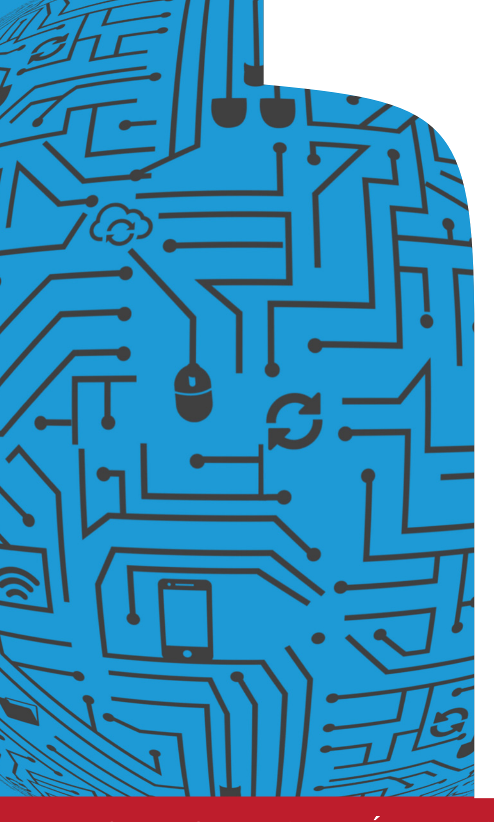
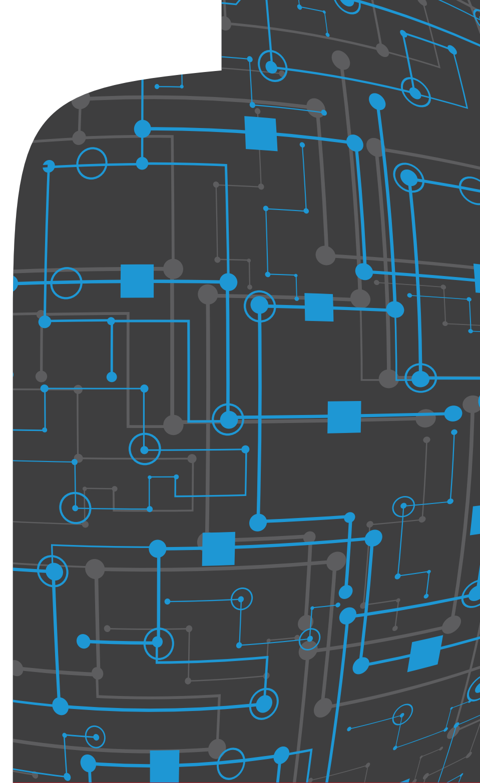




INSTITUTO DE **INGENIERÍA Y AGRONOMÍA**

INGENIERÍA EN INFORMÁTICA

INTRODUCCIÓN A LA ORGANIZACIÓN Y ARQUITECTURA DE LA COMPUTADORA

Ing. Jorge R. Osio
Dr. Ing. D. Martin Morales
Autores

EDITORIAL | UNAJ

Universidad Nacional
ARTURO JAURETCHE

Morales, Daniel Martin
Ingeniería en Informática : introducción a la organización y arquitectura de la computadora / Daniel Martin Morales ; Jorge Rafael Osio. - 1a ed . - Florencio Varela : Universidad Nacional Arturo Jauretche, 2018.
Libro digital, PDF

Archivo Digital: descarga y online
ISBN 978-987-3679-28-5

1. Ingeniería Informática. 2. Computación. 3. Iniciación Informática. I. Osio, Jorge Rafael II. Título
CDD 629.89

Universidad Nacional Arturo Jauretche
Rector: **Lic. Ernesto Fernando Villanueva**

Director del Instituto de Ingeniería y Agronomía: Ing. Miguel Binstock
Vicedirector: Dr. Ing. Martín Morales

Programa Tecnologías de la información y la comunicación (TIC)
en aplicaciones de interés social.
Director del programa: Dr. Ing. Martín Morales

Coordinación editorial: Gabriela Ruiz
Diseño de tapa y maquetación: Editorial UNAJ
Correctora: Érica Andrea Marino

© 2018, UNAJ
Av. Calchaquí 6200 (CP1888)
Florencio Varela Buenos Aires, Argentina
Tel: +54 11 4275-6100
editorial@unaj.edu.ar
www.unaj.edu.ar

Queda hecho el depósito que marca la Ley 11.723

Universidad Nacional Arturo Jauretche

INGENIERÍA EN INFORMÁTICA

Introducción a la organización y arquitectura de la computadora

Ing. Jorge R. Osio
Dr. Ing. D. Martin Morales
Autores

Índice

Introducción 11
1. Sistemas de numeración y código 13
1.1. Sistema Binario 13
1.1.1. Números signados 14
1.1.1. Módulo y signo 15
1.1.1.2. Complemento a 1 16
1.1.1.3. Complemento a 2 17
1.2. Sistema Hexadecimal 19
1.4. Código de operación 22
1.5. Mnemónicos y ensambladores 22
1.6. Octal 22
1.7. Binario codificado en decimal 24

2. Sistemas digitales 27
2.1. Lógica Digital 27
2.2. Circuitos digitales 27
2.3. El transistor 27
2.4. Operadores lógicos 28
2.5. Algebra de Boole 29
2.5.1. Leyes de Boole para simplificar ecuaciones 30
2.5.2. Simplificación de circuitos 31
2.5.3. Expresiones booleanas desde tablas de verdad 32
2.6. Clasificación de los circuitos integrados 33
2.7. Lógica combinacional 34
2.7.1. Comparadores 34
2.7.2. Decodificadores 35
2.7.3. Codificadores 36
2.7.4. Multiplexores 36
2.7.5. Demultiplexores 37
2.8. Implementación de partes de la computadora mediante compuertas 38
2.8.1. Sumador completo 38
2.8.2. Unidad Aritmético Lógica 39
2.8.3. Latch 41
2.8.4. Flip-flop 43

3. Microprocesadores y microcontroladores 45
3.1. Características generales de un microprocesador 45
3.1.1. Descripción de los buses de comunicación 46
3.1.1.1. Bus de Datos 46
3.1.1.2. Bus de direcciones 46
3.1.1.3. Bus de control 47
3.1.2. Descripción del bloque de memoria 47
3.1.2.1. Memoria de datos 47
3.1.2.2. Memoria de instrucciones 47
3.1.3. Descripción de la CPU 47
3.1.3.1. Registros internos 48
3.1.3.2. Unidad de control 49
3.1.3.3. Unidad de procesos 50
3.1.4. Descripción de la unidad de entrada/salida 51
3.1.5. Ejemplo de ejecución de una instrucción en un microprocesador de 8 bits 52
3.2. Características generales de un microcontrolador 53

4. Eficiencia y rendimiento en sistemas de procesamiento 55
4.1. Aclaraciones de Amdahl 55
4.2. Generalización de la ley de Amdahl 56

5. Descripción del procesador CPU08 59
5.1. Arquitectura de ejecución del CPU 08 59
5.2. El “Prefetch” en el CPU08 60
5.3. Unidad de control 60
5.4. Unidad de procesos 60
5.5. Descripción de registros de la CPU 61
5.5.1. Acumulador (A) 61
5.5.2. Registro índice (H : X) 62
5.5.3. Puntero de pila (SP) 62
5.5.4. Contador de programa (PC) 63
5.5.5. Registro de código de condición (CCR) 63

6. Modos de direccionamiento 65
6.1. Modo de direccionamiento inmediato 65
6.2. Modo de direccionamiento inherente 66
6.3. Modo de direccionamiento directo 67
6.4. Modo de direccionamiento extendido 68

6.5. Modo de direccionamiento indexado ... 70
6.5.1. Direccionamiento indexado "sin desplazamiento (Offset)" ... 70
6.5.2. Direccionamiento indexado con "desplazamiento (offset) de 8 bits" ... 71
6.6. Modo de direccionamiento relativo ... 72

7. Memoria ... 75
7.1. Introducción ... 75
7.2. Ubicaciones de memoria no implementada ... 75
7.3. Ubicaciones de memoria reservada ... 75
7.4. Sección Entrada/Salida (I/O) ... 75
7.5. Memoria de acceso aleatorio (RAM) ... 76
7.6. Memoria FLASH (FLASH) ... 77

8. Programación a bajo nivel (Assembler) ... 79
8.1. Introducción ... 79
8.2. El lenguaje de bajo nivel ... 79
8.3. Software de programación y simulación ... 79
8.4. Planificación y programación de una aplicación en Assembler ... 83
8.4.1. Diagrama de flujo ... 84
8.4.2. Programación de la aplicación ... 85
8.4.3. Descripción del código ... 85

Apéndice A. Descripción de arquitecturas clásicas ... 89
A.1. Arquitectura Harvard ... 89
A.2. Arquitectura Von Neumann ... 90
A.3. Arquitectura Harvard vs. arquitectura Von Neumann ... 91
A.4. CISC VS RISC ... 91

Apéndice B. Características del set de instrucciones ... 93
B.1. Movimiento de datos ... 94
B.2. Aritméticas ... 95
B. 3. Lógicas ... 98
B.4. Manipulación de datos ... 98
B.5. Manipulación de bits ... 98
B.6. Control de flujo de programa ... 99
B.7. Operaciones en BCD ... 99
B.8. Especiales ... 100
Referencias ... 101

Introducción

Para comprender el funcionamiento de un sistema de cómputo es necesario conocer previamente el sistema de numeración que utiliza y las operaciones que se pueden realizar.

Por otro lado, para entender la organización de dicho sistema se debe estudiar la base de la lógica programada y de los circuitos digitales. Teniendo claros estos conceptos se procederá al estudio del funcionamiento del sistema de procesamiento.

Este libro se orienta a describir el funcionamiento de la arquitectura básica de un microprocesador, donde se detallan cada una de las partes en base a la Arquitectura de Von Newmann.

Es importante conocer las características de bajo nivel de un procesador (programación en Assembler) para entender mejor el funcionamiento interno, es por eso por lo que se estudia la programación de bajo nivel de un procesador de 8 bits como el CPU08, el cual posee una cantidad reducida de registros y un set de instrucciones que facilita la programación.

Este texto está especialmente orientado a los alumnos de la carrera de Ingeniería informática, donde esta temática se dicta en la Asignatura de Organización y arquitectura de computadoras y es de difícil comprensión para estudiantes con un perfil orientado a la programación en alto nivel.

1 Sistemas de numeración y código

Las CPUs trabajan adecuadamente con información en un formato diferente al que la gente está habituada a manejar a diario. Típicamente se trabaja en el sistema de numeración de base 10 (decimal, con niveles de 0 a 9). Las computadoras digitales binarias trabajan con el sistema de numeración en base 2 (binario, 2 niveles), puesto que este permite representar cualquier información mediante un conjunto de dígitos, que solo serán "ceros" (0) o "unos" (1).

Un uno o un cero puede representar la presencia o ausencia de un nivel lógico de tensión sobre una línea de señal, o bien el estado de encendido o apagado de una simple llave.

En este apartado se explican los sistemas de numeración más comúnmente utilizados por las computadoras, es decir, binario, hexadecimal, octal y binario codificado en decimal (BCD) [1] y [2].

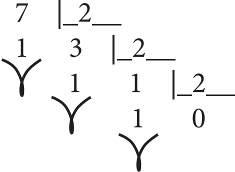
Las CPUs además usan códigos especiales para representar información del alfabeto o sus instrucciones. La comprensión de estos códigos ayudará a entender cómo una computadora se las ingenia para entender cadenas de dígitos que solo pueden ser unos o ceros.

1.1. Sistema Binario

Para un número decimal (base 10) el peso (valor) de un dígito es diez veces mayor que el que se encuentra a su derecha. El dígito del extremo derecho de un número decimal entero es el de las unidades, el que está a su izquierda es el de las decenas, el que le sigue es el de las centenas, y así sucesivamente.

Para un número binario (base 2), el peso de un dígito es dos veces mayor que el que se encuentra a su derecha. El dígito del extremo derecho de un número binario entero es el de la unidad, el que está a su izquierda es el de los duplos, el que le sigue es el de los cuádruplos, el que le sigue es el de los óctuplos, y así sucesivamente. Para algunas computadoras es habitual trabajar con números de 8, 16, 32 o 64 dígitos binarios.

Para pasar un número de decimal a binario se deben realizar sucesivas divisiones por la base 2 del sistema binario, luego el último resto será el bit más significativo del número binario y el primero, el bit menos significativo. Por ejemplo si queremos pasar el número 7 a binarios tendremos:



Por lo tanto el número 7_{10} se representa en binario como el 111_2.

Por otro lado, si deseamos representar el número expresado en binario en el sistema decimal, deberemos realizar la suma sucesiva de cada uno de los dígitos multiplicando por la base elevada a una potencia que representa la posición del bit en el número. Por ejemplo, si pasamos el 111_2 a decimal tendremos:

$111_2 = 1 . 2^2+1 . 2^1+1 \ 2^0 = 7_{10}$

Generalizando, el número binario $b_3b_2b_1b_0 = b_3 . 2^3 + b_2 . 2^2 + b_1 . 2^1 + b_0 . 2^0$

1.1.1. Números signados

En el Sistema Decimal un número con signo se representa mediante un símbolo delante del dígito más significativo, este puede ser “+” o “-”.

En el Sistema binario, en cambio, debido a que la intención de representar números binarios usando solo dos símbolos tiene el objetivo de facilitar la realización de cálculo y operaciones en los sistemas de cómputo, se decidió representar el signo mediante un bit, en donde el valor “1” representa el signo negativo y el valor “0” representa el número positivo. A partir de esta decisión aparecen tres métodos de representación de números con signo en el Sistema Binario.

❒ Módulo y signo
❒ Complemento a 1
❒ Complemento a 2

En los tres métodos mencionados la representación de los números positivos se realiza de manera similar, pero los números negativos se representan de diferente manera, según el el método seleccionado.

1.1.1.1. Módulo y signo

Por convención un número positivo se distingue por tener su bit más significativo (MSB) en 0 y un número negativo por tenerlo en 1. Esto permite identificar el signo en los tres métodos de representación con signo, lo que cambia en cada sistema es el método que se aplica para obtener el número negativo.

Dado un número de *n* bits, el bit más significativos se reserva para el signo y los *n*- 1 bits restantes se utilizan para representar el módulo. Entonces, al utilizar un bit para el signo se puede decir que el máximo número que se puede representar con *n* bits en módulo está dado por $2^{(n-1)}-1$.

Por lo tanto, el máximo número positivo que se puede representar con *n* bits es $+2^{(n-1)}-1$ y el mínimo número negativo a representar con *n* bits será: $-2^{(n-1)}-1$

Ejemplo: Con 3 bits un número sin signo puede ir de 0 a 7, pero un número con signo de 3 bits puede representar desde el -3 al +3. En este sistema de representación un número positivo y el mismo número negativo se diferencian simplemente por el bit más significativo, ya que el +3 y -3 se representan como 011 y 111, respectivamente.

En la figura siguiente se observan todos los números signados que se pueden representar con 3 bits, destacando la diferencia del bit más significativo entre dos números de igual módulo pero distinto signo.

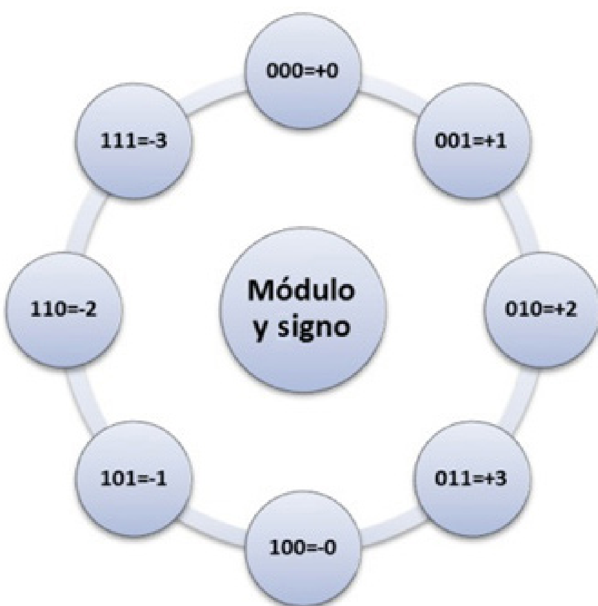


Figura 1. Gráfico de los números que se pueden representar con 3 bits en módulo y signo.

Este método de representación tiene la desventaja de tener una doble representación del cero (+-0) y tiene cierta dificultad en la realización de operaciones de suma y resta, dado que para sumar o restar dos números previamente deberemos determinar si sus signos son iguales o distintos.

1.1.1.2. Complemento a 1

En este sistema de representación los números positivos se representan igual que en el de módulo y signo, pero los números negativos se representan en base a la siguiente definición.

Dado un número N se llama complemento a 1 (N*1) de N a aquel número de igual módulo que N pero de signo opuesto y se obtiene como se indica a continuación:

$N^*1 = 2^n - 1 - N$

Si tenemos el número $+14_{10} = 001110_2$ (representado con 6 bits) y queremos obtener el -14 en complemento a 1, hacemos el siguiente cálculo:

$2^n = 2^6 = 1000000$
$2^n - 1 = 2^6 - 1 = 111111$

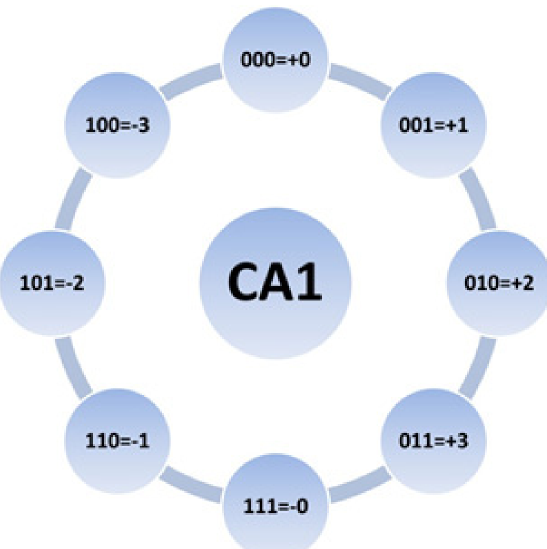


Figura 2. Representación de los distintos números de 3 bits en complemento a 1.

Entonces en número N*1 = 2^6 - 1 - 14 = 111111 – 001110 = 110001

En el caso de tener un número negativo en decimal como dato y de requerir pasarlo a binario en complemento a 1, primero deberemos pasar el 12 a binario y luego aplicarle el complemento a 1.

En la figura 2 se muestran todos los valores que se pueden representar en complemento a 1 con 3 bits. En ella se observa la doble representación del cero como 000 (+0) y 111 (-0). Por otro lado, en la figura se puede notar que, dado un número positivo o negativo, su complemento tiene los *n* bits invertidos, de esta manera si el 3 se representa como 011, el -3 se representará como 100.

Como en el método de representación anterior, el máximo número positivo que se puede representar con *n* bits está dado por $+2^{(n-1)} - 1$ y el mínimo número negativo a representar con *n* bits será $-2^{(n-1)} - 1$.

La principal ventaja de este método de representación es que permite realizar operaciones de suma y resta de manera fácil, a diferencia del de módulo y signo. Adicionalmente, se debe resaltar que el pasaje de un número positivo a uno negativo se puede hacer fácilmente en un sistema de cómputo, por medio de la ALU (Unidad Aritmético Lógica). Por otro lado, sigue teniendo el problema de la doble representación del cero.

1.1.1.3. Complemento a 2

Aquí nuevamente los números positivos se representan de la misma manera que en los métodos anteriores, pero los números negativos siguen la siguiente convención:

Dado un número N, su complemento a 2 (N*2) será:

N*2 = 2^n – N

A simple vista se puede observar que la diferencia entre N*1 y N*2 es el 1, que en este último caso no está restando a N. Al omitir esta resta se obtienen los siguientes beneficios:

- ❒ Hay una única representación del cero.
- ❒ Esto permite representar un número negativo más en lugar del -0.

En la figura 3 se observan todos los números que se pueden representar en complemento a 2 con 3 bits, en donde se observa que se puede representar el -4 en lugar del -0.

Finalmente los rangos de representación quedarán con el máximo número positivo a representar con *n* bits como $+2^{(n-1)}-1$ y el mínimo número negativo dado por $-2^{(n-1)}$.

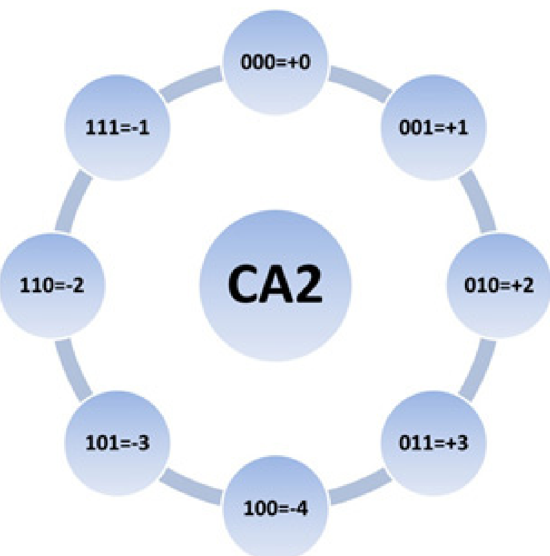


Figura 3. Representación de los posibles números de tres bits en complemento a 2.

