since they control RAM enable, keyboard clock, etc. To silence the speaker, bits 1 or 0 of port 61h are set to 0 (without disturbing the remaining bits of port 61h).

10.4.2 Generating Sounds

You can communicate with the speaker controller using IN and OUT instructions. The following lists the steps in generating a beep:

1. Send the value 182 to port 43h. This sets up the speaker.
2. Send the frequency number to port 42h. Since this is an 8-bit port, you must use two OUT instructions to do this. Send the least significant byte first, then the most significant byte.
3. To start the beep, bits 1 and 0 of port 61h must be set to 1. Since the other bits of port 61h have other uses, they must not be modified. Therefore, you must use an IN instruction first to get the value from the port, then do an OR to set the two bits, then use an OUT instruction to send the new value to the port.
4. Pause for the duration of the beep.
5. Turn off the beep by resetting bits 1 and 0 of port 61h to 0. Remember that since the other bits of this port must not be modified, you must read the value, set just bits 1 and 0 to 0, then output the new value.

The following code fragment generates a beep with a frequency of 261.63 Hz (middle C on a piano keyboard) and a duration of approximately one second:

```

mov     al, 182           ; Prepare the speaker for the
out     43h, al           ; note.
mov     ax, 4560          ; Frequency number (in decimal)
                        ; for middle C.
out     42h, al           ; Output low byte.
mov     al, ah            ; Output high byte.
out     42h, al
in      al, 61h           ; Turn on note (get value from
                        ; port 61h).
or      al, 00000011b     ; Set bits 1 and 0.
out     61h, al           ; Send new value.
mov     bx, 25            ; Pause for duration of note.
.pause1:
mov     cx, 65535
.pause2:
dec     cx
jne     .pause2
dec     bx
jne     .pause1
in      al, 61h           ; Turn off note (get value from
                        ; port 61h).
and     al, 11111100b     ; Reset bits 1 and 0.
out     61h, al           ; Send new value.

```

Another way to control the length of beeps is to use the timer interrupt. This gives you better control over the duration of the note and it also allows your program to perform other tasks while the note is playing.

10.4.3 Frequency Table

The following table lists frequencies and frequency numbers for the three octaves around middle C on a piano keyboard.

Note	Frequency	Frequency #
C	130.81	9121
C#	138.59	8609
D	146.83	8126
D#	155.56	7670
E	164.81	7239
F	174.61	6833
F#	185.00	6449
G	196.00	6087
G#	207.65	5746
A	220.00	5423
A#	233.08	5119
B	246.94	4831
Middle C	261.63	4560
C#	277.18	4304
D	293.66	4063
D#	311.13	3834
E	329.63	3619
F	349.23	3416
F#	369.99	3224
G	391.00	3043
G#	415.30	2873
A	440.00	2711
A#	466.16	2559
B	493.88	2415
C	523.25	2280
C#	554.37	2152
D	587.33	2031
D#	622.25	1917
E	659.26	1809
F	698.46	1715
F#	739.99	1612
G	783.99	1521

Note	Frequency	Frequency #
G#	830.61	1436
A	880.00	1355
A#	923.33	1292
B	987.77	1207
C	1046.50	1140