Como ejemplo supongamos que tenemos el $+23_{10}$ y queremos hallar el -23 en CA2 con 8 bits (esto es $n = 8$), entonces:

$N =+ 23_{10} = 00010111_2$
$2^n = 2^8 = 100000000$
$N^*2 = 2^n - N = 100000000 - 00010111 = 11101001$

Por último, sobre la base de la relación entre complemento a 1 y complemento a 2, se puede establecer la siguiente regla mnemotécnica para pasar un número positivo a negativo en complemento a 2:

1. Invertir todos los bits del número binario, esto es N*1.
2. Sumar 1 a lo obtenido en el paso anterior, esto es N*2 = N*1 +1

1.2. Sistema Hexadecimal

El sistema de numeración de **base 16 (Hexadecimal)** resulta ser una práctica solución de compromiso. Un dígito hexadecimal se puede representar exactamente a cuatro dígitos binarios, de este modo un número binario de 8 dígitos se puede expresar mediante dos dígitos hexadecimales. La sencilla correspondencia que existe entre las representaciones de un dígito hexadecimal y la de cuatro dígitos binarios, permite realizar mentalmente la conversión entre ambos. Para un número **hexadecimal (base 16),** el peso de un dígito es de dieciséis veces más, respecto del que se encuentra a su derecha. Por ejemplo, FA en hexadecimal se puede expresar en decimal como $F \times 16^1 + A \times 16^0$. El dígito del extremo derecho de un número hexadecimal entero, es de las unidades, el que está a su izquierda es el de los décimos séxtuplos, y así sucesivamente.

La conversión de decimal a hexadecimal se realiza de la misma forma que la conversión de decimal a binario, solo que se debe dividir sucesivamente por la base 16 y el número hexadecimal resultante saldrá de los sucesivos restos, tomando el último como dígito más significativo.

La tabla 1 muestra la relación existente, en la representación de valores, en los sistemas Decimal, Binario y Hexadecimal. Estos tres diferentes sistemas de numeración resultan ser tres modos diferentes de representar físicamente las mismas cantidades. Se utilizan las letras de la A a la F para representar los valores hexadecimales correspondientes, que van del 10 al 15 en decimal, ya que cada dígito hexadecimal puede representar 16 cantidades distintas. Debido a que el sistema de representación decimal solo incluye diez símbolos (del 0 al 9), se debe recurrir a algún otro símbolo de un solo dígito, para de esta manera, representar los valores hexadecimales que van del 10 al 15.

Para distinguir un número hexadecimal de uno decimal en la programación de bajo nivel, se antepone el símbolo "$" a cada cantidad hexadecimal y el símbolo "!" a cada cantidad decimal. Por ejemplo, !64 es el decimal "sesenta y cuatro"; entonces $64 es hexadecimal "seis-cuatro", que es equivalente al decimal 100.

Tabla 1. Equivalentes entre los sistemas Decimal, Binario y Hexadecimal

Base 10 Decimal	Base 2 Binario	Base 16 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
16	0001 0000	10
17	0001 0001	11
100	0110 0100	64
255	1111 1111	FF
1024	0100 0000 0000	400
65535	1111 1111 1111 1111	FFFF

El Hexadecimal es el modo adecuado tanto para expresar como para discutir acerca de información numérica procesada por una computadora, ya que resulta fácil realizar la conversión entre un dígito hexadecimal y sus 4 bits equivalentes. La notación hexadecimal es mucho más compacta y fácil de interpretar que la binaria mientras se mantienen las connotaciones binarias.

1.3. Código ASCII

Las CPUs deben manejar otros tipos de información además de los números. Tanto los textos (caracteres alfanuméricos) como las instrucciones deben codificarse de tal modo que la CPU interprete esta información. El código más común para la información tipo texto es el ***American Standard Code for Information Interchange*** **(ASCII) [1].**

El código ASCII es una correlación ampliamente aceptada entre caracteres alfanuméricos y valores binarios específicos. En este código, el número $41 corresponde a una letra A mayúscula, el $20 al carácter espacio, etc. El código ASCII traduce un carácter a un código binario de 7 bits, aunque en la práctica la mayoría de las veces la información es transportada en caracteres de 8 bits, con el bit más significativo en cero. Este estándar hace posibles las comunicaciones entre equipos producidos por diversos fabricantes, puesto que todas las máquinas utilizan el mismo código. La Tabla 2 muestra la relación existente entre los caracteres ASCII y los valores hexadecimales.

Tabla. 2. Conversión de ASCII a Hexadecimal

ASCII	Hex	Símbolo	ASCII	Hex	Símbolo	ASCII	Hex	Símbolo	ASCII	Hex	Símbolo
0	0	NUL	32	20	(espacio)	64	40	@	96	60	‘
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	“	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	‘	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	•

1.4. Código de operación

Las CPUs utilizan otro código para realizar acciones y realizar operaciones sobre el sistema. Este código se denomina *código de operación* (opcode) [2]**.** Cada código de operación instruye a la CPU en la ejecución de una muy específica secuencia de etapas que debe seguirse para cumplir con la operación que representa dicho opcode.

Las computadoras de distintos fabricantes usan diferentes repertorios de códigos de operación, previstos en la unidad de control de la CPU.

El **repertorio (set) de instrucciones** para una CPU es el conjunto de instrucciones que esta es capaz de ejecutar. Los códigos de operación son una representación del set de instrucciones y los mnemónicos son otra. Aun cuando difieren de una CPU a otra, todas las CPUs digitales binarias realizan el mismo tipo de tareas básicas de modo similar. El set de instrucciones del procesador HC08, que se describe en el capítulo 8, incluye 288 opcodes distintos.

1.5. Mnemónicos y ensambladores

Un opcode tal como $4C es interpretado por la CPU, pero no es fácilmente manejable por una persona. Para resolver este problema se usa un sistema de mnemónicos de instrucción equivalentes. El opcode $4C corresponde al mnemónico **INCA**, cuya operación consiste en “incrementar el registro acumulador en uno”. La relación entre el mnemónico de cada instrucción y el opcode que la representa es rara vez utilizada por el programador, pues el proceso de traducción lo realiza automáticamente un programa de computadora específico denominado **ensamblador** o **compilador.** Este programa es el que convierte los mnemónicos de las instrucciones de un programa en una lista de **códigos de máquina** (códigos de operación e información adicional), para que puedan ser utilizados por la CPU.

Un ingeniero desarrolla un grupo de instrucciones para una CPU con la forma de mnemónicos y luego utiliza un ensamblador para compilar y transformar estas instrucciones a los opcodes que la CPU pueda entender.

1.6. Octal

Antes de comenzar la discusión sobre los sistemas de numeración y códigos, se describirán dos códigos más, muy utilizados en su momento. La notación **octal (base 8)** que fue usada para trabajar con algunas computadoras primitivas, pero

rara vez es utilizada en la actualidad [1]. Esta usa los números que van del 0 al 7 para representar un conjunto de tres dígitos binarios de un modo análogo a la hexadecimal, en el que se recurre a un conjunto de cuatro dígitos binarios. El sistema octal tiene la ventaja de no necesitar símbolos no numéricos (como los símbolos hexadecimales ya vistos, que van de la "A" a la "F").

Tabla 3. Conversión de Decimal, Hexadecimal y Binario a Octal.

Dec.	**Hex.**	**Oct.**	**Bin.**	**Dec.**	**Hex.**	**Oct.**	**Bin.**
0	0	000	00000000	32	20	040	00100000
1	1	001	00000001	33	21	041	00100001
2	2	002	00000010	34	22	042	00100010
3	3	003	00000011	35	23	043	00100011
4	4	004	00000100	36	24	044	00100100
5	5	005	00000101	37	25	045	00100101
6	6	006	00000110	38	26	046	00100110
7	7	007	00000111	39	27	047	00100111
8	8	010	00001000	40	28	050	00101000
9	9	011	00001001	41	29	051	00101001
10	A	012	00001010	42	2A	052	00101010
11	B	013	00001011	43	2B	053	00101011
12	C	014	00001100	44	2C	054	00101100
13	D	015	00001101	45	2D	055	00101101
14	E	016	00001110	46	2E	056	00101110
15	F	017	00001111	47	2F	057	00101111
16	10	020	00010000	48	30	060	00110000
17	11	021	00010001	49	31	061	00110001
18	12	022	00010010	50	32	062	00110010
19	13	023	00010011	51	33	063	00110011
20	14	024	00010100	52	34	064	00110100
21	15	025	00010101	53	35	065	00110101
22	16	026	00010110	54	36	066	00110110
23	17	027	00010111	55	37	067	00110111
24	18	030	00011000	56	38	070	00111000
25	19	031	00011001	57	39	071	00111001
26	10	032	00011010	58	3A	072	00111010
27	1B	033	00011011	59	3B	073	00111011
28	1C	034	00011100	60	3C	074	00111100
29	1D	035	00011101	61	3D	075	00111101
30	1E	036	00011110	62	3E	076	00111110
31	1F	037	00011111	63	3F	077	00111111

Cambiar la notación hexadecimal usada hoy en día por la octal acarrea dos desventajas. La primera es que las CPUs utilizan "words" (palabras) de 4, 8, 16 o 32 bits, estas "palabras" no resultan fácilmente fraccionables en grupos de tres bits (algunas CPUs muy antiguas usaban palabras de 12 bits, divisibles en cuatro grupos de tres bits). La segunda es el hecho de carecer de compatibilidad con la hexadecimal, en cuanto a cantidad de bits necesarios para la representación. Por ejemplo, el valor ASCII de la letra "A" mayúscula es **10000001 (base 2)**, **41(base 16)** en hexadecimal y **101 (base 8)** en octal. Cuando se menciona el valor ASCII para la "A", es más fácil decir "**cuatro - uno"** que "**uno - cero - uno".**

La tabla 3 presenta las correlaciones entre octal y binario. La columna "binario directo" muestra dígito por dígito el pasaje de los dígitos octales a grupos de 3 bits. Cada grupo de 4 bits se convierte directamente en un dígito hexadecimal.

Cuando mentalmente se convierten valores octales a valores binarios de un byte, el valor octal resultante se representa mediante 3 dígitos octales. Cada dígito octal representa a su vez 3 bits con lo que resulta un bit extra (3 dígitos x 3 bits = 9 bits).

Típicamente se opera de izquierda a derecha, lo que facilita olvidarse del tratamiento que debe recibir el bit extra del extremo izquierdo (noveno) de un octal. Cuando se convierte de hexadecimal a binario, resulta más sencillo, porque cada dígito hexadecimal se transforma exactamente en cuatro bits. Dos dígitos hexadecimales coinciden exactamente con los ocho bits de un byte.

1.7. Binario codificado en decimal

El sistema Binario codificado en decimal (**BCD**) es una notación híbrida usada para expresar valores decimales en forma binaria. Un BCD utiliza cuatro bits para representar cada dígito decimal.

De esta manera, cuatro dígitos binarios pueden expresar dieciséis diferentes cantidades físicas, y existen seis combinaciones consideradas no válidas (específicamente, los valores hexadecimales de la "A" a la "F").

Cuando la CPU realiza una operación de suma BCD, efectúa una suma binaria y luego hace un ajuste que genera un resultado BCD. Como un simple ejemplo, se presenta la siguiente suma BCD.

9 + 1 = 10; en decimal.
La computadora hace la siguiente suma en BCD:
0000 1001 + 0000 0001 = 0000 1010; en Binario.

Pero 1010 en Binario es equivalente a “A en Hexadecimal” que es un código BCD no válido. Cuando la CPU termina el cálculo, realiza un chequeo para ver si el resultado es un código BCD válido. Si hubo un “acarreo” (un desborde) de un dígito BCD a otro o si hubiese algún código no válido, se desencadenaría una secuencia de etapas para corregir el resultado y llevarlo al formato BCD apropiado.

El número 0000 1010 (en Binario) es corregido y se transforma en 0001 0000 (en Binario) o 10 (en BCD). Esta corrección consiste en analizar el resultado y determinar si en algún dígito BCD (4 bits) se superó el valor 1001, de ser así, se deberá sumar 6 o 0110 en binario para lograr el desborde y poder representar un digito decimal con 4 bits de manera correcta.

En la mayoría de los casos es ineficiente utilizar la notación BCD para los cálculos de la CPU. Es mejor convertir la información de Decimal a Binario en el momento de su ingreso, realizar todos los cálculos en Binario y convertirlos nuevamente a BCD o Decimal, solo si es necesario presentarlos en pantalla.

No todos los microcontroladores son capaces de realizar cálculos en BCD, ya que se debe tener la indicación del acarreo dígito a dígito que no está presente en todas las CPU (el acarreo entre el bit 3 y el bit 4 de un número binario se denomina “Indicador de semiacarreo”).

Forzar a una computadora a comportarse como nosotros resulta menos eficiente que permitirle trabajar en su sistema de numeración natural.

Sistemas digitales

2.1. Lógica Digital

La Computadora necesita almacenar datos e instrucciones en memoria. El sistema Binario presenta solo dos estados que se corresponden con el verdadero y falso de lógica.

Motivación de la lógica digital:

• Menos posibilidad de falla.
• Fácil de identificar con solo dos estados posibles.

2.2. Circuitos digitales

• Se utilizan sistemas y circuitos donde solo existen dos estados posibles. Los estados son representados mediante dos niveles diferentes de tensión (continua). Estos valores son Alto (H) que representa un 1 lógico y Bajo (L) que representa un cero lógico.
• Los dígitos (bits) que se utilizan son los del sistema binario que hemos estudiado hasta el momento. La palabra *bits* proviene de la abreviación de las palabras inglesas: ***B****inary dig****IT***.
• Los valores que se utilizan para representar los 1 y 0 se denominan "Niveles lógicos".
• La información que manejan los sistemas digitales se presenta en forma de señales que representan secuencias de bits, señal alta (tensión de 5 volt) un "uno binario", señal baja (0 volt) un "cero binario".

2.3. El transistor

El transistor es un dispositivo electrónico que tiene tres terminales y es usado como amplificador e interruptor, entre otras aplicaciones.

El funcionamiento se basa en que la tensión aplicada a un terminal *b* llamado "base", que controla el comportamiento de la tensión en los dos terminales restantes *c* y *e*, "colector" y "emisor", respectivamente.

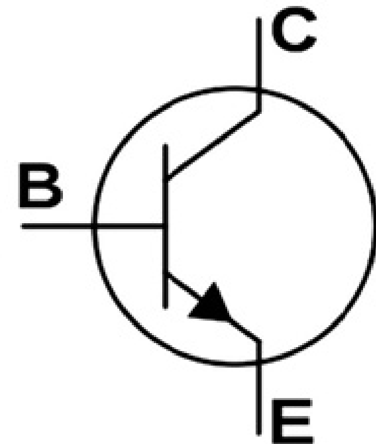


Figura 4. Símbolo del transistor

Funcionamiento del transistor bipolar (BJT):

- Al colocar un nivel lógico 1 en la entrada *b* (base), se generará un canal de conducción entre los terminales *c* y *e* que hará que la tensión sobre la salida C tenga un nivel lógico cero. En otras palabras, cuando hay un 1 en la entrada la salida se pone a cero lógico.
- Un cero lógico en la entrada b, no generará el canal de conducción entre colector y emisor, y de esta manera la salida C quedará a un nivel lógico alto fijado por la tensión de alimentación VCC que se conecta al colector a través de una resistencia. En otras palabras, cuando hay un cero lógico en la entrada *b*, la salida se pone a un nivel alto (1 lógico).
- Como conclusión se podría decir que el transistor actúa como un negador. Un 1 en la entrada se transforma en cero a la salida y viceversa.

2.4. Operadores lógicos

Los operadores principales son tres:

NOT, OR, AND

- La compuerta NOT niega la entrada. Si la entrada es cero la salida valdrá 1.
- La compuerta OR devuelve un cero solo cuando todas las entradas son cero. En otras palabras, cuando una de las entradas vale 1 la salida vale 1.
- La compuerta AND devuelve un 1 solo cuando todas sus entradas valen 1, para el resto de los casos la salida vale cero.

Operadores derivados:

NAND, NOR, XOR

- La NAND se comporta de forma opuesta a la AND. Esto significa que la salida valdrá cero cuando todas las entradas sean 1 y en el resto de los casos la salida valdrá 1.
- La NOR de manera opuesta a la OR. La salida valdrá cero cuando alguna de sus entradas valga 1. Cuando todas las entradas valgan cero la salida valdrá 1.
- En el caso de la XOR. La salida valdrá 1 cuando las entradas sean distintas y valdrá cero cuando las entradas sean iguales.

2.5. Algebra de Boole

Son las matemáticas de los circuitos lógicos. Aquí se formula un conjunto de propiedades para realizar operaciones lógicas.

Las operaciones se aplican sobre variables lógicas, entonces la variable A negada se representa como $\overline{A}$. Una AND entre A y B se representa como el producto booleano entre las variables A y B. Esto es X = A . B. Por su parte, la OR se representa mediante la suma booleana, esto es A + B.

Antes de avanzar con las propiedades de Boole, es necesario definir algunos conceptos importantes:

- **Variable:** es un símbolo (letra mayúscula) que se utiliza para representar magnitudes lógicas, una variable puede tomas valores de 1 o 0.
- **El complemento:** es el inverso de una variable y se indica mediante una barra encima de la variable.
- **Literal:** hace referencia a una variable o a su complemento, es una definición general de las dos anteriores.

Operaciones Booleanas:

Suma Booleana: Como se describió anteriormente, una suma booleana equivale a una compuerta OR. Las reglas de una suma booleana son:

1 + 1 = 1
1 + 0 = 1
0 + 1 = 1
0 + 0 = 0

Multiplicación Booleana: La multiplicación booleana equivale a la compuerta AND y sus reglas son:

0 . 0 = 0
1 . 0 = 0
0 . 1 = 0
1 . 1 = 1

2.5.1. Leyes de Boole para simplificar ecuaciones

Las leyes de Boole son las doce propiedades que se describen en la Tabla 4. Las seis primeras corresponden a la operación OR y las seis restantes corresponden a la operación AND.

- La ley de Identidad para la OR dice que una suma entre una variable A y 0 da la variable A. Para la AND dice que un producto entre una variable A y 1 da la misma variable.
- La ley Nula para la OR dice que una suma entre 1 y A da 1. Para la AND dice que un producto entre cero y A da cero.
- La ley de Idempotencia dice que una AND entre una variable A y la misma variable A da A. Para la OR, Una OR entre la variable A y A da A.
- La ley Inversa dice que A por A negado da 0. Una OR entre A y A negado da 1.
- La ley Conmutativa es igual que para operaciones aritméticas. Una AND entre *a* y *b* da lo mismo que una AND entre *b* y *a*. Lo mismo pasa con la conmutatividad en la operación OR.
- La ley Asociativa también es igual que en las operaciones aritméticas. Es lo mismo hacer la AND entre *a* y *b* y luego al resultado aplicarle la AND con *c*, que hacer la AND entre *b* y *c*, y al resultado aplicarle la AND con *a*. Se cumple lo mismo para la OR.
- La ley Distributiva se aplica de la misma manera que en las propiedades aritméticas.
- La ley de Absorción dice que A. (A + B) dará igual a A. Esto surge de aplicar distributiva donde queda A . A + A . B, luego por Idempotencia A . A es igual a A. Hasta aquí nos queda A + A . B, luego en una OR entre la variable A y otro término que contiene la variable A dará como resultado la variable A. Esto es así por la ley de Idempotencia de la OR.
- La ley de DeMorgan sirve para simplificar una ecuación lógica donde una negación afecta a una o varias operaciones lógicas.

Por ejemplo, si tengo el producto $\overline{A.B}$, esto se puede remplazar por $\overline{A} + \overline{B}$. En otras palabras se individualiza la negación para que afecte solo a las variables y se cambia la operación AND por una OR.
Para el caso en que tenga una suma $\overline{A+B}$, se puede remplazar por las variables individuales negadas y una operación AND entre ellas. En otras palabras, se individualiza la negación para que afecte solo a las variables y se cambia la operación OR por una AND.

Por último, si aplicamos negación dos veces a una variable, se obtiene la misma variable. Esto significa que si una variable vale cero y se niega, valdrá 1. Ahora, si se vuelve a negar la variable, el 1 se transforma en cero que era el valor original.

Tabla 4. Leyes de Boole

Función OR	Función AND
A + 0 = A	A . 1 = A
A + 1 = 1	A . 0 = 0
A + A = A	A . A = A
A + A' = 1	A . A' = 0
A + B = B + A	A . B = B . A
(A + B) + C = A + (B + C)	(A . B) . C = A . (B . C)
A . (B + C) = A . B + A . C	(A + B)(A + C) = A + BC
A + AB = A	A . (A + B) = A
$\overline{(A + B)} = \overline{(A)}.\overline{B}$	$\overline{AB} = \overline{A}+\overline{B}$

2.5.2. Simplificación de circuitos

El objetivo es **reducir el número de compuertas,** lo que se refleja en una reducción en el costo de los componentes electrónicos. Por otra parte se disminuye el espacio que ocupan los circuitos integrados en las placas electrónicas.

Para cumplir estos requerimientos se necesita un circuito que realice la misma función original con menos compuertas. En la búsqueda de circuitos equivalentes, el álgebra booleana es la herramienta más adecuada y valiosa, debido que cada operación en una ecuación de Boole representa una compuerta, y si aplicamos las leyes de Boole podremos simplificar la ecuación, lo que equivale a una disminución en la cantidad de compuertas.

Ejemplo de simplificación:
Se debe simplificar $X = (AB+\overline{B})BC$
Usando la propiedad distributiva queda:
$X = ABBC + \overline{B}\ BC$
El paso siguiente es analizar cada término y ver qué propiedad de Boole se puede aplicar. En el primer término por Idempotencia B . B = B
$X = ABC + \overline{B}\ BC$

En el segundo término por ley inversa $\overline{B}$ por B es igual a 0
X = ABC + 0 . C
Luego por ley nula cero por cualquier variable da cero
X = ABC + 0
Por último cuando se hace una OR entre una o más variables y cero, se obtienen las mismas variables. La ecuación simplificada queda como se muestra a continuación
X = A . B . C

2.5.3. Expresiones booleanas desde tablas de verdad

A continuación se verán dos métodos para encontrar la expresión de Boole partiendo de la tabla de verdad. Los métodos desarrollados son *productos de sumas* y *sumas de productos.*

Producto de sumas: Se usa cuando la minoría de las salidas de la tabla son cero. Si se tiene la siguiente tabla:

A	B	X
0	0	1
0	1	0
1	0	1
1	1	1

En productos de sumas, los términos se vincularán con los demás mediante la operación AND (producto de Boole). Luego, dentro de cada término las variables se vincularán entre ellas mediante la operación OR (suma de Boole). Además, se tendrán tantos términos como salidas sean cero, en la tabla anterior hay un solo resultado cero y por lo tanto se tendrá un solo término.

Para este ejemplo hay una única salida que da cero. Entonces, la expresión de Boole se obtendrá de tener en cuenta las entradas de la tabla y poner las variables de tal manera que, para esas entradas, la salida de cero. En este ejemplo la entrada A vale cero y B vale uno, para el caso en que la salida da cero, es por eso que en la ecuación se deberá poner A sin negar y B negada para que el resultado de la OR entre ellas dé cero, cuando se tengan las entradas mencionadas. Entonces, el resultado de este ejemplo quedará:

$$X = (A + \overline{B})$$

Suma de productos: Se usa cuando la minoría de las salidas de la tabla de verdad es 1.

Por ejemplo:

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

En sumas de productos, los términos se vincularán con los demás mediante la operación OR (suma de Boole). Luego, dentro de cada término las variables se vincularan entre ellas mediante la operación AND (producto de Boole). Además, se tendrán tantos términos como salidas sean 1.

Para este ejemplo hay dos salidas que dan 1. Entonces, la expresión de Boole se obtendrá de tener en cuenta las entradas de la tabla y poner las variables de tal manera que para esas entradas la salida dé uno. En este ejemplo para el primer término la entrada A vale cero y B vale cero y la salida da uno, es por eso que en la ecuación se deberá poner $\overline{A}$ y $\overline{B}$ para que el resultado de la AND entre ellas dé 1.

Este término quedará: $(\overline{A}\ \overline{B})$

Queda para el lector realizar la deducción del otro término, sabiendo que para ese caso los valores de entrada de A y B valen 1

La expresión final quedará:

$$X = (\overline{A}\ \overline{B}) + (A.B)$$

2.6. Clasificación de los circuitos integrados

Los circuitos integrados son circuitos que incluyen en un chip varias compuertas y se clasifican en:

- **Circuito SSI:** *Small Scale Integrate*.(1 a 10 compuertas).
- **Circuito MSI:** *Medium Scale Integrate* (10 a 100 compuertas).
- **Ciruito LSI:** *Large Scale Integrate* (100 a 100.000 compuertas).
- **Circuito VLSI:** Very Large Scale Integrate > 100.000 compuertas.

2.7. Lógica combinacional

A medida que se incrementa la dificultad, será necesario conectar compuertas entre sí con la intención de generar una salida determinada para diferentes combinaciones de la variable de entrada.

Los circuitos que cumplen estas características se denominan lógica combinacional [2].

En la lógica combinacional la salida depende siempre del estado de las entradas.

Estudiaremos los siguientes circuitos lógicos

- Comparadores
- Decodificadores
- Codificadores
- Multiplexores
- Demultiplexores

2.7.1. Comparadores

- La función básica de un comparador consiste en comparar las magnitudes de dos números binarios para determinar si son iguales o distintos.
- Un comparador básico se implementa con la compuerta OR exclusiva o XOR. Debido a que esta compuerta devuelve un 1 lógico si las entradas son distintas y un cero si son iguales

El comparador de 2 bits requiere para su implementación el uso de dos compuertas XOR y para unificar la salida, se debe agregar un negador por compuerta y una compuerta AND a la salida como muestra la figura 5.

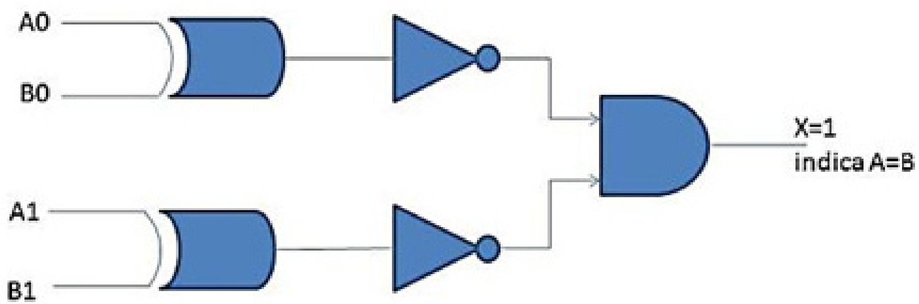


Figura 5. Comparador de dos datos de 2 bits

2.7.2. Decodificadores

La función de un decodificador es detectar la presencia de una determinada combinación de bits, los que generalmente representan un código. Cuando el decodificador detecta la presencia de este código genera una señal indicadora a la salida. En términos más generales, un decodificador puede tener n entradas y puede representar en una de las salida la presencia de un código.

Decodificador Binario

Supongamos que queremos determinar cuándo aparece el numero binario 1001 en la entrada de un circuito digital. Para resolverlo podemos pensar en una compuerta AND que produce una salida de nivel alto, solo cuando todas sus entradas son 1. Esto quiere decir que solo tenemos que asegurarnos de que todas las entradas sean 1 para el número 1001. El circuito de la Figura 6 nos permite conseguir dicho resultado.

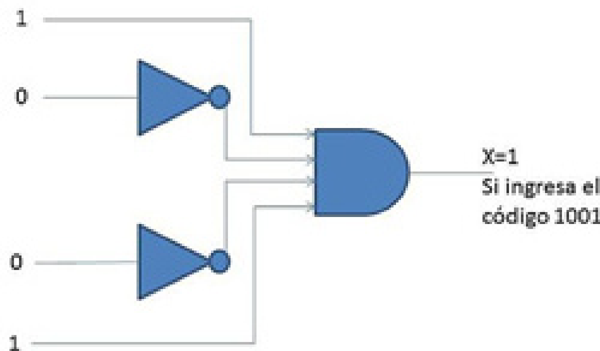


Figura 6. Decodificador básico

Aplicación de un decodificador en las Memorias RAM

Imaginemos una memoria de 4 chips, el chip cero tiene las direcciones de 0 a 1 MB, el chip uno de 1 a 2 MB, ... el 3 de 3 a 4 MB.

Cuando se presenta una dirección al sistema de memorias los 2 bits de orden más significativo sirven para seleccionar uno de los 4 chips, el resto de los bits indica una dirección dentro del chip seleccionado. El selector de chip se realiza mediante un decodificador.

Utilizando el circuito de la figura 7, donde los dos bits de entrada son representados por A, B y analizando las salidas, solo una de las cuatro salidas puede estar en 1 a la vez, dependiendo del valor de las entradas. De esta forma cada una de las salidas del decodificador habilita un chip de memoria.

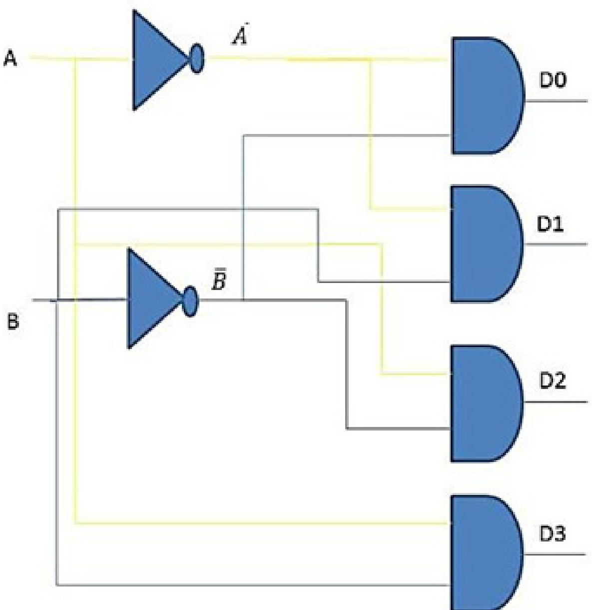


Figura7. Decodificador de dos entradas a cuatro salidas

2.7.3. Codificadores

Es un circuito lógico combinacional que realiza exactamente la función inversa del decodificador.

Un codificador permite que se introduzca en sus entradas un nivel alto que represente un dígito. Por ejemplo, se puede pensar en un programa ensamblador, como un codificador de software, ya que interpreta las instrucciones nemónicas en Assembler (ej, ADD, MOV) con las que se ha redactado un programa y lleva a cabo la codificación necesaria para convertir cada sentencia en una instrucción de código de maquina Binario (código de operación).

2.7.4. Multiplexores

Un multiplexor es un circuito que permite dirigir la información digital procedente de diversas fuentes a una única línea para ser transmitida a través ella hacia un destino común

El multiplexor básico posee varias líneas de entrada de datos y una única línea de salida. Los multiplexores también son conocidos como "selectores de datos".

La figura 8 muestra un multiplexor de cuatro entradas y una salida, donde para poder seleccionar entre las cuatro entradas se requieren dos líneas de selección.

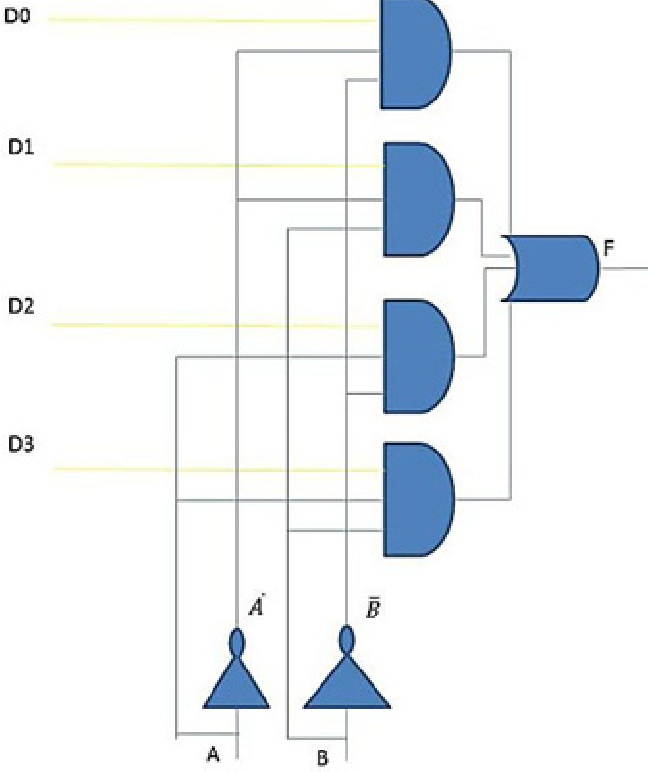


Figura 8. Multiplexor de cuatro entradas y una salida

Aplicación del multiplexor en el bus de comunicaciones de un sistema

- Un **bus** es una ruta interna por la que son enviadas señales eléctricas (datos) de una parte a otra de la computadora.
- Un **bus compartido** es aquel que está conectado a varios bloques de un sistema.
- Un bus compartido puede contener dispositivos de memoria y de entrada/salida a los que pueden acceder los procesadores del sistema.
- El acceso al bus compartido es controlado mediante un **arbitraje de bus**, un tipo especial de multiplexor.

2.7.5. Demultiplexores

Un demultiplexor básicamente realiza la función contraria del multiplexor. Toma datos de una línea de entrada y los distribuye a un determinado número de salidas. Conocido también como "distribuidor de datos".

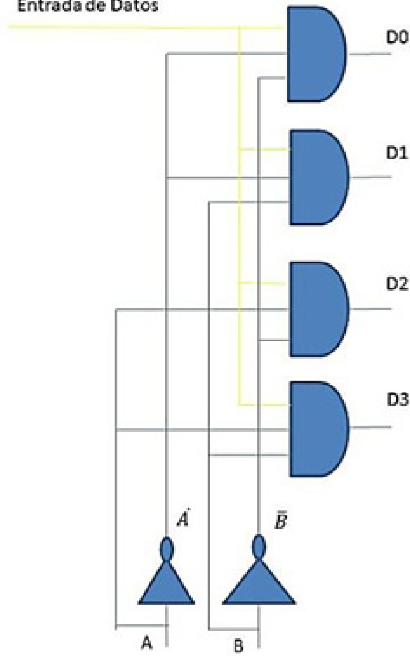


Figura 9. Demultiplexor de una entrada y cuatro salidas

2.8. Implementación de partes de la computadora mediante compuertas

Entre los elementos que se pueden generar mediante compuertas se encuentran [3]:

- Sumador completo
- ALU
- LATCH
- FlipFlop

2.8.1. Sumador completo

El sumador completo es modularizable, esto significa que si se requiere un sumador de dos datos de 8 bits se pueden interconectar ocho sumadores de 1 bit.

El sumador tiene las entradas de los bits a sumar, llamadas A y B. Además, tienen el acarreo de entrada que se utiliza para introducir un acarreo en la suma que proviene de los bits menos significativos. Por otro lado, el acarreo de salida permite trasladar un acarreo hacia los bits más significativos de la suma.

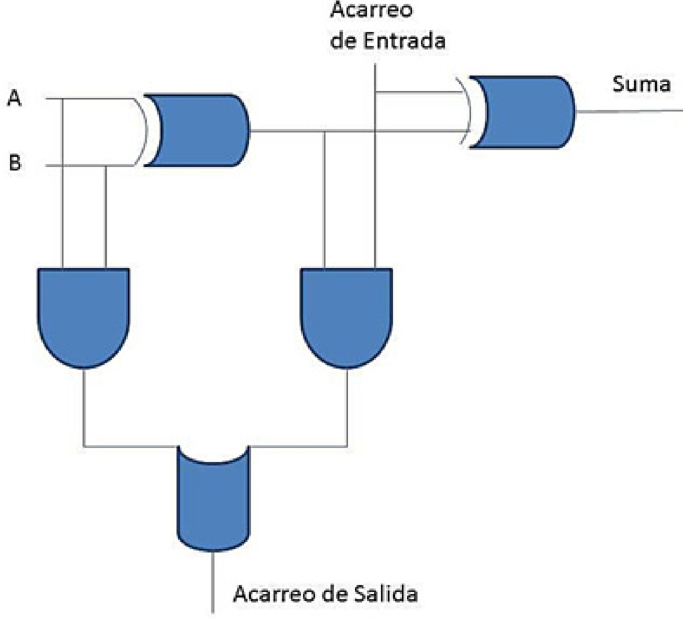


Figura 10. Sumador completo

2.8.2. Unidad Aritmético Lógica

La Unidad Aritmético Lógica (ALU) es la parte del procesador que se encarga de realizar las operaciones aritméticas y lógicas. El resto de los elementos de la computadora tienen como tarea proveer datos al procesador y ejecutar instrucciones. La ALU, como el resto de los componentes digitales de la computadora tienen su fundamento en los dispositivos básicos que hemos estudiado hasta el momento (las compuertas).

Una Unidad Aritmética Lógica debe calcular por lo menos las tres operaciones lógicas conocidas como AND, OR y NOT, además de las operaciones de suma/resta aritmética. Por lo general, una ALU, para procesar palabras de *n* bits se construye con "*n* circuitos" idénticos que hacen operaciones de 1 bit (escalabilidad). En la Figura 11 se muestra el circuito de una ALU de 1 bit, donde analizaremos su funcionamiento para las cuatro operaciones que se han descrito hasta el momento.

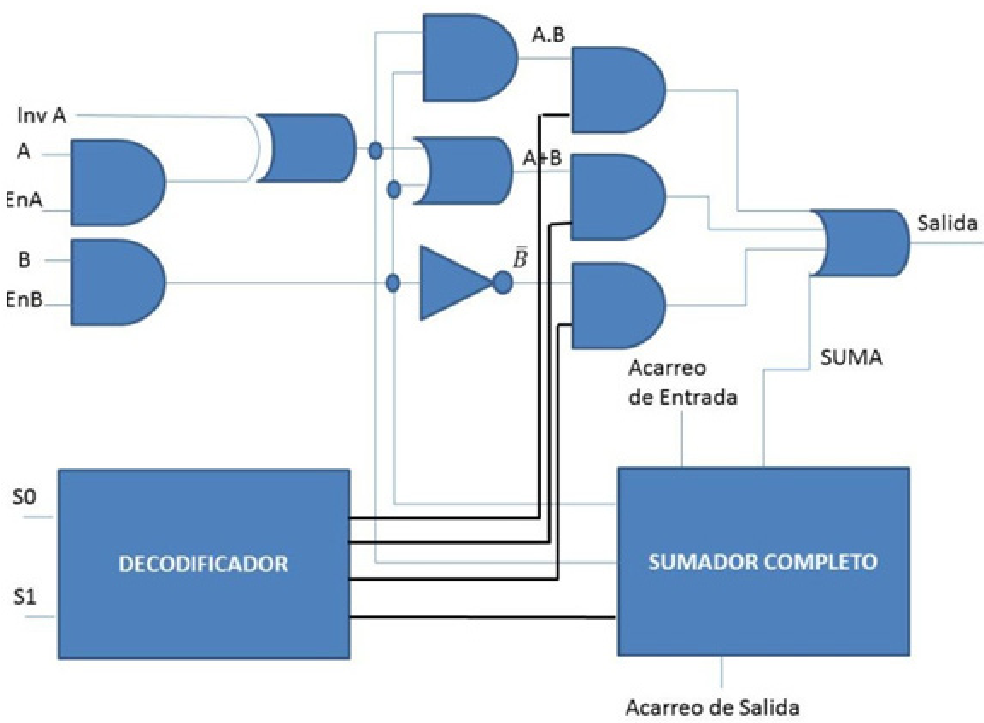


Figura 11. ALU de un bit

Las cuatro operaciones se pueden seleccionar mediante las líneas de entrada al decodificador S0 y S1. Las dos líneas de selección entran a un decodificador de dos a 4 bits. Como se describió antes, solo una de estas líneas estará habilitada para cada combinación de las entradas. La siguiente tabla muestra cuáles son los valores de S0 y S1 que permiten seleccionar las distintas operaciones.

Tabla 5. Esquema de selección del decodificado

S1:S0	OPERACIÓN
00	AND
01	OR
10	NOT
11	+

En la parte superior izquierda de la Fig. 11 se encuentra la lógica necesaria para realizar las tres operaciones lógicas descritas hasta el momento. Donde, solo uno de estos resultados pasará a la compuerta OR de la salida, dependiendo de las señales de habilitación S0 y S1.

Además de permitir usar las entradas A y B como entrada de operandos, es posible hacer que las dos sean cero poniendo en bajo ENA y ENB. También se puede obtener A negada activando INVA. En condiciones normales ENA = 1, ENB = 1 e INVA = 0.

Por último, el circuito de la parte inferior derecha corresponde a un sumador completo, que está encargado de calcular la suma entre A y B e incluye manejo de acarreos de entrada y de salida. Como se indicó antes, es común utilizar combinación de estos circuitos para hacer operaciones entre palabras de *n* bits, simplemente interconectando *n* bloques lógicos ALU de 1 bit.

2.8.3. Latch

Para crear una memoria de 1 bit es necesario un circuito que pueda recordar los estados anteriores de sus valores de entrada. Existen varias formas de crear estos circuitos con las compuertas que se han estudiado hasta el momento.

El **latch** es un dispositivo de almacenamiento temporal biestable (dos estados). Generalmente se agrupa en una categoría diferente a las de los Flip-Flop.

Latch $\overline{S}$ **-** $\overline{R}$

Son dispositivos lógicos biestables (dos estados). Un latch SR con entradas activas en nivel bajo está formado por dos compuertas NAND.

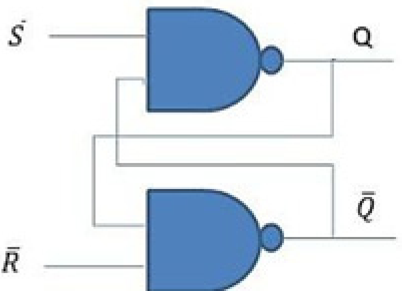


Figura 12. Latch $\overline{S}$ -$\overline{R}$

El latch tiene dos entradas $\overline{S}$ y $\overline{R}$ y dos salidas Q y $\overline{Q}$. La operación SET implica colocar en 1 Q y la operación RESET colocar en 0 Q. A diferencia de los circuitos combinacionales las salidas no dependen únicamente de las entradas.

La figura 13 muestra el circuito lógico equivalente al de la figura 12, que es más simple de analizar y muestra como la salida Q se pone a 1 lógico cuando la entrada S recibe un nivel bajo (cero lógico).

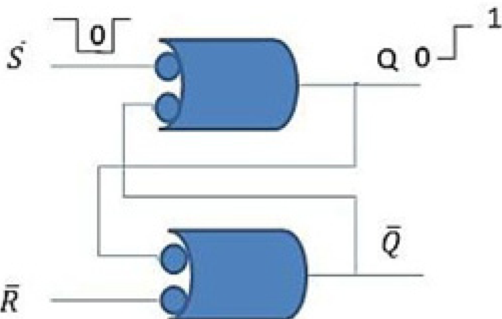


Figura 13. Colocar un 1 en Q cuando el valor anterior era un cero

Si se ingresa un nivel bajo por la línea RESET ($\overline{\boldsymbol{R}}$) cambiará a cero la salida Q. Por otro lado, cuando se pone un nivel bajo en las entradas SET y RESET a la vez, se genera un estado invalido debid o a que no se puede guardar un 1 y un cero a la vez en una unidad de memoria. Finalmente, cuando las entradas SET y RESET se mantienen en alto (1 lógico) las salidas mantienen su estado (condición de no cambio).

Latch S-R (Set-Reset) con entrada de habilitación

S y R Controlan el estado al que va a cambiar el latch cuando se aplica un nivel alto en la entrada de habilitación EN (Enable). El latch no cambia de estado hasta que la entrada EN está en nivel alto. Se sigue presentando el estado no válido.

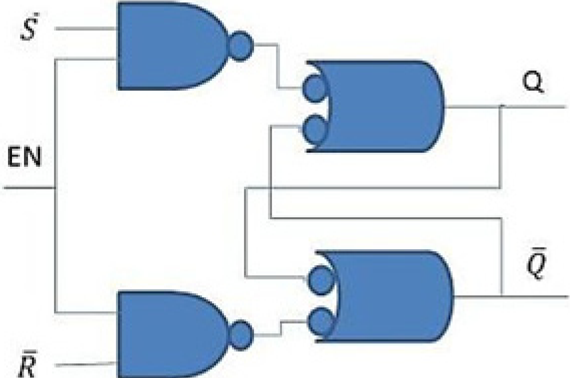


Figura 14. Latch con entrada de habilitación

Latch D (Date) Con entrada de habilitación

Se diseñó con la intención de resolver el estado de ambigüedad del latch $\overline{\boldsymbol{S}}\,\overline{\boldsymbol{R}}$ (S = R = 0). El circuito tiene una sola entrada D, que cuando está en nivel alto y la habilitación EN también está en nivel alto, el latch se coloca en estado SET (1). Cuando la entrada D está en nivel bajo y la habilitación EN está en nivel alto, el latch se coloca en RESET (0). Mientras la línea EN se mantenga en nivel bajo la salida no cambiará.

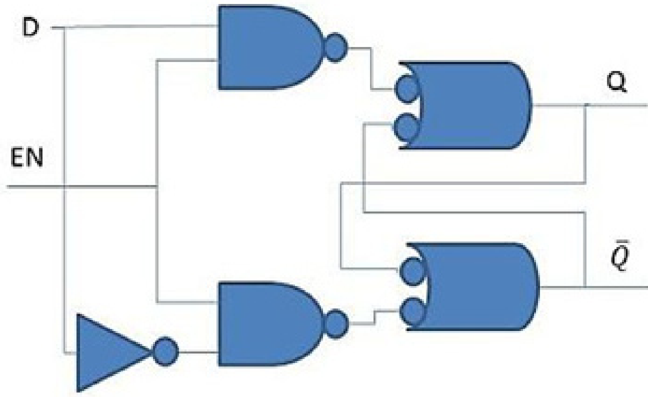


Figura 15. Latch tipo D

2.8.4. Flip-flop

Los flip-flop, son también dispositivos de dos estados que pueden permanecer en cualquiera de ellos. Esta capacidad se debe a la realimentación de cada una de las salidas con la entrada opuesta.

La diferencia principal entre estos dos tipos de dispositivos es el método que se utiliza para cambiarles el estado (valor de Q). Los flip-flops son dispositivos sincrónicos de dos estados. Para el caso de los flip-flops la definición síncrona indica que la salida cambia de estado únicamente cuando en la entrada de reloj hay un flanco ascendente.

Para diseñar un flip-flop es necesario generar un pequeño pulso en el flanco ascendente de la señal de reloj. En la figura 16 se puede observar el circuito generador de flancos en la entrada CLK. La diferencia entre un flip-flop y un latch es que un flip-flop se dispara por un flancos y un latch se dispara por nivel.

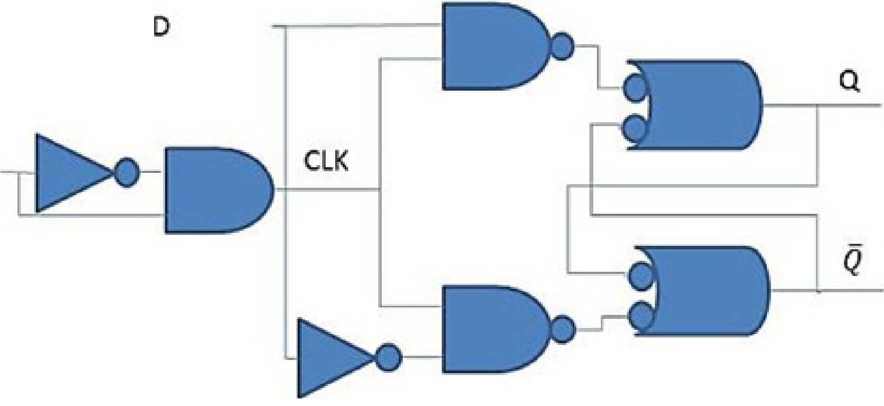


Figura 16. Flip Flop D

Los flip flops son unidades de memoria que se utilizan en los sistemas de procesamiento. Su principal aplicación se da en la fabricación de registros y memorias cache como unidades de memoria de alta velocidad.

3 Microprocesadores y microcontroladores

Es común que se defina al procesador como el cerebro de la computadora, ya que una de sus funciones es ejecutar los programas almacenados en la memoria principal de la computadora, que involucran una amplia variedad de operaciones.

Los componentes que integran esta computadora sencilla están interconectados por un bus. El bus es una colección de conductores (Ej.: pistas de una placa) dispuestos de forma paralela que se utilizan para trasmitir las señales (datos, direcciones y señales de control) hacia y desde la memoria.

3.1. Características generales de un microprocesador

En un sistema microprocesador básico, (basado en la arquitectura de la máquina de Von Newman), como el que muestra la figura siguiente, se distinguen la CPU, el bloque de memoria y los módulos de Entrada/Salida. La CPU representa al microprocesador y junto con el resto de los bloques conforma la máquina de Von Newmann.

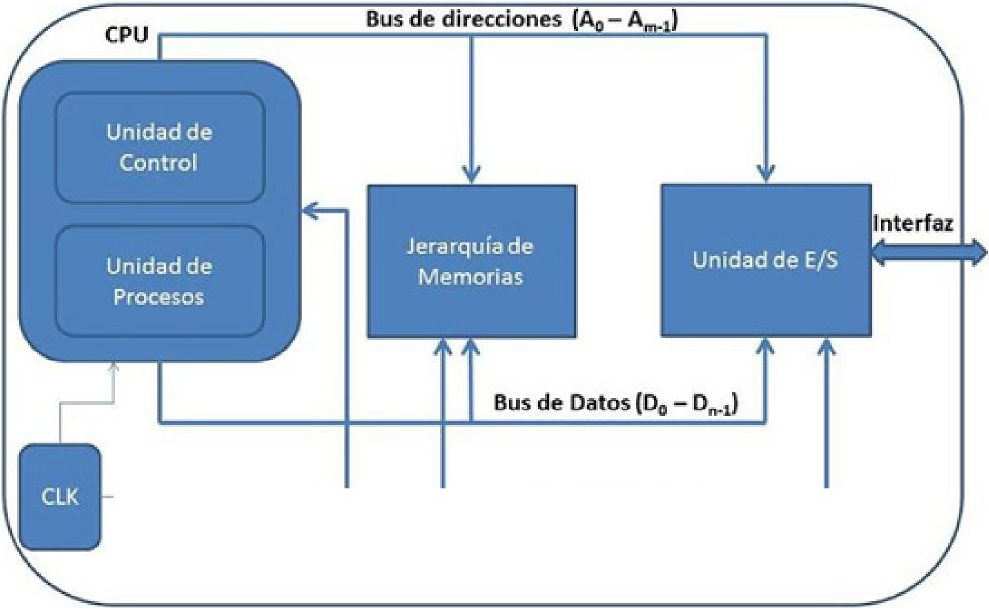


Figura 17. Diagrama de la computadora de von Newmann

La **CPU** es la Unidad Central de Procesos (procesador) y está formada por la Unidad de Control y la Unidad de procesos.

La **memoria** se divide en memoria de programa y memoria de datos, en la que se almacenan las instrucciones de programa y de datos, respectivamente.

La Unidad de entrada salida permiten intercambiar información con el mundo exterior.

Todo el sistema debe estar sincronizado para funcionar correctamente, es por eso que se utiliza un clock de sincronismo que posibilita la interacción entre todos los bloques de la PC de manera ordenada y coordinada.

Los tres bloques principales deben estar vinculados mediante un bus de comunicación, (bus de datos, bus de direcciones y bus de control), por donde viajan los datos digitales.

3.1.1. Descripción de los buses de comunicación

3.1.1.1. Bus de Datos

El bus de datos permite al microprocesador intercambiar datos con los bloques de memoria y los módulos de E/S. El bus de datos tiene la particularidad de ser bidireccional. Permite el envío y recepción de datos. Este bus tiene una cantidad de líneas que representa el tamaño de palabra (cantidad de bits de los datos) con la que opera el microprocesador.

Las líneas del bus de datos deben ser tri-state, por lo que pueden estar en bajo (L), en alto (H) o en alta impedancia (HZ). Entonces, cuando un dispositivo se encuentra conectado al bus de datos, sus líneas podrán adquirir los estados L y H, luego los demás dispositivos estarán con sus líneas en HZ (desconectados eléctricamente).

3.1.1.2. Bus de direcciones

Mediante este bus el microprocesador selecciona la dirección de memoria o el dispositivo de E/S con el cual desea intercambiar información. Por este motivo dicho bus es unidireccional y la cantidad de líneas que lo constituyen determinan la cantidad de direcciones de memoria que puede direccionar la CPU (máximo direccionamiento). Si el bus tiene 16 líneas, esto quiere decir que la CPU podrá direccionar 2^{16} posiciones de memoria (65.536 posiciones).

3.1.1.3. Bus de control

Por estas líneas circulan las señales de control y sincronización del sistema, entre las más comunes se encuentran:

- Señal de clock
- Señal de reset
- Señal de lectura/escritura de memoria

3.1.2. Descripción del bloque de memoria

La memoria se puede separar en dos bloques bien definidos, el de memoria de datos y el de instrucciones.

3.1.2.1. Memoria de datos

La memoria de datos se utiliza para almacenar los datos y variables del programa. Estos datos son normalmente de 8 bits (Byte), 16 bits (Word) o 32 bits (Long Word). Para el almacenamiento de datos y variables se utiliza memoria RAM, la cual es de lectura/escritura y se caracteriza por ser volátil.

3.1.2.2. Memoria de instrucciones

La memoria de instrucciones contiene el programa que ejecuta el microprocesador, que está formado por instrucciones que se ejecutan de manera secuencial. En dicha memoria las instrucciones se codifican mediante un Código de Operación (OP), formado por uno o varios bytes. Para el almacenamiento del programa se utiliza memoria ROM (no volátil), que es de solo lectura y permite almacenar el programa a ejecutar por el microprocesador y valores constantes a ser utilizados por dicho programa.

3.1.3. Descripción de la CPU

La unidad central de procesos está formada por registros internos, la unidad de control y la unidad de procesos.

3.1.3.1. Registros internos

Estos registros son bloques biestables (flip-flop) que permiten almacenar los datos básicos con los cuales va a trabajar la CPU para almacenar operandos y resultados durante la ejecución de las instrucciones que forman el programa.

En el registro, el control de entrada es gobernado por una señal de sincronismo y el valor del registro quedará disponible a la salida cuando el control de salida CS este en "0 lógico", cuando CS esté en "1" la salida estará en alta impedancia (deshabilitada).

En la Figura 18 se ejemplifica el funcionamiento de los registros mediante el envío de datos de un registro a otro, en donde se observa que para enviar un dato desde el registro A hacia el B, se debe colocar el control de salida "CS1" del registro A en bajo, esperar a que el Bus 2 esté disponible, y luego poner en alto el control de entrada "CE2" del Registro B, y por último poner CE1 en alto.

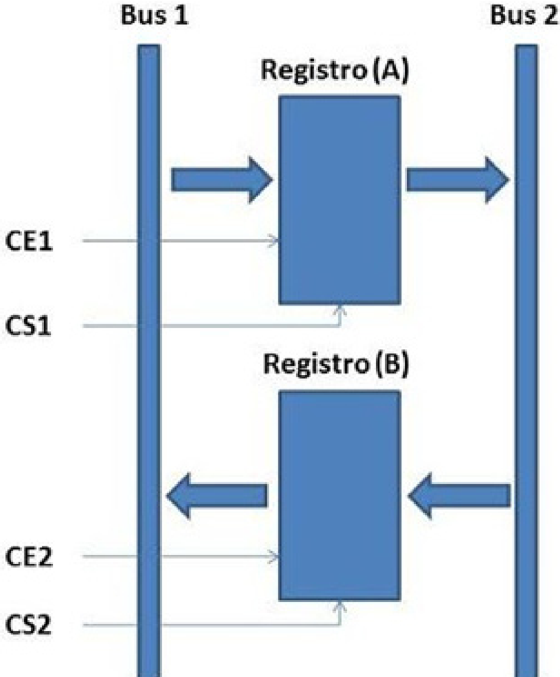


Figura 18. Esquema de una transferencia de datos entre registros

Es importante destacar que los datos almacenados en los registros de la CPU tienen un tiempo de acceso mucho menor al de los datos almacenados en la memoria interna y externa.

3.1.3.2. Unidad de control

La unidad de control de la CPU se encarga de decodificar las instrucciones almacenadas en memoria y ejecutar las microoperaciones que la conforman.

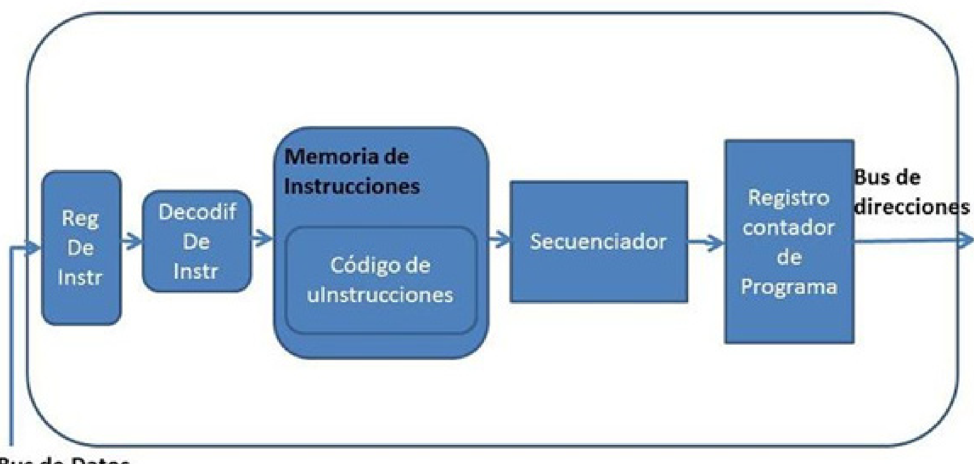


Figura 19. Estructura interna de la unidad de control

En la Figura 19 se muestra un diagrama de la unidad de control, en donde se recibe el código binario de la instrucción a ejecutar por el bus de datos y se almacena en el registro de instrucciones. Luego, el decodificador de instrucciones interpreta el código de operación de la instrucción que será ejecutada mediante microinstrucciones ubicadas en una Memoria ROM de la CPU llamada "Memoria de instrucciones" o "Memoria de microprograma".

El tamaño del OPCODE de las instrucciones codificadas es de 8 bits para el procesador en estudio, el HC08. Este código es cargado en el registro de instrucciones y extendido de 8 a 12 bits mediante la adición de cuatro ceros a la derecha. Este nuevo valor es la dirección de entrada a la memoria de microprograma en donde comienzan a ejecutarse las microinstrucciones y a realizarse las microoperaciones que se corresponden con la instrucción en ejecución.

De acuerdo con la secuencia de microinstrucciones, el bloque secuenciador se encargará de activar un conjunto de señales que se corresponden con las microoperaciones y le indican a los diferentes componentes de la arquitectura la operación a realizar.

La dirección de la próxima instrucción a ejecutar es provista por el secuenciador y puede provenir de tres lugares diferentes: del registro de instrucción, del controlador de interrupciones o de la memoria de microprograma.

- El registro de búsqueda de dirección, registro de decodificación de dirección y el contador de programa (PC) contienen, respectivamente, las direcciones de las instrucciones que en el ciclo actual van a ser buscadas, decodificadas y ejecutadas.
- El PC está asociado con la pila, que se utiliza para almacenar direcciones de retorno y direcciones de comienzo de los saltos a subrutinas o a códigos de interrupciones.
- El controlador de interrupciones realiza todas las funciones relacionadas con el procesamiento de las interrupciones, tales como determinar cuándo una interrupción está enmascarada y generar la dirección del vector de interrupción asociado.

El **Contador de Programa (PC)** es un registro que contiene la dirección de memoria donde se encuentra la siguiente instrucción a ejecutar.

3.1.3.3. Unidad de procesos

El bloque principal es la **ALU o Unidad Lógico-Aritmética** (ver figura 20), que permite realizar las operaciones aritméticas y lógicas básicas como sumas, restas, AND, OR, OR exclusiva, negación, etcétera.

El secuenciador de la unidad de control activa las líneas de selección de la ALU para realizar la operación indicada por la instrucción en curso.

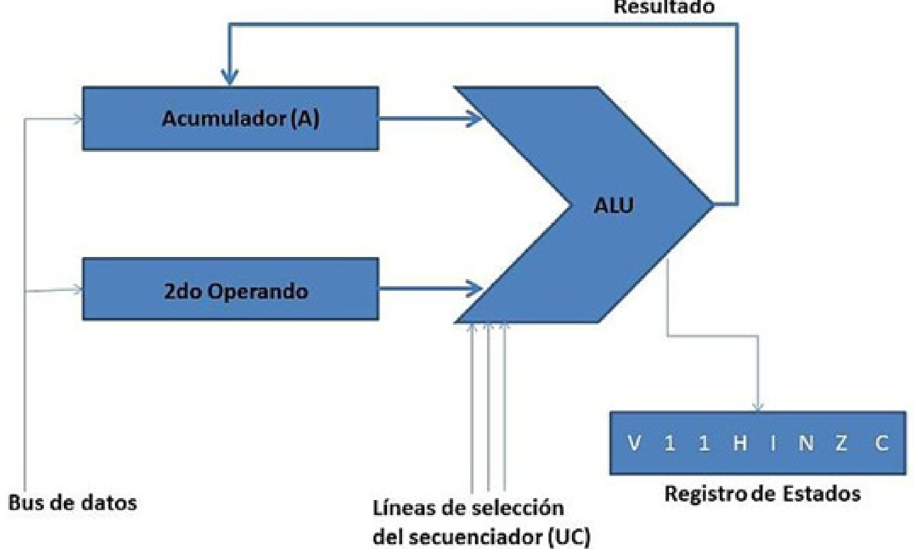


Figura 20. Estructura interna de la unidad de procesos

Los operandos se suministran por medio de dos registros cargados desde el bus de datos, uno de los registros es el **acumulador**, que se utiliza para realizar todas las operaciones, y el otro registro puede ser uno cualquiera o una dirección de memoria, que aquí denominamos **Registro 2º Operando.**

- **Registro Acumulador:** Contiene siempre el resultado de la última operación realizada en la ALU.
- **Registro 2º Operando:** Es el segundo operando necesario para realizar la operación que se corresponde con la instrucción a ejecutar, el código de operación de la instrucción a ejecutar indica que registro representará este operando, según los diferentes modos de direccionamiento.

Esta forma de realizar las operaciones permite simplificar la ejecución de las instrucciones, debido a que cada instrucción solo tiene que suministrar un operando, el otro se encuentra cargado previamente en el acumulador.

El **registro de estado** está formado por bits denominados banderas (flags) que se ponen a 1 o 0 de acuerdo con el resultado de la instrucción ejecutada. Los flags más comunes son:
- Z, bit zero, se pone en 1 si el resultado es cero.
- C, bit carry, se pone en 1 si hubo acarreo de orden superior.
- V, bit overflow, se pone en 1 si hubo desborde.
- I, bit de interrupción; Este bit es independiente del resultado. Poniendo este flag en 1 se pueden inhibir las interrupciones enmascarables.

3.1.4. Descripción de la unidad de entrada/salida

Permiten la comunicación del sistema microprocesador con otros dispositivos. Los dispositivos de E/S se denominan habitualmente **periféricos**, por ejemplo, teclados, displays, unidades de memoria, etcétera.

Cualquier periférico necesita un módulo adicional que permite realizar la conexión de este con los buses del sistema Microprocesador, este módulo se denomina **interfaz**:

Existen varios métodos para manejar los dispositivos de E/S:

1.- Mediante **instrucciones específicas de E/S**, que se emplean en el programa de control para acceder al periférico.

2.- Mediante **Acceso directo a memoria (DMA)**. La CPU pone en los buses de direcciones y de datos en triestado. Un dispositivo controlador de DMA toma el control de los buses y pasa los datos directamente entre el dispositivo E/S y la memoria.
3.- Mediante **Técnicas de interrupción**. El periférico activa las líneas de interrupción de la CPU, que detienen el programa en ejecución y trasladan el contador de programa a la dirección de inicio de la "subrutina de interrupción", creada especialmente para atender al periférico que solicita la interrupción.
4.- Mediante el **tratamiento de las E/S como posiciones de memoria**. Permite el empleo de las mismas instrucciones para acceso a memoria que para E/S. Una zona del mapa de memoria es reservada para los dispositivos de E/S. Estas posiciones se llaman **Puertos de E/S**.

3.1.5. Ejemplo de ejecución de una instrucción en un microprocesador de 8 bits

La ejecución de una instrucción se lleva a cabo en dos fases:

Fase de búsqueda

Se inicia en el contador de programa (PC), que contiene la dirección de memoria en la que se encuentra el código binario de la instrucción. Esta dirección se coloca en el registro de direcciones de la CPU y de ahí a la memoria a través del bus de direcciones, como se muestra en la figura 21(flecha de color negro). Una vez decodificada la dirección en la memoria, su contenido se traslada al bus de datos hacia el registro de instrucciones de la unidad de control, como se muestra, (con color verde), en la figura siguiente.

Fase de ejecución

En esta fase se decodifica la instrucción dentro de la unidad de control. Se busca su código de microinstrucciones en la memoria interna de la CPU y se activan las señales correspondientes del secuenciador para ejecutar la acción indicada por dicha instrucción. En el caso de la instrucción LDA que se ejecuta en la figura, las señales del secuenciador se corresponderán con la búsqueda del operando que está en la dirección 000…0001 cuyo valor es el 7, para luego almacenar dicho valor en el registro acumulador.

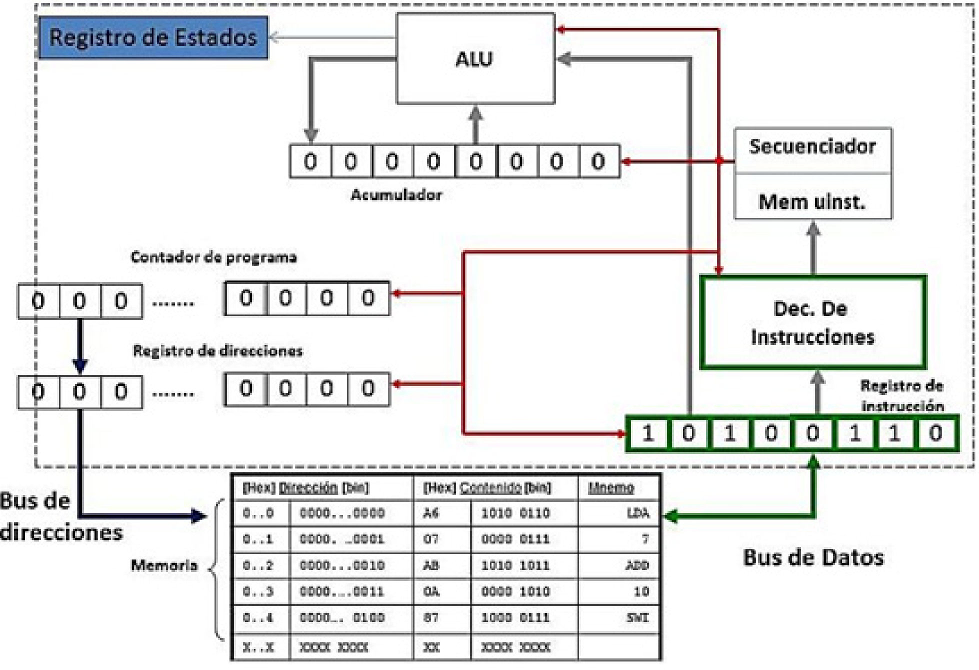


Figura 21. Diagrama general de un microprocesador de 8 bits.

Las instrucciones que constituyen el programa se almacenan en memoria en paquetes de 8 bits (Bytes). El primer byte indica el código de operación (***COP o OPCODE***), que representa la operación que realiza la instrucción. El o los siguientes bytes normalmente proporcionan la información necesaria para acceder al dato (operando) sobre el que va a operar la instrucción. Estos bytes pueden ser el propio dato, la dirección de memoria donde se encuentra el dato, etc. Las diferentes posibilidades para acceder a ese dato se denominan **modos de direccionamiento** del microprocesador y se desarrollarán en el capítulo 6.

3.2. Características generales de un microcontrolador

Un **microcontrolador** puede definirse como un sistema integrado por: CPU, memoria, reloj oscilador y módulo I/O (módulo de entrada/salida) en el mismo circuito integrado. Cuando carece de alguno de estos elementos, ya sea I/O o memoria, el circuito integrado lleva el nombre de **microprocesador** [3] y [4].

El MC68HC908 es una unidad microcontroladora (MCUs) de bajo costo, alto desempeño de la familia M68HC08 de 8-bit [5]. Esta familia tiene un conjunto

de instrucciones complejas (CISC) con una Arquitectura Von Neumann. Todos los MCUs de la familia M68HC08 usan una unidad microprocesadora llamada “CPU08” que se describe en el capítulo 5, junto con una amplia variedad de módulos y un espacio de memoria RAM y Flash que permiten realizar múltiples aplicaciones.

4 Eficiencia y rendimiento en sistemas de procesamiento

Gene Amdah enunció una fórmula para determinar la potencial velocidad de ejecución de un programa usando múltiples procesadores [6].

– Se concluyó que para que la Aceleración (*speedup*) sea alta, el código tiene que ser paralelizable.

Para comparar la ejecución de un programa corriendo en un simple procesador con una máquina de múltiples núcleos se tiene la siguiente definición:

- **f:** Fracción de código infinitamente paralelizable sin sobrecarga de programación.
- **(1-*f*):** fracción de código no paralelizable.
- **T:** tiempo total de ejecución del programa en un simple procesador.
- **N:** es el número de procesadores que aprovechan al máximo las porciones de código paralelizables.

$$speedup = \frac{\textit{tiempo de ejecución de un programa en simple procesador}}{\textit{tiempo de ejecución de un programa en múltiples procesadores}}$$

$$= \frac{1}{(1-f) + \frac{f}{N}}$$

Conclusiones:

— Para f pequeña, los procesadores paralelos tienen poco efecto.
— Cuando el *número de procesadores N* en muy grande la velocidad tiende a $1/(1 - f)$, esto significa que el rendimiento es decreciente con el uso de gran cantidad de procesadores

4.1. Aclaraciones de Amdahl

• "Estas conclusiones son demasiado pesimistas", se afirmó en un congreso. Allí se dio como ejemplo el caso en el que un servidor puede mantener varios subprocesos o tareas múltiples para manejar diversos clientes y ejecutar los hilos o tareas

en paralelo hasta el límite del número de procesadores. Muchas aplicaciones de base de datos implican cálculos con cantidades masivas de datos que se pueden dividir en múltiples tareas en paralelo.

• Sin embargo la ley de Amdahl ilustra los problemas que enfrenta la industria en el desarrollo de procesadores multinúcleo, en máquinas con un número cada vez mayor de núcleos. El software que se ejecuta en máquinas de este tipo debe adaptarse a un entorno de ejecución altamente paralelo para aprovechar la potencia del procesamiento paralelo.

4.2. Generalización de la ley de Amdahl

La ley de Amdahl puede ser generalizada para evaluar cualquier diseño o mejora técnica en un sistema informático. Si se considera cualquier mejora de la característica de un sistema que resulta en un aumento de velocidad, la aceleración se puede expresar como:

$$\text{Aceleración} = \frac{\text{Rendimiento después de la mejora}}{\text{Rendimiento antes de la mejora}} = \frac{\text{tiempo de ejecución antes de la mejora}}{\text{tiempo de ejecución después de la mejora}}$$

• La generalización de Amdahl establece que la mejora obtenida en el rendimiento al utilizar algún modo de ejecución más rápido está limitada por la fracción de tiempo en el que se pueda utilizar ese modo más rápido.

• La generalización de Amdahl define la ganancia de rendimiento o aceleración (*speedup*) que puede lograrse al utilizar una característica particular.

Ejemplo 1:

• Si una tarea hace un amplio uso de operaciones de punto flotante, con 40% del tiempo consumido por operaciones de punto flotante.

• Con un nuevo diseño de hardware, el módulo de punto flotante se acelera en un factor K. entonces, el aumento de velocidad en general es:

$$speedup = \frac{1}{0{,}6 + \frac{0{,}4}{k}}$$

Ejemplo 2:

- Se actualiza un equipo que mejora el modo de ejecución de instrucciones en un factor 15.
- Este nuevo modo rápido se utiliza durante el 40% del tiempo, ¿cuál es la aceleración global en este caso?

Para este ejercicio se debe utilizar la siguiente relación:

$$Aceleración= \frac{Rendimiento\ después\ de\ la\ mejora}{Rendimiento\ antes\ de\ la\ mejora} = \frac{(0,4\ x15)+0,6}{0,4\ +0,6}$$

5 Descripción del procesador CPU08

El CPU08 [7] y [8] es el procesador de los MCUs (microcontroladores) de la familia HC908. Dentro de la estructura interna de un HC908, el módulo del CPU se vincula con el resto de los módulos del MCU por medio de un bus de datos interno de 8 bits, y un bus de Direcciones de 16 bits, que le permite direccionar código de hasta 64K bytes. Este bus es denominado IBUS (Bus interno). La frecuencia máxima del bus interno es de 8 MHZ. Esta frecuencia de bus implica que cada ciclo de *Clock* (reloj) del bus es de 125 nS.

5.1. Arquitectura de ejecución del CPU 08

El CPU 08 está dividido en dos bloques; la "**Unidad de control** (*Control Unit*)" y la "**Unidad de procesos** (*Process Unit*)".

El primer bloque contiene una máquina de estados finitos, unidades de control y tiempo, para manejar la "unidad de ejecución". Además, dos señales de la Unidad de control manejan la prebúsqueda "*prefetch*" y la carga de instrucciones, una es "*Opcode Look ahead*" (Señal para la operación de prebúsqueda) y la otra es *Lastbox* (Señal para el último ciclo de la instrucción en curso).

El segundo bloque contiene la ALU (Unidad aritmético lógica, encargada de todas las operaciones lógicas y aritméticas binarias), Registros internos de CPU (Acumulador, puntero de pila, contador de programa, registro índice y registro de código de condiciones (CCR)) y la interfaz con el bus interno.

El CPU 08 pertenece a la arquitectura del tipo "Von Neumann" clásica ([3] y [4]), característico de la familia 68xx de Motorola y ampliamente utilizado en el mundo, (en el apéndice A se describen las arquitecturas clásicas más utilizadas en el diseño de microcontroladores). En este tipo de arquitectura, existe un solo bus de datos, tanto para memoria de programas como para memoria de datos, lo que da origen a un mapa "lineal" de acceso a memoria y por consiguiente no existen instrucciones especiales y diferentes para trabajar con "datos" o con "código de programa". De esta forma, todas las instrucciones son aplicables en cualquier parte del mapa de memoria sin importar si se trabaja con datos o código. Es por esto por lo que no es raro encontrar aplicaciones cuyos programas corren desde RAM como si estuvieran en Flash. Se debe destacar que esto no podría hacerlo una arquitectura Harvard clásica.

5.2. El “Prefetch” en el CPU08

El CPU08 implementa “Códigos de operación” con un mecanismo de prebúsqueda (*Prefetch*) hacia adelante. Esto incrementa el desempeño eliminando tantos “ciclos muertos” de bus como sea posible. Así se obtienen instrucciones con menor número de ciclos de reloj, y mejora la velocidad real de ejecución de código.

El flujo de instrucciones del CPU08 fue desarrollado para ser tan eficiente como sea posible en una estructura del tipo *pipeline*. Gracias a esta estructura es que se ahorran tantos ciclos de reloj como sea posible en la ejecución de distintas instrucciones del CPU08.

5.3. Unidad de control

La unidad de control del HC08 está formada por un secuenciador, un bloque de memoria de control y lógica de control aleatorio. Estos bloques forman una máquina de estados que genera todos los controles para la unidad de procesos. El secuenciador provee el próximo estado de ejecución de la instrucción basado en el contenido del registro de instrucción (IR) y el estado actual de la ejecución.

La memoria de control ejecutará las microinstrucciones, produciendo una salida que representa el próximo estado decodificado para la unidad de procesos. Este resultado, con la ayuda de lógica aleatoria, se usa para generar la salida de la unidad de control que configura la unidad de procesos. La lógica aleatoria selecciona la señal apropiada y agrega tiempos a las salidas de la memoria de control. Durante la ejecución de una instrucción la unidad de control dispara una señal por ciclo de bus, pero tarda casi un ciclo completo en decodificar y generar todos los controles antes de enviar las señales a la unidad de procesos.

El secuenciador también controla un registro (contador de programa) que es usado para precargar la próxima instrucción a ejecutar.

5.4. Unidad de procesos

La unidad de ejecución (UE) contiene los registros, la unidad aritmético lógica (ALU) y la interfaz del bus.

Por cada ciclo de bus se calcula una nueva dirección pasando valores por el registro seleccionado a lo largo de los buses de direcciones internos hasta los *buffers* de direcciones. La unidad de procesos también contiene algunas funciones lógicas especiales para instrucciones inusuales como multiplicación sin signo (MUL) y división (DIV).

5.5. Descripción de registros de la CPU

La CPU [8] contiene cinco registros tal como se lo presenta en la figura 22. Los registros de la CPU son registros de memoria que se alojan dentro del microprocesador (no son parte del mapa de memoria).

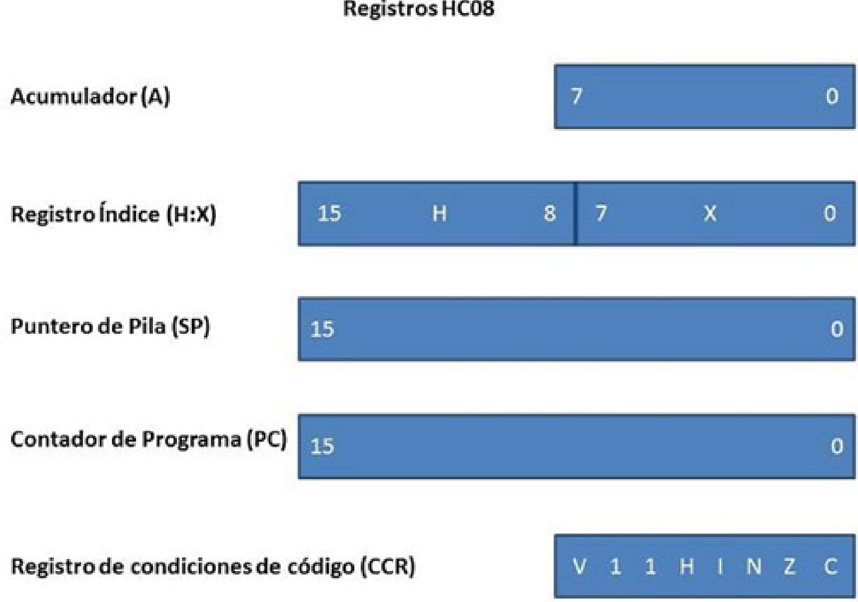


Figura 22. Registros del CPU08.

En el CPU08 (CPU de la familia HC908) se pueden ejecutar una amplia variedad de instrucciones complejas y simples (Ver *Apéndice B*) con sus respectivos modos de direccionamiento.

5.5.1. Acumulador (A)

El acumulador es un registro de propósitos generales de 8 bits usado para almacenar operandos, resultados de cálculos aritméticos y de manipulación de datos. Además, es directamente accesible desde la CPU para operaciones no aritméticas. El acumulador es usado durante la ejecución de un programa donde el contenido de alguna posición de memoria es almacenado en él. También, la instrucción *almacenar* (sta) causa que el contenido del acumulador sea almacenado en alguna posición de memoria preestablecida.

5.5.2. Registro índice (H : X)

El registro índice se utiliza para todas las operaciones indexadas (con y sin offset) que posee el CPU. Tiene 16 bits de longitud, formado por una parte "baja" (el byte de menor peso) denominado "X" y una parte alta (el byte de mayor peso) denominado "H". Estos registros se encuentran concatenados para formar un único registro H : X. Esto permite direccionamientos indexados de hasta 64 Kbytes de espacio de memoria.

En el registro índice puede utilizarse la parte baja ("X"), en los distintos modos de direccionamiento. Solo se debe tener en cuenta que cuando en una instrucción con direccionamiento indexado, se menciona el registro "X", en realidad se está haciendo mención al registro concatenado H : X de 16 bits de largo, por lo que deberá ponerse a cero (forzar el valor $00) la parte superior del registro índice, o sea "H", para mantener total compatibilidad con la familia anterior de microcontroladores (cuyo registro índice era de 8 bits). De esta forma, cuando se utilice el registro índice, el contenido del mismo será $00xx, donde "xx" contendrá el valor del registro "X" propiamente dicho.

5.5.3. Puntero de pila (SP)

El puntero de pila (SP) es un registro de 16 bits que contiene la dirección del próximo lugar en la pila (*Stack*). Es el registro utilizado por el CPU para mantener en "orden" (guardar y rescatar los datos en RAM) los registros principales del CPU ante una excepción en la secuencia del programa, como lo son los saltos a subrrutinas y los distintos pedidos de interrupción.

Durante un reset, el puntero de pila, es preseteado a $00FF. La instrucción *Reset Stack Pointer* (RSP), cambia el byte menos significativo a $FF y no afecta al byte más significativo.

El puntero de pila es reducido cuando un dato es almacenado (*Push*) dentro de la pila e incrementado cuando un dato es recuperado (*Pull*) desde la pila.

La localización de la pila es arbitraria, y puede ser "reubicada" en cualquier parte de la memoria RAM. Moviendo el puntero fuera de la página cero o página directa ($0000 a $00FF) se libera el espacio del direccionamiento directo. Para una operación correcta, el puntero de pila debe apuntar solamente a posiciones de RAM, aunque por su longitud, pueda "barrer" todo el espacio de memoria del MCU. Gracias a esta característica, en los modos de direccionamiento con el SP (*Stack Pointer*) con 8 bits y 16 bits de *offset*, el puntero de pila (SP) puede

funcionar como un segundo registro índice de 16 bits o bien para acceder a datos en la pila. El uso del SP como un segundo registro índice es muy utilizado en los compiladores de lenguaje de alto nivel como los compiladores “C” y otros.

5.5.4. Contador de programa (PC)

El Contador de programa (PC) es el registro utilizado por el CPU para mantener el control de las direcciones de las próximas instrucciones a ser ejecutada. Este registro tiene una longitud de 16 bits, y tiene la ventaja de poder moverse entre $0000 y $FFFF. De esta forma, el PC puede moverse por los 64 Kbytes de espacio de memoria (salvo en los microcontroladores que tienen memoria menor a 64 Kbytes).

Durante el *Reset*, el contador de programa (PC) se carga con la dirección contenida en el “vector de reset” que para el MC68HC908 se encuentra en las posiciones $FFFE y $FFFF. La dirección contenida en el vector, es la dirección de la primera instrucción a ser ejecutada después de salir del estado de *Reset*.

5.5.5. Registro de código de condición (CCR)

El registro de código de condición contiene una máscara de interrupción y cinco indicadores de estado que reflejan el resultado de operaciones aritméticas y lógicas efectuadas por el CPU con anterioridad.

Los cinco *flags* (banderas) son: Desborde “*overflow*” (V), semiacarreo (H), máscara de interrupción (I), negativo (N), cero (Z) y acarreo (C).

Luego de un *reset*, los *flags* tomarán los siguientes valores: V = x; H = x; I = 1; N = x; Z = x; C = x; donde x es un valor indeterminado.

Flag V (Bit de desborde)

Es utilizado en “chequeos” de operaciones aritméticas signadas. El CPU pone en 1 el bit de overflow cuando ocurre un desborde en una operación aritmética con signo (complemento a dos). Las instrucciones de salto condicionales signados como BGT, BGE, BLE, y BLT usan este bit de desborde.

Flag H (semiacarreo)

El bit es “Seteado” si un *carry* (acarreo) ocurre desde el bit 3 al bit 4, se utiliza en operaciones aritméticas BCD.

Flag I (Mascara global de interrupciones)

Cuando está seteado se deshabilita las interrupciones del CPU.

Flag N (Negativo)

Es seteado, si el bit 7 está seteado en el acumulador.

Flag Z (cero)

Este bit es seteado si todos los bits en el acumulador son ceros.

Flag C (acarreo / préstamo)

Seteado si se produce un *carry* o *borrow* durante una operación.

6 Modos de direccionamiento

Los modos de direccionamiento de la CPU proveen la capacidad de acceder a memoria de diferentes maneras. Los **modos de direccionamiento [5], [7]** difieren la manera en que una instrucción obtendrá el dato requerido para su ejecución.

La CPU del MC68HC908 [7] usa siete modos de direccionamiento para acceder a memoria, estos son: inherente, inmediato, extendido, directo, indexado (sin desplazamiento, con desplazamiento de 8 bits, con desplazamiento de 16 bits) y relativo.

En los pequeños microcontroladores MC68HC908, todas las variables del programa y los registros de I/O caben en el área de memoria que va de $0000 a $00FF, donde el modo de direccionamiento usado es el directo.

En los siguientes párrafos se provee una descripción general y ejemplos de los distintos modos de direccionamiento. El término **dirección efectiva** es usado para indicar la dirección de la posición de memoria donde el operando para una instrucción es buscado o almacenado. En el *apéndice B* está disponible la descripción de cada instrucción.

Para cada modo de direccionamiento se explica en detalle una instrucción. Esta explicación describe qué sucede en la CPU durante cada ciclo de reloj del procesador.

6.1. Modo de direccionamiento inmediato

Especifica el valor directamente, no la dirección del valor. El valor está indicado por el símbolo #. Este modo de direccionamiento tiene un solo operando que está contenido en el byte o los bytes que se encuentran en las direcciones de memoria siguientes a la del código de operación, y es usado cuando hay un valor o constante conocido, al momento de escribir el programa, que no cambiará durante la ejecución del programa. Esta es una instrucción de 2 bytes, uno para el código de operación y otro para el byte de dato inmediato.

Ejemplo:
Código de operación: A6
Formato de instrucción en Assembler: LDA #$FF; carga el acumulador con el valor inmediato. En la figura 23 se representa el espacio de memoria y el contenido del registro de este ejemplo.

Secuencia de ejecución:

Dir.	**Contenido**
$EE00	$A6 (1)
$EE01	$FF (2)

Explicación:
(1) La CPU lee el código de operación $A6. Carga el acumulador con el valor inmediato siguiente al código de operación.
(2) La CPU lee el dato inmediato $FF de la posición de memoria $EE01 y lo carga en el acumulador.

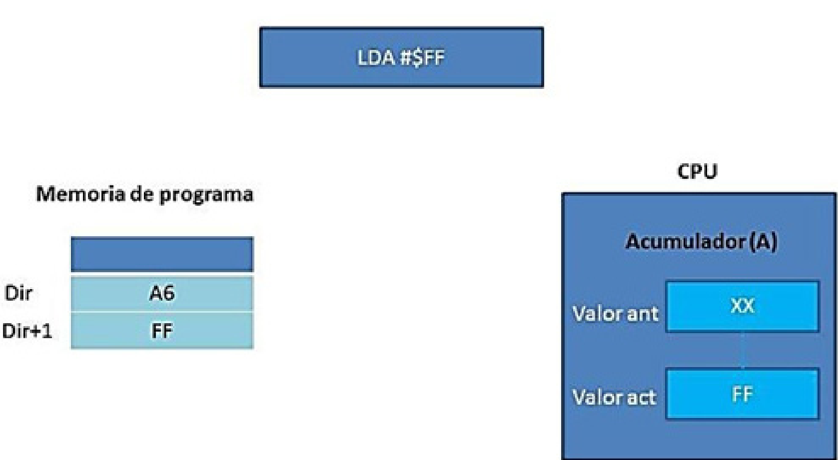


Figura 23. Ejemplo de direccionamiento inmediato.

6.2. Modo de direccionamiento inherente

En el modo de direccionamiento inherente, toda la información requerida para la operación ya es implícitamente conocida por la CPU y no es necesario recuperar un operando exterior desde la memoria. Los operandos (si los hay) son solo los registros de la CPU, o bien valores de datos almacenados en la pila. Esta es una instrucción de un solo byte, debido a que los operandos están en registros internos al procesador.

Ejemplo:
Código de operación: 4F
Formato de instrucción en Assembler: CLRA; limpia el acumulador.

Secuencia de ejecución:

Dir.	**Contenido**
$EE00	$4F (1) y (2)

Explicación:
(1) La CPU lee el código de operación $4F – borrado del acumulador.
(2) La CPU almacena el valor 00 en el acumulador.

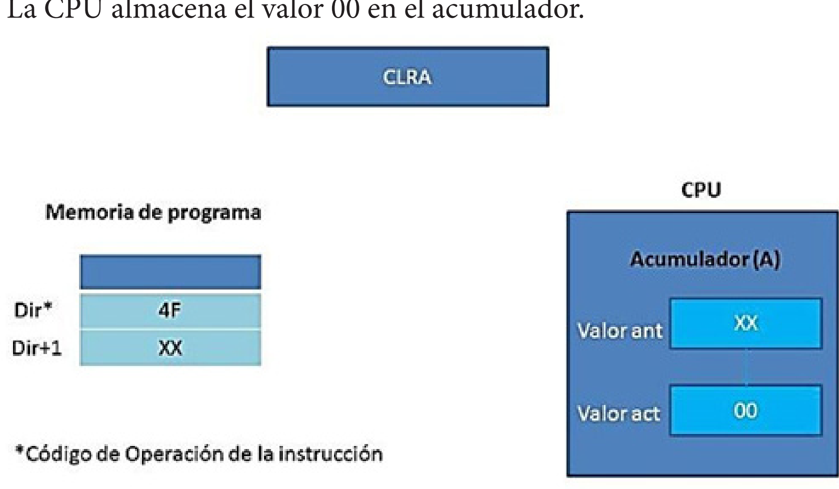


Figura 24. Ejemplo de direccionamiento inherente.

6.3. Modo de direccionamiento directo

El modo de direccionamiento directo, es similar al extendido, excepto que la parte alta de la dirección del operando es de valor $00. De tal manera, que solo es necesario incluir el byte de menos peso de la dirección del operando en la instrucción. Es usado para acceder a los primeros 256 bytes de memoria. Esta área de memoria se denomina **página directa** e incluye a los registros de RAM e I/O del interior del chip. Este modo es eficiente tanto en economía de espacio de memoria como en tiempo de ejecución.

Esta es una instrucción de dos bytes, uno para el código de operación y otro para el byte de menor peso de la dirección del operando.

Ejemplo:
Código de operación: B6
Formato de instrucción en Assembler: LDA $80; carga el acumulador desde una dirección de página directa.

Secuencia de ejecución:

Dir.	**Contenido**
$EE00	$B6 (1)
$EE01	$80 (2) y (3)

Explicación:

(1) La CPU lee el código de operación $B6 - Carga del acumulador usando el modo de direccionamiento directo.

(2) La CPU lee $80 de la posición de memoria $EE01. Este $80 es interpretado como parte baja de una dirección de página directa (va desde $0000 hasta $00FF).

(3) La CPU arma la dirección directa completa agregando 00 en la parte alta, y queda $0080. Esta dirección es colocada en el bus de direcciones, la CPU lee el valor del dato contenido en la posición de memoria $0080 y lo carga en el acumulador (ver figura 25).

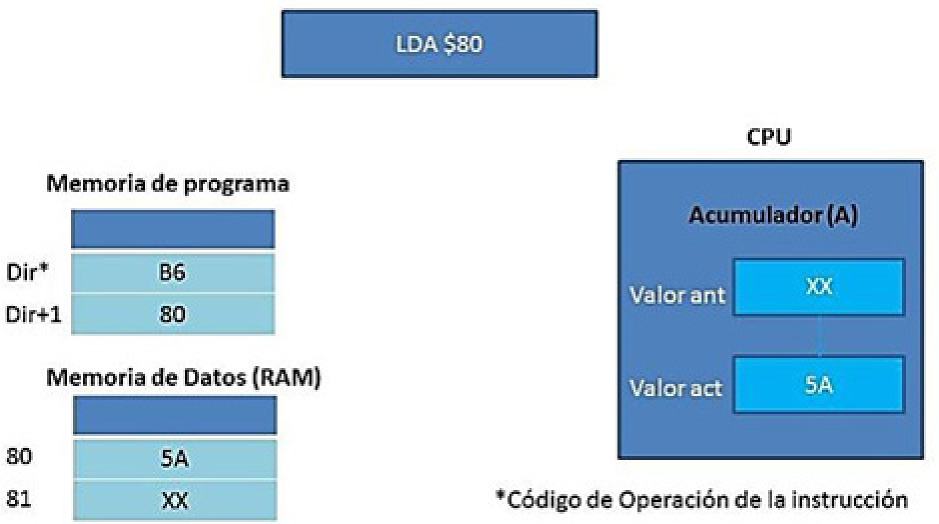


Figura 25. Ejemplo del modo de direccionamiento directo.

6.4. Modo de direccionamiento extendido

En el modo de direccionamiento extendido, la dirección del operando está contenida en los dos bytes siguientes al código de operación. Este modo es usado para acceder a cualquier posición de memoria mayor a $00FF, incluyendo espacio de RAM mayor a $00FF, ROM o FLASH. Esta es una instrucción de 3 bytes, uno para el código de operación y otros dos para la dirección del operando.

Ejemplo:
Código de operación: C6
Formato de instrucción en Assembler: LDA $F800; Carga el acumulador desde una dirección extendida.

Secuencia de ejecución:

Dir	**Contenido**
$EE00	$C6 (1)
$EE01	$F8 (2)
$EE02	$00 (3) y (4)

Explicación:
(1) La CPU lee el código de operación $C6. Carga del acumulador usando el modo de direccionamiento extendido.
(2) La CPU lee $F8 de la posición de memoria $EE01. Este $F8 es interpretado como la parte alta de la dirección.
(3) La CPU lee $00 de la posición de memoria $EE02. Este $00 es interpretado como la mitad de menor peso de una dirección.
(4) La CPU arma la dirección extendida completa $F800 con los dos valores previamente leídos. Esta dirección es colocada en el bus de direcciones para obtener su contenido desde la memoria. Por último se carga dicho contenido en el acumulador.

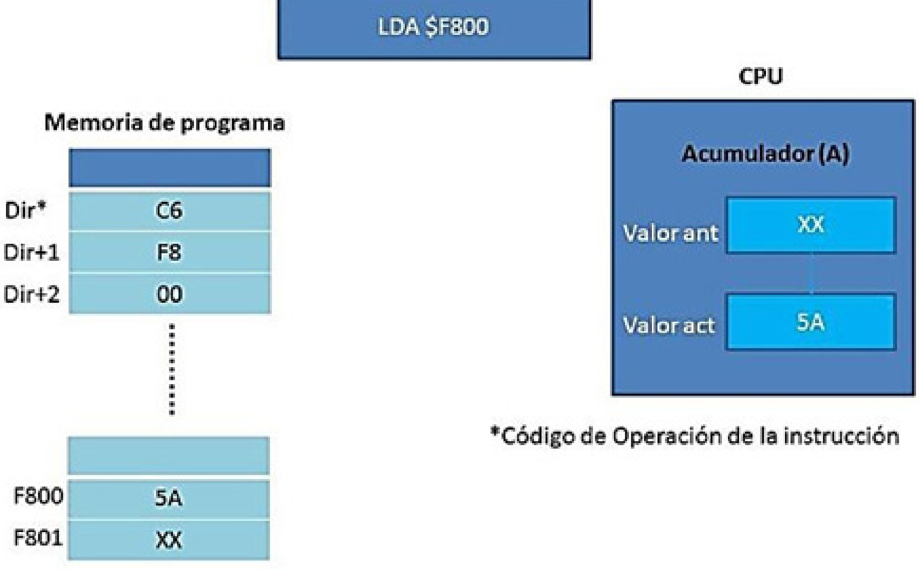


Figura 26. Ejemplo de direccionamiento extendido.

6.5. Modo de direccionamiento indexado

En el modo de direccionamiento indexado la dirección efectiva del operando es variable y depende de dos factores:

1) el contenido actual del registro índice (H : X);
2) el desplazamiento (*offset*) contendido en el o los bytes siguientes al código de operación.

La CPU del MC68HC908 soporta tres tipos de direccionamientos indexados: sin desplazamiento, con desplazamiento de 8 bits y con desplazamiento de 16 bits. Un buen compilador usará el modo de direccionamiento indexado que requiera el menor número de bytes para expresar el desplazamiento.

6.5.1. Direccionamiento indexado "sin desplazamiento (Offset)"

Se especifica el contenido del Registro índice H : X como dirección del operando. En el HC908 el registro índice es de 16 bits, por lo que el compilador Assembler interpretará "H : X" cuando solo vea "X" en la instrucción con direccionamiento indexado.

En el modo de direccionamiento indexado sin desplazamiento, la dirección efectiva del operando para la instrucción está contenida en los 16 bits del registro índice.

Ejemplo:
Código de operación: 7F
Formato de instrucción en Assembler: CLR ,X; limpiar la dirección apuntada por X.

Secuencia de ejecución:

Dir	**Contenido**
$EE00	$7F (1), (2) y (3)

Explicación:
(1) La CPU lee el código de operación $7F – borrar la dirección apuntada por el registro índice.
(2) La CPU arma la dirección completa sumando $0000 al contenido del registro índice de 16 bits (X : H).
(3) La CPU borra el valor del dato contenido en esa posición de memoria.

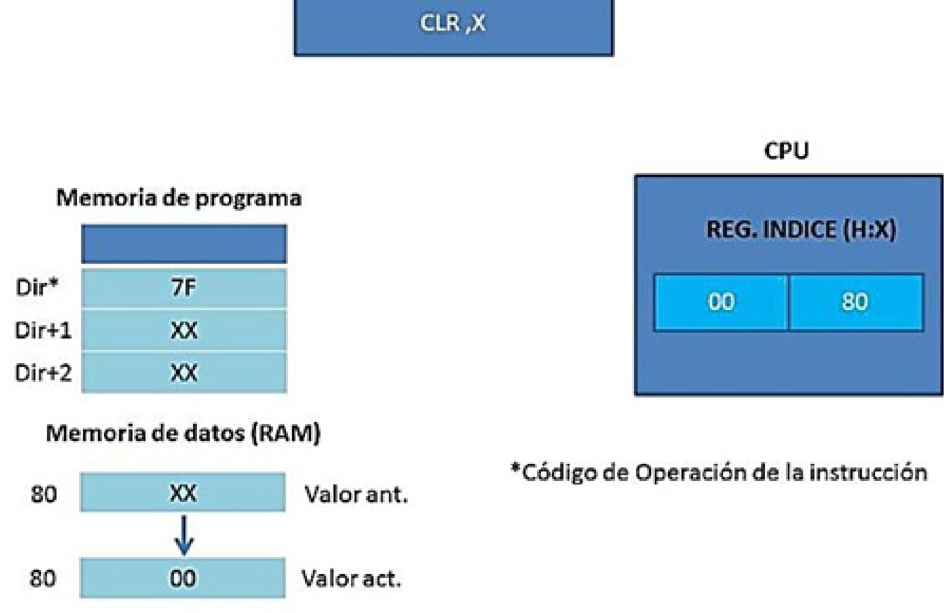


Figura 27. Ejemplo del modo de direccionamiento indexado sin desplazamiento.

6.5.2. Direccionamiento indexado con "desplazamiento (offset) de 8 bits"

En el modo de direccionamiento indexado con desplazamiento de 8 bits, la dirección efectiva es la suma del contenido del registro índice de 16 bits y el byte de desplazamiento siguiente al código de operación. El byte de desplazamiento suministrado en la instrucción es un número entero no signado de 8 bits.

Esta es una instrucción de dos bytes, uno para el código de operación y otro para el byte de desplazamiento. El contenido del registro índice no es alterado.

Ejemplo:
Código de operación: 6F
Formato de instrucción en Assembler: CLR2, X; Limpia la dirección indicada por el décimo primer ítem de la tabla apuntada por X.
Secuencia de ejecución:

Dir	Contenido
$EE00	$6F (1)
$EE01	$2(2), (3) y (4)

Explicación:

(1) La CPU lee el código de operación \$6F – borrado del contenido de la dirección apuntada por el registro índice, usando el modo de direccionamiento indexado con desplazamiento de 8 bits.

(2) La CPU lee \$2 de la posición de memoria \$EE01. Este \$2 es interpretado como un desplazamiento de 8 bits.

(3) La CPU arma la dirección completa sumando el valor antes leído (\$2) al contenido del registro índice de 16 bits (X : H) que es la dirección \$80.

(4) La CPU borra el valor del dato contenido en esa posición de memoria (\$82).

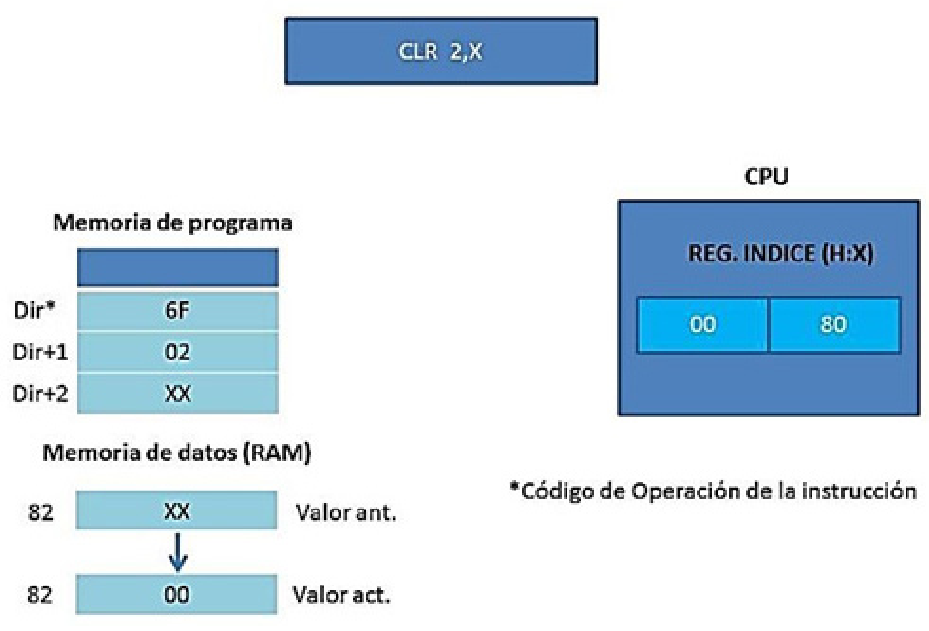


Figura 28. Ejemplo de direccionamiento indexado con 8 bits de *offset.*

6.6. Modo de direccionamiento relativo

El modo de direccionamiento relativo es usado solamente por las instrucciones de bifurcación (saltos condicionales). Las instrucciones de bifurcación, salvo las bifurcaciones en su versión de manipulación de bits, generan 2 bytes de código de máquina: uno para el código de operación y otro para el desplazamiento relativo. Ya que es deseable bifurcar en cualquier sentido, el byte de desplazamiento es un número signado según el convenio de “complemento a dos” con un rango que va desde -128

hasta +127 bytes (respecto de la dirección de la instrucción inmediata posterior a la instrucción de bifurcación). Si la condición de salto es verdadera, el contenido de los 8 bits, del byte con signo, siguiente al código de operación es sumado al contenido del contador de programa para formar la dirección de bifurcación efectiva, es decir:

Contador de programa = Contador de programa + 8 bit *offset* signado

En el caso de que la condición dé falso, el procesador continua con la ejecución de la instrucción inmediata posterior a la instrucción de bifurcación (el Contador de programa no es afectado). Un programador especifica el destino de una bifurcación como una dirección absoluta (o etiqueta que hace referencia a una dirección absoluta). El compilador calculará el desplazamiento relativo de 8 bits con signo, que es colocado en memoria luego del código de operación del salto condicional.

El direccionamiento relativo tiene básicamente las siguientes características:

- Se usa solo para instrucciones de salto (branch).
- El PC es incrementado en 2 desde la dirección del código de operación (debido a la precarga).
- El salto puede tener 8 bit de offset. El rango es de -128 + 127 desde PC
- La dirección efectiva (EA) se obtiene como el contenido del PC + el desplazamiento (8 bits de offset).
- El desplazamiento calculado por el compilador es desplazamiento = EA- PC

Ejemplo:

Código de operación: 27 AE

Formato de instrucción en Assembler: BEQ $F800; salta a $F800 si Z = 1 (si el resultado de la operación anterior es igual a cero).

Secuencia de ejecución:

Dir	**Contenido**
$F850	$27 (1)
$F851	$AE (2) y (3)

Explicación:

(1) La CPU lee el código de operación $27 - Bifurcar si Z = 1. El bit Z del registro de código de condición será 1 si el resultado de la operación aritmética o lógica previa fue cero.

(2) La CPU lee $AE de la posición de memoria $F851. Este $AE es interpretado como el valor de desplazamiento relativo. Después de este ciclo el contador de programa apunta al primer byte de la próxima instrucción ($F852).

(3) Si el bit Z está en cero, nada sucede en este ciclo y el programa debe continuar con la próxima instrucción. Si el bit Z está en 1, la CPU armará la direc-

ción completa sumando el desplazamiento signado antes leído ($AE) al contenido del registro contador de programa para obtener la dirección destino de la bifurcación. Esto provoca que la ejecución del programa continúe desde una nueva dirección ($F800).

Figura 29. Ejemplo de direccionamiento relativo

Memoria

7.1. Introducción

La CPU08 puede direccionar 64 Kbytes de espacio de memoria [3]. El mapa de memoria que se encuentra en la figura 30 muestra:

- 4096 bytes de FLASH de programa, a partir de la dirección $EE00, para MC68HC908QT4 y MC68HC908QY4 [8]
- 128 bytes de RAM que van de la dirección $80 a la $FF
- 6 bytes de vectores de *reset* definidos al final de la memoria FLASH a partir de la dirección $FFFA

7.2. Ubicaciones de memoria no implementada

Acceder a ubicaciones no implementadas puede tener respuestas impredecibles en la operación del MCU, como por ejemplo la ejecución de un *reset*, o un error del sistema.

7.3. Ubicaciones de memoria reservada

Acceder a ubicaciones de memoria reservadas puede tener efectos impredecibles sobre la operación del MCU. En la Figura 30 la memoria reservada está indicada con la palabra reservada o con la letra R.

7.4. Sección Entrada/Salida (I/O)

Las direcciones $0000–$003F contienen registros de control, estado y datos de los módulos I/O del microcontrolador [8].

Figura 30. Mapa de memoria de la línea HC908QT4/QY4

7.5. Memoria de acceso aleatorio (RAM)

Las direcciones $0080–$00FF corresponden a la ubicación de la memoria RAM [3], [4]. La ubicación de la Pila de RAM es programable. El registro puntero de pila de 16-bit permite a la pila ser ubicada en cualquier espacio de memoria de los 64 Kbytes, por defecto dicho registro apunta a la dirección $FF.

Es importante comprender que para una operación correcta el puntero de pila debe apuntar solo a las direcciones de RAM, debido a que estas direcciones se pueden modificar en tiempo de ejecución. Antes de procesar una interrupción, la CPU usa 5 bytes de la pila para guardar el contenido de los registros de la CPU (acumulador, registro índice, CCR y contador de programa).

Durante una llamada a subrutina, la CPU usa 2 bytes de la pila para almacenar la dirección de retorno de la subrutina. El contenido del puntero de Pila se decrementa durante el ingreso de datos y se incrementa al extraer datos.

Se debe tener cuidado al usar subrutinas anidadas, debido a que se estará guardando de manera continua la dirección de retorno de cada subrutina en ejecución. La CPU puede sobreescribir los datos en la RAM durante una subrutina o durante la operación de apilado en una interrupción, pero se debe tener cuidado al guardar datos o modificar datos en la pila dentro de una subrutina, pues se puede sobreescribir la dirección de retorno.

7.6. Memoria FLASH (FLASH)

La memoria FLASH puede ser leída, programada y borrada eléctricamente. La memoria flash consiste de un arreglo de 4096 bytes. Los rangos de direcciones para la memoria de programa y vectores de *reset* son:

- Desde la dirección $EE00 hasta la $FDFF; memoria de programa, 4.096 bytes: en los microcontroladores MC68HC908QY4 y MC68HC908QT4 [8].
- Desde la dirección $F800 hasta la $FDFF; memoria de programa, 1.536 bytes: en los Microcontroladores MC68HC908QY2, MC68HC908QT2, MC68HC908QY1 y MC68HC908QT1
- Desde la dirección $FFD0 hasta la $FFFF; vectores de interrupción del usuario incluidos los vectores de reset, 48 bytes.

8 Programación a bajo nivel (Assembler)

8.1. Introducción

El lenguaje de bajo nivel (Assembler o ensamblador) se considera extremadamente importante dentro de la enseñanza de Organización y arquitectura de computadoras, principalmente para la comprensión del funcionamiento interno del procesador en cuanto al movimiento de datos, ejecución de instrucciones, estructura de registros, etcétera.

Si bien actualmente es mucho más eficiente programar en lenguajes de alto nivel, la programación en Assembler es muy utilizada en el diseño de sistemas operativos. Buena parte de ellos está escrita en este lenguaje por la necesidad de interacción entre el HW y el SO. Esto significa que es necesario comprender la programación y la ejecución de instrucciones para entender el funcionamiento de los sistemas operativos.

El objetivo no es formar expertos programadores en Assembler, pero es necesario comprender el funcionamiento de un programa y aprender a programar operaciones básicas como movimiento de datos, comparación entre datos y operaciones aritmético-lógicas.

En las siguientes subsecciones se describirá lo más importante de la programación a bajo nivel, tal como la descripción de instrucciones, el simulador y la programación paso a paso de una aplicación en lenguajes de bajo nivel.

8.2. El lenguaje de bajo nivel

El código Assembler requiere escribir varias instrucciones para la realización de una acción simple, es por eso que tiene poca aceptación entre los programadores y más aún cuando el programador sabe programar en un lenguaje de alto nivel. En este capítulo se describirán los pasos a seguir para el planteo de una aplicación y su posterior programación en lenguaje de bajo nivel.

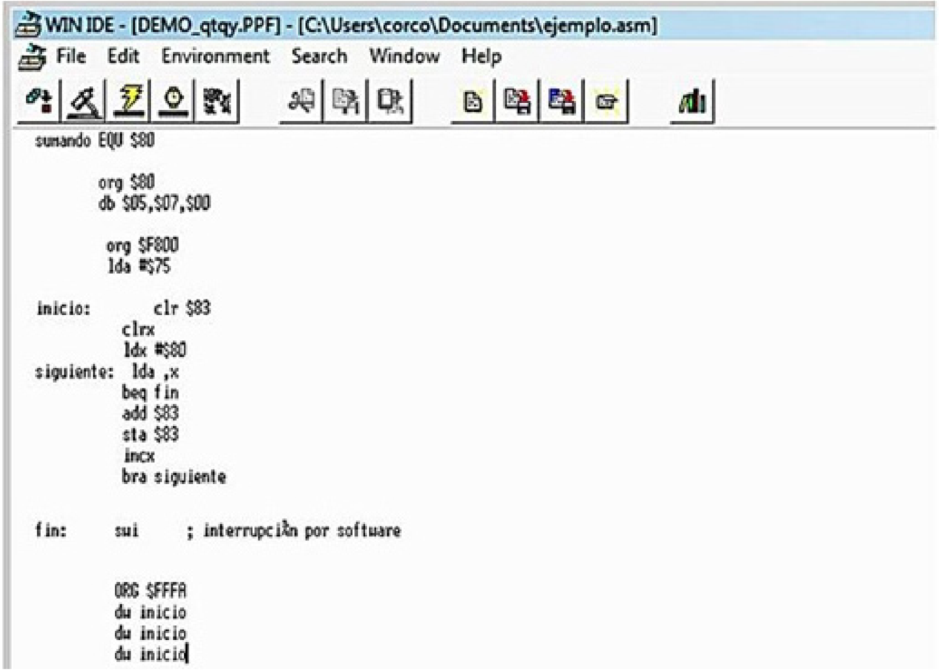
8.3. Software de programación y simulación

Para la simulación se utiliza el software WinIDE, que es un entorno de desarrollo integrado por varios programas, de los cuales se usa el compilador y el simulador [5].

La pantalla principal del WinIDE se muestra en la figura siguiente, donde se puede observar un programa ejemplo escrito en Assembler. En este ejemplo se distinguen las etiquetas inicio, siguiente y fin. Dichas etiquetas se deben escribir a partir de la primera posición de la hoja en el lado izquierdo, este espacio está reservado para etiquetas y no se deben escribir instrucciones para evitar errores de compilación.

El código de programa se debe escribir dejando una o dos tabulaciones a la derecha, tal cual se observa en la figura 31.

Para escribir comentarios se utiliza el carácter ";". Esto es importante para dejar bien documentado cada programa. También se debe tener en cuenta que los comentarios no deben ser muy extensos, de otra forma el compilador emitirá un error. Si es necesario un comentario extenso, lo mejor es continuar en el siguiente renglón.



Fgura 31. Entorno de programación WinIDE

En la figura 32 se muestran las diferentes herramientas de software del entorno WinIDE, el compilador que se utiliza para verificar que el código no tiene errores y se ubica en la primera posición del menú.

Una vez que se obtiene una compilación exitosa se puede pasar a la simulación presionando el quinto botón de la figura, que permite realizar simulación pura en la PC.

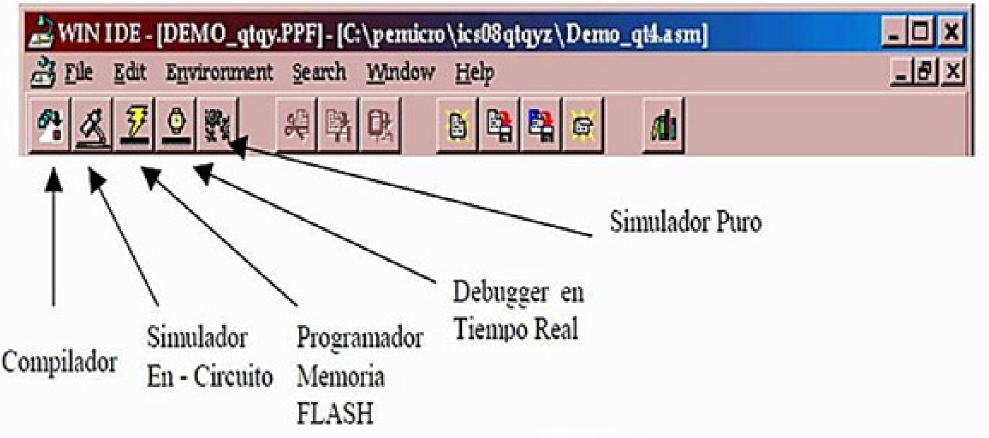


Figura 32. Herramientas de software del entorno WinIDE

Al terminar de escribir el programa en Assembler se presiona la opción de compilador y si el código no tiene errores de sintaxis se generará una ventana como la que se muestra en la figura 33, donde en Status aparecerá la frase *Successful assembly* y aparecerá un tilde de color verde a la izquierda de ok. Al realizar la compilación se generará un archivo de código binario con extensión "s.19", donde se almacena el programa en formato binario listo para ser guardado en la memoria del sistema de procesamiento.

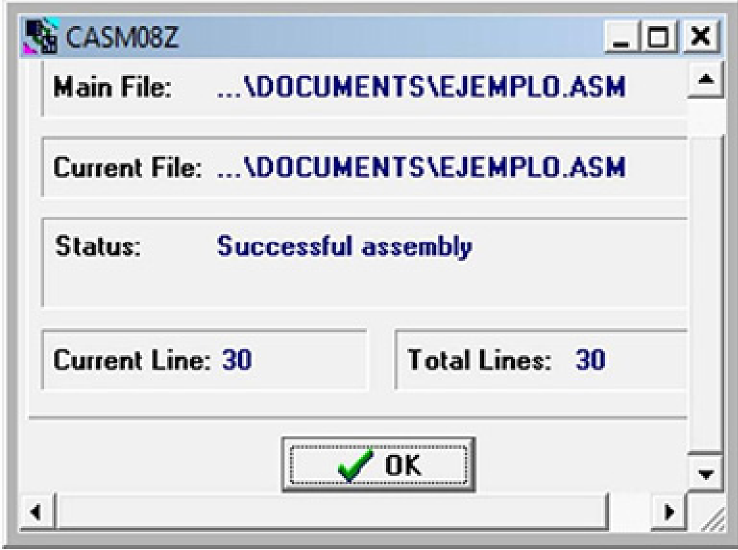


Figura 33. Compilación exitosa

En esta instancia se sabe que el código no tiene errores de sintaxis, pero nos interesa saber si el programa hace lo que nosotros planificamos, para lo cual se usa el simulador. Al presionar la opción de simulador puro aparece la ventana que se muestra en la figura 34, aquí solo se debe presionar la opción *simulation only*. Al realizar la simulación por primera vez, aparecerá una ventana que permite elegir sobre que microcontrolador se desea ejecutar el programa, se recomienda seleccionar el HC809qy4, por ser el que más espacio de memoria posee y su memoria Flash inicia en la dirección $EE00.

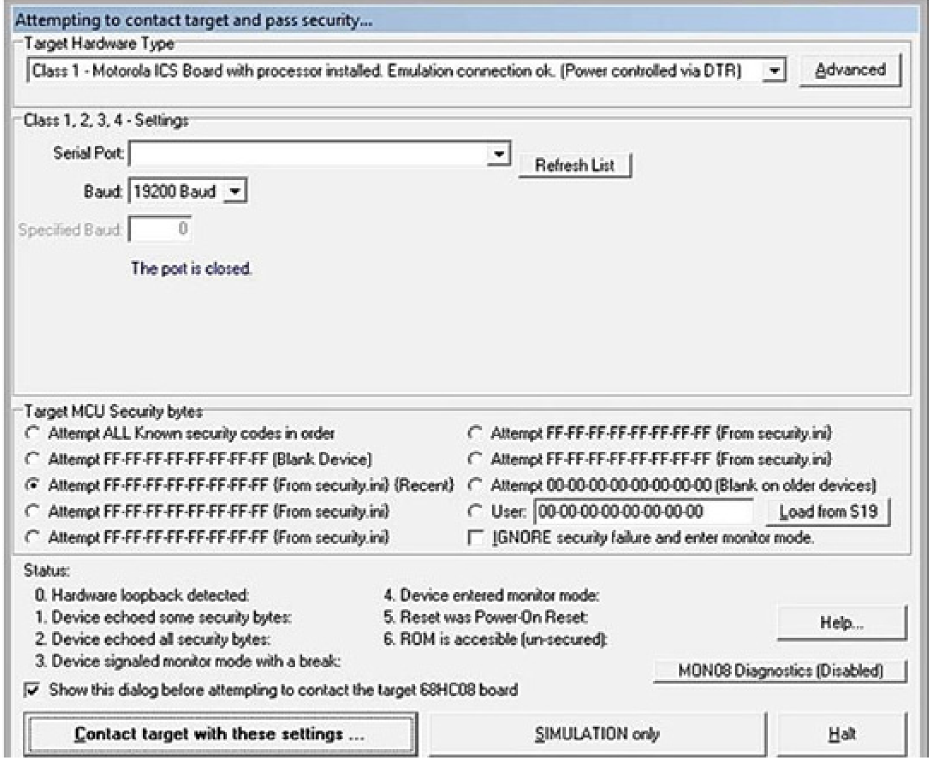


Figura 34. Ventana de selección de las características de la simulación

Al abrirse la ventana de simulación se observarán las características que se muestran en la figura 35. En la parte superior izquierda se mostrará el estado de los registros del procesador en cada instancia de la simulación. En la parte inferior derecha se muestra una ventana que contiene tres columnas, la primera indica la dirección de memoria, la segunda el programa en hexadecimal y la tercera el programa en Assembler. La flecha azul que se observa en esta ventana indica cuál es la instrucción que se ejecutará en cada paso de la simulación.

Cuando se ejecuta un programa es muy importante poder conocer el contenido de la memoria de datos, para esto se tiene en la parte superior derecha una ventana que muestra el contenido de la memoria a partir de la dirección $80, que es donde se encuentran los datos utilizados por el programa.

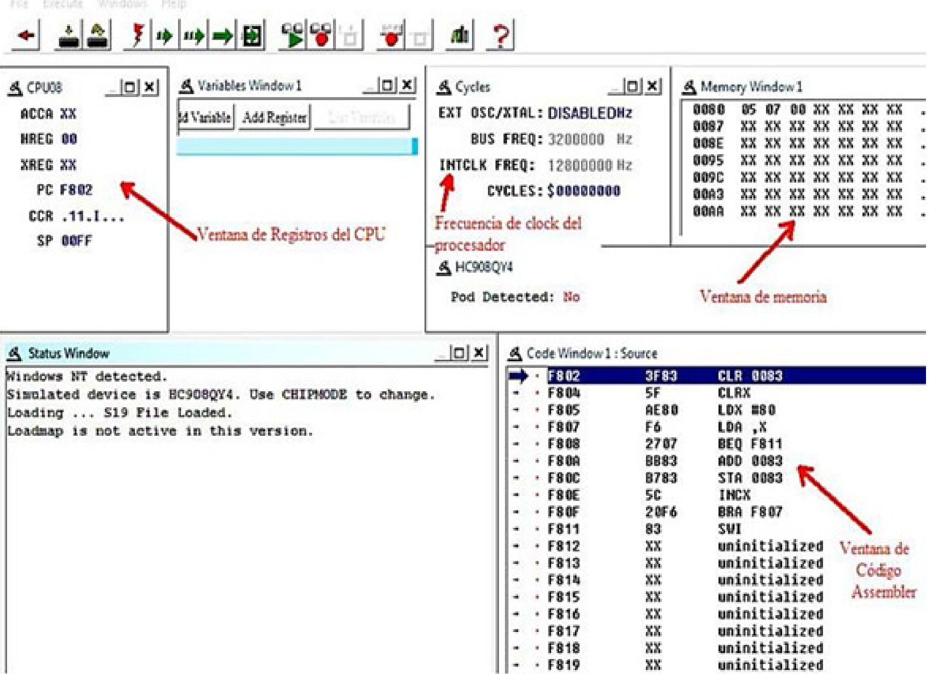


Figura 35. Ventana de simulación

Para realizar la simulación paso a paso, se deberá presionar el botón , cada vez que se presione se ejecutará una nueva instrucción y se podrán visualizar los cambios en los registros y en la memoria de datos.

8.4. Planificación y programación de una aplicación en Assembler

Para realizar una aplicación en Assembler, en primer lugar se debe planificar la aplicación mediante la descripción ordenada de cada una de sus partes. Esto se efectúa mediante el diagrama de flujo, el cual no debe contener nombres de registros ni direcciones para que la descripción sea bien genérica e independiente del hardware, como se muestra en la figura 36. Luego, se procederá a la programación en Assembler, siguiendo el orden indicado en el diagrama.

8.4.1. Diagrama de flujo

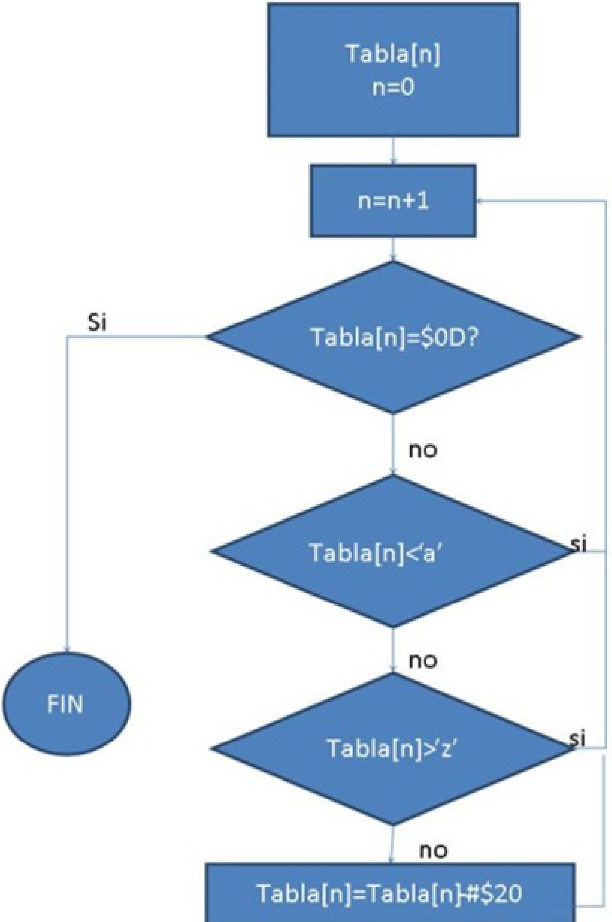


Figura 36. Diagrama de flujo del ejemplo.

El diagrama de flujo de la figura anterior representa la planificación y programación de una aplicación que lee caracteres ascii desde una tabla en memoria que comienza en la dirección $90 y termina con el carácter $0D. La aplicación deberá detectar las letras minúsculas y reemplazarlas por mayúsculas, esto se hace restando el número $20 en hexadecimal a la letra mayúscula, debido a que las minúsculas comienzan a partir del $41 y las mayúsculas a partir del $61, lo que significa que entre una minúscula y una mayúscula hay $20 caracteres. Por último, cuando se detecte el carácter $0D significará que se llegó al último elemento de la tabla y el programa finalizará.

8.4.2. Programación de la aplicación

```
     org $90 ; inicio de la memoria de datos (RAM)
     db  'A','f','B','c',$0D  ; tabla de caracteres ascii a partir de
;la dirección $90
     ORG $EE00 ; inicio de la memoria de programa
     ldx #$8F
ciclo:    ;
     incx
     lda ,x
     CBEQA  #$0D,fin
     cmp #$60
     ble ciclo
     cmp #$7B
     bhe ciclo
     sbc #$20
     sta ,x
     bra ciclo
fin:swi ; esta instrucción se pone para indicar el fin del programa

org $fffa ; dirección de inicio de los vectores de reset
dw $ee00  ;SWI
dw $ee00  ;IRQ
dw $ee00  ;RESET
```

8.4.3. Descripción del código

A partir de la dirección 90 se almacenan los datos de la tabla de forma consecutiva

org $90
db $6D,$41,$72,$43,$65,$0D; tabla de caracteres ascii

A partir de la dirección EE00 se almacena el programa de la aplicación

ORG $EE00

Con la instrucción LDX cargo la dirección $8F en el registro índice. Se debe aclarar que el registro índice es ideal para usar como puntero para recorrer una tabla de datos.

ldx #$8F

La etiqueta **ciclo** representa un bucle que se va a ejecutar hasta encontrar el caracter 0D en la tabla de caracteres

ciclo:

Se incrementa el puntero para apuntar a la dirección del siguiente caracter en la tabla

incx

Se guarda el caracter apuntado por x en el acumulador

lda ,x

Se compara el caracter del acumulador con el $0D para saber si se llegó al final de la tabla. CBEQA es un salto condicional y en caso en que el contenido del acumulador sea igual a 0D, entonces se producirá un salto al final del programa. Si el contenido del acumulador es distinto de 0D, entonces se sigue ejecutando la siguiente instrucción.

CBEQA #$0D,fin

La siguiente instrucción compara el contenido del acumulador con el caracter "a" minúscula.

cmp #'a'

blo analiza si la comparación anterior dio que el contenido del acumulador es menor que 'a'. Si es menor significa que no es una letra minúscula y entonces se produce un salto a la etiqueta ciclo. En el caso donde el contenido del acumulador sea igual o mayor que 'a', entonces se ejecutará la siguiente instrucción

blo ciclo

Ahora se realiza una comparación entre el contenido del acumulador y la z minúscula

cmp#'z'

bhi analiza si el contenido del acumulador es mayo a la letra 'z'. En el caso de que sea mayor significa que el caracter del acumulador está no es una letra minúscula y se produce un salto a ciclo para analizar el siguiente caracter. Si el contenido del acumulador es menor a 'z', significa que la letra es una minúscula y se ejecutará la siguiente instrucción

bhi ciclo

La siguiente instrucción le resta 20 al contenido del acumulador, esto significa que se está convirtiendo la minúscula que está en el acumulado en mayúscula, debido a que las mayúsculas se encuentran 20 posiciones debajo de las minúsculas. El resultado de la resta se guarda en el acumulador.

sbc #$20

La siguiente instrucción guarda el contenido del acumulador en la posición de memoria a la que apunta x. En otras palabras se está reemplazando el carácter de la tabla por una mayúscula.

sta , x

La instrucción **bra** realiza un salto obligado a la etiqueta ciclo para analizar el siguiente caracter de la tabla.

bra ciclo

La etiqueta fin indica que cuando termine de ejecutarse la aplicación se producirá una interrupción por software que reiniciará el programa.

fin: swi

APÉNDICE A Descripción de arquitecturas clásicas

A.1. Arquitectura Harvard

La arquitectura Harvard almacena instrucciones y datos en memorias separadas como muestra el diagrama de la figura 37. Muchas arquitecturas de procesadores contienen la estructura Harvard.

La instrucción se trae a la CPU en un solo acceso a la memoria de programa, mientras tanto el bus de datos está libre y puede accederse a través de él a los datos que se necesitan para ejecutar la instrucción anterior a la que se está trayendo de la memoria de programa en ese momento.

Al tener dos buses separados, el bus de instrucciones es más ancho que el bus de datos. Esto permite que las instrucciones se codifiquen en palabras de más de 8 bits. Concretamente, la codificación se realiza de acuerdo con los requisitos de la arquitectura.

A su vez, al codificarse en una sola palabra, cada instrucción se trae a la CPU en un único ciclo de instrucción, (equivalente a cuatro ciclos de reloj), mediante el bus de memoria de programa. Esta instrucción contiene toda la información requerida y se ejecuta en un solo ciclo.

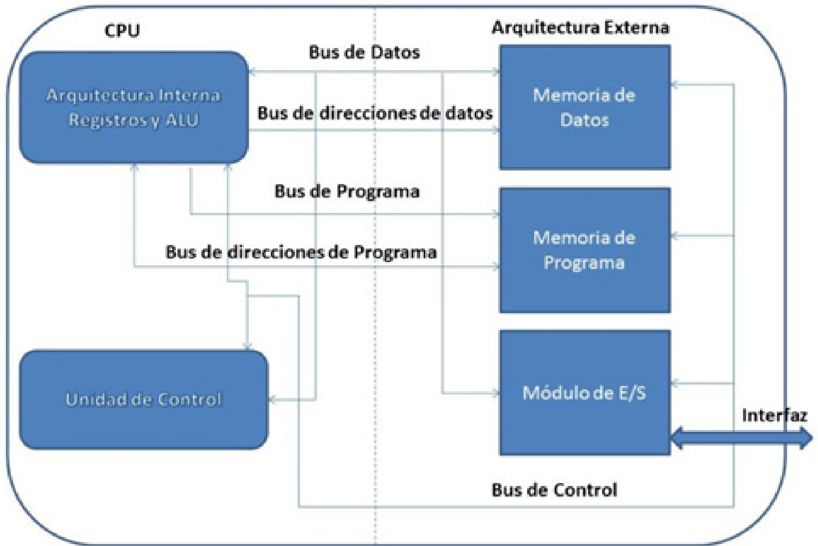


Figura 37. Diagrama en bloques de la arquitectura Harvard

Durante el ciclo de ejecución de la instrucción el proceso es el siguiente:

La instrucción traída durante el ciclo de instrucción anterior se almacena en el registro de instrucciones (IR) durante el ciclo Q1. La instrucción es decodificada y ejecutada durante los ciclos Q2, Q3 y Q4. Si la instrucción conlleva un acceso a la memoria de datos para lectura, este acceso se realiza durante el ciclo Q2. Si la instrucción conlleva un acceso a la memoria de datos para escritura, este acceso se realiza durante el ciclo Q4. Esta arquitectura suele utilizarse en DSPs (Procesadores de señales digitales) para procesamiento de audio y video.

A.2. Arquitectura Von Neumann

La característica más destacada de la arquitectura Von Neumann es que almacena datos e instrucciones en una misma memoria, como muestra la figura 38.

Requiere acceso (o varios accesos) a memoria para traer la instrucción. Si esta instrucción maneja datos de memoria, se debe(n) realizar otro(s) acceso(s) para traer, operar y volver a almacenar los datos. El bus se congestiona con tanto acceso. En la arquitectura Von Neumann se necesitan habitualmente varios paquetes de 8 bits para codificar una instrucción. Así, por ejemplo, un microcontrolador con 4 Kbytes de memoria de programa podría almacenar 2K instrucciones aproximadamente (a una media de 2 bytes por instrucción, aunque depende de la aplicación).

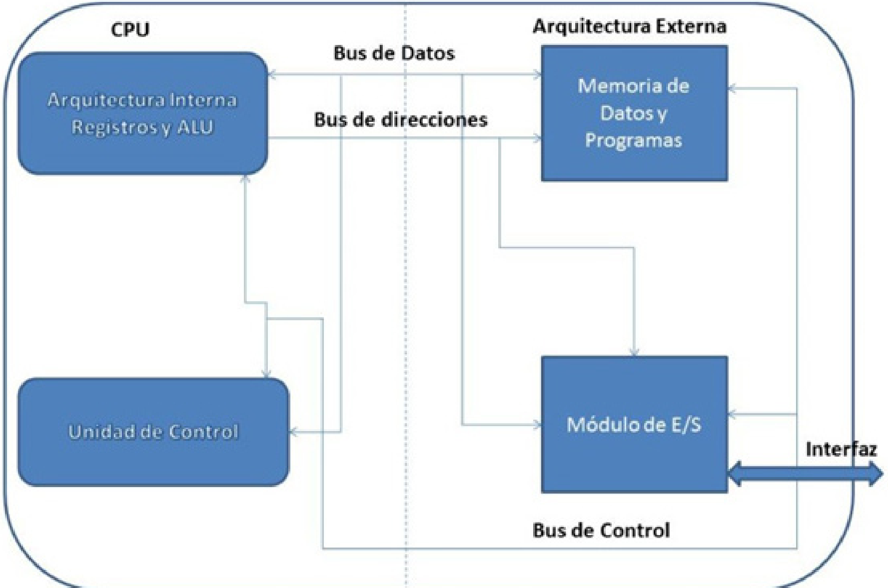


Figura 38. Diagrama en bloques de la arquitectura Von Neumann

El canal de transmisión de los datos entre CPU y memoria genera un cuello de botella para el rendimiento del procesador. En la mayoría de las computadoras modernas, la velocidad de comunicación entre la memoria y la CPU es más baja que la velocidad a la que puede trabajar esta última, reduciendo el rendimiento del procesador y limitando seriamente la velocidad de proceso eficaz, sobre todo cuando se necesitan procesar grandes cantidades de datos. La CPU se ve forzada a esperar continuamente a que lleguen los datos necesarios desde o hacia la memoria.

La velocidad de procesamiento y la cantidad de memoria han aumentado mucho más rápidamente que el rendimiento de transferencia entre ellos, lo que ha agravado el problema del cuello de botella.

A.3. Arquitectura Harvard vs. arquitectura Von Neumann

Von Neumann:

- Permite almacenar datos e instrucciones en el mismo módulo de memoria.
- Más flexible y fácil de implementar.
- Adecuado para muchos procesadores de propósito general.

Harvard:

- Usa módulos de memoria separados para almacenar instrucciones y datos.
- Es fácilmente implementable el pipeline.
- Alto rendimiento de memoria
- Ideal para DSP (Procesadores digitales de señales).
- El tiempo de acceso se mejora respecto de la arquitectura von Neumann, en la que programa y datos se traen a la CPU usando el mismo bus.

A.4. CISC VS RISC

Hay otras formas de diferenciar arquitecturas de computadoras. Los modelos de arquitecturas son conocidos como RISC (Computadora con set de instrucciones reducido) y CISC (Computadora con set de instrucciones complejas). En la tabla siguiente se indican las diferencias entre estos dos modelos.

Tabla 6. Modelo CISC vs. modelo RISC

Característica	RISC	CISC
Tamaño de instrucción	Una palabra	1 a 54 bytes
Tiempo de ejecución	1 ciclo d e clock	De 1 a 100 ciclos
Modos de direccionamiento	pocos	muchos
Tamaño del set instrucciones	Grande	Pequeño

Durante los años 80 se desató una gran controversia sobre las ventajas y desventajas de cada estilo de arquitectura. Como resultado, RISC salió vencedor. De cualquier manera la arquitectura CISC ha sobrevivido, la familia Intel's x86, ha adoptado muchas ideas desde el campo del estilo RISC para mantener el buen desempeño.

Características del set de instrucciones

El conjunto de instrucciones del HC08 se puede clasificar en las siguientes categorías [7] y [8]:

- Movimiento de Datos
- Aritméticas
- Lógicas
- Manipulación de datos
- Manipulación de bits
- Control del programa
- Operaciones BCD
- Especiales

Previamente a la descripción de los distintos tipos de instrucciones debemos indicar el significado de algunos símbolos utilizados para la programación en Assembler.

- El símbolo **!** indica que el número es decimal. Este número será trasladado a un valor binario antes de ser almacenado en memoria para ser usado por la CPU.
- El símbolo **$** precediendo a un número indica que el número es hexadecimal; por ejemplo $24 es 24 (base 16) en hexadecimal o el 36 (base 10) en decimal.
- El símbolo # indica que lo que sigue es un operando y este es buscado en la posición de memoria siguiente a la del código de operación. Se puede usar una variedad de símbolos y expresiones siguiendo al caracter #. Todo número que se utilice dentro de un programa que no esté precedido por el símbolo # será interpretado como una dirección por el procesador.

Tabla 7. Símbolos en Assembler

Prefijo	Indica que el valor siguiente es:
!	Decimal
$	Hexadecimal
@	Octal
%	Binario
‘ (apóstrofe)	Caracter ASCII

Nota: Ya que no todos los ensambladores usan las mismas reglas de sintaxis ni los mismos caracteres especiales, es necesario referirse a la documentación del ensamblador que se deseé usar para entender que representa cada símbolo, en la tabla 7 se muestran los símbolos usados en los microcontroladores de Freescale.

B.1. Movimiento de datos

Las instrucciones de movimiento de datos se pueden dividir a su vez en:

- Instrucciones de carga de registros del CPU
  - LDA #$80; carga el número 80 en HEXA en el acumulador.
  - LDA $F0; carga el contenido de la dirección F0 en el acumulador.
  - LDX #$80; guarda el número 80 en HEXA en el registro índice (este número será interpretado por las instrucciones que usen el registro índice como una dirección).
  - LDA, X; guarda el contenido de la dirección a la que apunta el registro índice en el acumulador.
- Almacenamiento de registros del CPU
  - STA $80; guarda el contenido del acumulador en la dirección 80.
  - STX $80; guarda el contenido del registro índice en la dirección 80.
- Operaciones con la pila
  - PSHA; guarda el contenido del acumulador en la pila.
  - PULA; guarda el último elemento ingresado en la pila en el acumulador.
- Movimiento de datos registro a registro.
  - TAP; transfiere el contenido del acumulador al CCR.
  - TPA; transfiere el contenido del CCR al acumulador.
- Movimiento de datos memoria a memoria
  - MOV $80, $90; mueve el contenido de la dirección 80 a la dirección 90.

El agregado de instrucciones que involucran al nuevo registro concatenado H : X de 16 bits como LDHX, STHX otorgan gran flexibilidad en el manejo de tablas y rutinas de acceso indexado, ahorrando código y aumentando la velocidad de ejecución de las mismas.

Además, se puede apreciar que por cada tipo de instrucción, se agrega un nuevo modo de direccionamiento, basado en el uso del *Stack Pointer* “SP” (puntero de pila) como “segundo registro índice”, lo que facilita el uso de lenguajes de alto nivel como el “C” y otros.

Las instrucciones PUSH y PULL permiten resguardar y rescatar el contenido del acumulador (ACC) y del puntero índice H : X en memoria RAM, ante subrutinas e interrupciones al programa (externas/internas), en forma más rápida y transparente.

Las instrucciones "MOV" en sus diferentes versiones facilitan el movimiento de datos sin afectar los registros del CPU, de esta forma se consiguen operaciones más rápidas y algoritmos más sencillos. Estas instrucciones son útiles en rutinas de transmisión y recepción de datos en las comunicaciones series asincrónicas SCI (UART) de los distintos MCUs de la familia, o bien en movimientos de datos de una tabla a otra.

B.2. Aritméticas

Las Instrucciones aritméticas se pueden clasificar en:

- Instrucciones de adición
  - ADD #$80; suma el contenido del acumulador con el número 80.
  - ADD $90; suma el contenido de la dirección 90 con el acumulador
- Instrucciones de sustracción
  - SUB #$80; resta al contenido del acumulador el número 80. El resultado se guarda en el acumulador.
  - SUB $90; resta al contenido de la dirección del acumulador el contenido de la dirección 90.
- Instrucciones de multiplicación
  - Para usar la instrucción de multiplicación se deben guardar previamente los números a multiplicar en A y X.
    LDA #$80.
    LDX #$2.
    MUL; realiza la multiplicación de 2 x 80 en HEXA.
- Instrucciones de división
  - LDA #$80.
    LDX #$2

DIV; realiza la división entre el contenido del acumulador y el registro índice.

- Instrucciones de complemento y negación
  - COM $80; aplica complemento a 1 al contenido de la dirección 80.
  - NEG $80; cambia el signo del contenido de la dirección 80 en complemento a 2.
- Instrucciones de comparación

  - CMP $80; compara el contenido del acumulador con el contenido de la dirección ;80 (la comparación se hace mediante la resta entre ambos valores).
- Otras instrucciones:
  - Clear
    - CLR $80; borra el contenido de la dirección 80.
  - Chequeo de cero o negativo
    - TST $80; verifica si el contenido de la dirección 80 es negativo o cero.
  - Reserva de espacio en pila
    - AIS #$80; suma el número 80 al contenido del puntero de pila.
  - Reserva de espacio en registro índice
    - AIX #$80; suma el número 80 al contenido del registro índice.

La familia HC08 contiene instrucciones de multiplicación y de división. La instrucción de multiplicación en el CPU08, es del tipo "No signado" (sin signo) de 8 x 8 bits. Se ejecuta en 5 ciclos de *Clock*. En los registros "A" y "X" se cargan los valores a multiplicar, el resultado de la operación se obtiene en los mismos registros "A" y "X", en "A" se encontrará la parte menos significativa del resultado, mientras que en "X" se encontrará la parte más significativa del resultado.

La instrucción división en el CPU08 es del tipo "No signado" (sin signo) de 16/8 bits. En el registro "H" se carga la parte más significativa del valor a dividir, en el registro "A" se carga la parte menos significativa de dicho valor, mientras el divisor se carga en el registro "X".

El resultado de la operación se carga en el registro "A", mientras que el resto o remanente se carga en el "H". Si el resultado de la operación es mayor que "$ FF", entonces se activará el *flag* de "CARRY" (C) en el CCR y el valor en el registro "H" será indeterminado.

Instrucción MUL

– X contendrá el MSB (bit más significativo) del producto.
– A contendrá el LSB (bit menos significativo) del producto.

Instrucción DIV:

– H es el MSB del dividendo.
– A es el LSB del dividendo.
– X no es afectado.

Instrucción AIS

AIS puede usarse para un rápido alojamiento o desalojo de espacio de la pila.
- Variables temporales.
- Procesos en “trama”.

Ejemplo:

```
SUB1        AIS   #-6      ; Aloja 6 bytes en la pila.
            AIS   #6       ; Desaloja 6 bytes de la pila.
            RTS
```

Instrucción AIX

AIX puede usarse:
- Eficiente incremento o decremento del registro H : X
  – La instrucción INCX / DECX solo afecta al registro X.
  – La instrucción INCX / DECX afecta el CCR, AIX no lo afecta.
- Bucles (loops) alrededor de un bloque de memoria
  – Direccionamiento indexado con post incremento solo para incrementos.
- Solamente disponible para instrucciones MOV y CBEQ.

La instrucción AIX permite “adicionar” en forma inmediata un número signado (positivo o negativo) al registro índice H : X, de esta forma pueden lograrse manejos de tablas más eficientes, búsquedas ascendentes o descendentes a partir de un punto, “saltos” discretos mayores a “1” en una tabla, tanto positivos como negativos. La instrucción AIX no afecta el CCR (registro de código de condiciones), y permite incrementar/decrementar al registro H : X en forma amplia (16 bits), y no reducida como las instrucciones INCX / DECX que solo afectan al registro “X”.

Ejemplo: En este ejemplo se calculan 8 bit de suma acumulativa de una tabla de 512 datos.

```
ORG    $0080       ; inicio de memoria de datos
TABLA          RMB    512    ; Tabla de datos inicia en la dirección 80 (reserva
512 bytes para ; la tabla)
ORG    $EE00          ; inicio de memoria de instrucciones
       LDHX  #511      ; Inicialización del contador de bytes de la tabla.
       CLRA             ; Inicializa la suma de elementos de la tabla.
BUCLES UMA ADD   TABLA, X    ; suma el acumulador con la posición 511
de la tabla.
AIX    #-1              ; Reduce el contador de byte (contenido de x)
                        ; se usa AIX por que se genera acarreo de H a X al reducir en 1
CPHX #0                 ; si es cero terminó (comparo el registro x con cero).
                        ; CPHX setea bits del CCR
BPL  BUCLES UMA  ; salta a BUCLES UMA si X no llegó a cero.
```

B. 3. Lógicas

- AND #$80; hace una AND entre el acumulador y el número 80, el resultado se guarda en el acumulador.
- ORA #$80; hace una OR entre el número 80 y el acumulador.
- EOR #$80; hace una OR exclusiva entre el número 80 y el acumulador.

B.4. Manipulación de datos

- ROL $80; desplaza a izquierda el contenido de la dirección 80, el contenido del carry se introduce por el bit MSL y el bit MSB se introduce en el carry.
- RORA; desplaza a derecha el contenido del acumulador, el contenido del *carry* se introduce por el bit MSL y el bit MSB se introduce en el carry.

B.5. Manipulación de bits

- BCLR 7,$80; borra el bit 7 de la dirección 80.
- CLC; borra el bit de *carry* del registro CCR.

B.6. Control de flujo de programa

- BRA $80; salto directo a la dirección 80.
- BRCLR 7, $80, bucle; salta a la etiqueta bucle si el bit 7 de la dirección 80 está borrado.
- CBEQ $80, bucle; compara el acumulador con el contenido de la dirección 80, si; iguales salta a bucle.
- DBNZ $80, bucle; reduce el contenido de la dirección 80 y salta si es cero.

Instrucción CBEQ combina las instrucciones CMP y BEQ

- Operaciones más rápidas de búsqueda/acceso a tablas.

Instrucción DBNZ combina las instrucciones DEC y BNE

- Bucles más rápidos y eficientes.

Con estas dos nuevas instrucciones, se consigue un manejo más sencillo y eficiente de operaciones repetitivas como "bucles", búsqueda/acceso a tablas.

- JMP $F800; salto extendido a la dirección F800.
- JSR $F800; salto extendido a la subrutina que está en la dirección F800.
- BSR bucle; salto a la subrutina bucle.
- RTS; instrucción que retorna de subrutina.

- SWI; interrumpe el flujo de ejecución de programa y ejecuta el código indicado en el vector de interrupciones SWI.

- RTI; Retorna de la interrupción a la dirección donde se estaba ejecutando el programa principal.

B.7. Operaciones en BCD

- DAA; ajusta el acumulador a un número decimal.
- NSA; intercambia los *nibbles* del registro acumulador (los 4 bit más significativos con los

; menos significativos).

B.8. Especiales

- RSP; resetea el registro puntero de pila a la dirección por defecto $FF.
- NOP; instrucción que no hace nada, consume un ciclo de reloj.
- STOP; detiene el procesador y espera por una interrupción.

Instrucción WAIT:

- El CPU08 detiene el procesamiento de instrucciones.
- Espera por una interrupción.
- No se detiene el oscilador.

Instrucción STOP:

- El CPU08 detiene el procesamiento de instrucciones
- Detiene el circuito del oscilador
  - Pone al MCU en estado “*low power*”
- Espera por una interrupción.

Referencias

[1] James W. Bignell y Robert L. Donovan (2005), *Electrónica digital* . México: Continental.
[2] A. C. Dowton (1993), "Computadores y microprocesadores: Componentes y sistemas", Versión en español de Ernesto Morales Peake, Addison-Wesley.
[3] H. Taub (Taub), *Circuitos digitales y microprocesadores*, Mc Graw-Hill.
[4] Fernando I. Szklanny, Horacio Martínez Del Pezzo (1979), *Introducción a los microprocesadores*, Arbó.
[5] Daniel Di Lella (2005), "Curso de microcontroladores HC908", Electrocomponentes S.A.
[6] William Stalling (02/2006), "Computer Organization and Architecture Designing for Performance",
[7] Reference Manual: *CPU Central Processor Unit Microprocesador, CPU08RM*, Rev. 4.
Pretice Hall (07/2005) (ISBN: 978-0-13-607373-4). 8th. Edition. Año 2010.
[8] Data sheet: MC68HC908QY4 Microcontrolers, Rev. 5.

INSTITUTO DE **INGENIERÍA Y AGRONOMÍA**

Introducción a la organización y arquitectura de la computadora presenta la descripción del funcionamiento de un sistema de procesamiento partiendo desde la descripción de los sistemas de numeración que utilizan, pasando por la organización interna del sistema, para concluir con la descripción de la arquitectura y los distintos elementos que provee para su programación a bajo nivel.

El texto se enfoca en la descripción de los contenidos mínimos que se requieren para comprender el funcionamiento interno del sistema de cómputo y su constitución, incluyendo los contenidos principales de la asignatura que habitualmente se encuentran distribuidos en diferentes bibliografías. Esta producción se logró mediante un esfuerzo conjunto entre el autor y la UNAJ con el objeto de mejorar la calidad de la enseñanza y contribuir con una Universidad pública inclusiva.