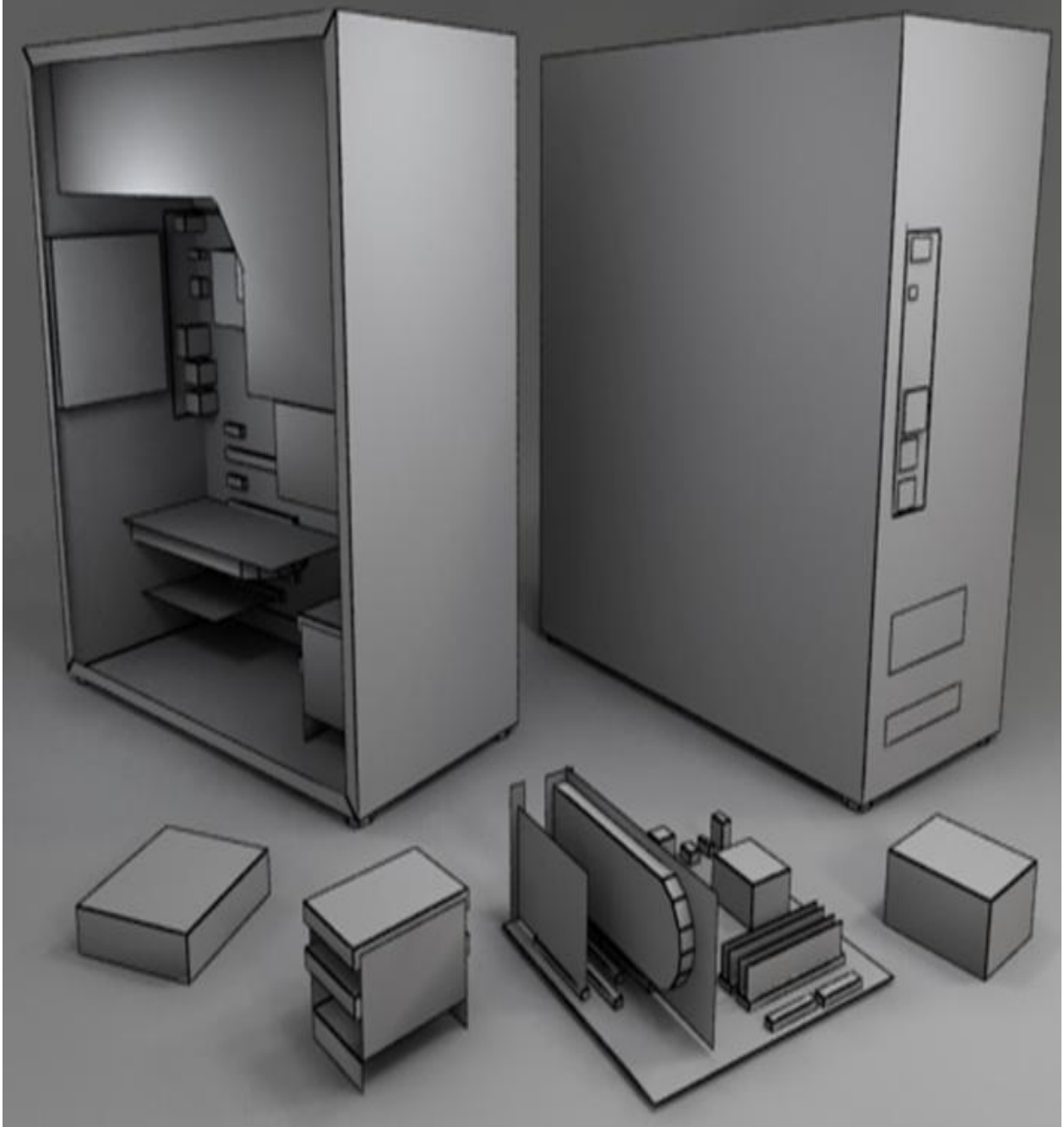
ARQUITECTURA Y ORGANIZACIÓN DE LA COMPUTADORA

Microprocesadores y Programación Assembler



Sergio Hernán Rocabado Moreno , Daniel Arias Figueroa

ARQUITECTURA Y ORGANIZACIÓN DE LA COMPUTADORA

Microprocesadores y Programación Assembler

Se trata de un libro dirigido específicamente al estudiante universitario de aquellas carreras en las que la arquitectura de computadores es una materia fundamental. El libro se compone de dos partes bien diferenciadas: una primera de tipo teórico, en el que se describe la estructura de una computadora analizando sus diversos componentes, y una segunda eminentemente práctica, donde se presenta la programación en lenguaje ensamblador mediante ejemplos prácticos que facilitan el aprendizaje de este lenguaje. En los anexos se repasan conceptos de sistemas de numeración, microprogramación y codificación de las instrucciones.

Editorial: FUNTICs
ISBN: 978-987-25293-0-7

Rocabado Moreno, Sergio Hernán
Arquitectura y organización de la computadora: microprocesadores y programación assembler / Sergio Hernán Rocabado Moreno y Daniel A. Arias Figueroa . - 1a ed. - Salta : FUNTICs, 2009.
CD-ROM.

ISBN 978-987-25293-0-7

1. Programación. 2. Assembler. 3. Microprocesadores. I. Arias Figueroa , Daniel A. II. Título
CDD 005.13

Fecha de catalogación: 19/08/2009

Ficha de Catalogación:

Titulo: ARQUITECTURA Y ORGANIZACIÓN DE LA COMPUTADORA

Subtitulo: MICROPROCESADORES Y PROGRAMACION ASSEMBLER

Nombre de los autores: Sergio Hernán Rocabado Moreno - Daniel A. Arias Figueroa

Primera Edición: Julio de 2009.

I.S.B.N. Nº: 978-987-25293-0-7

Tiradas: 100

Editorial: FUNTICs – Fundación para la Investigación y Desarrollo en Nuevas Tecnologías
Av. San Martín Nº 260 – CP. 4400 – Tel./Fax. 0387-4219889
Web: www.funtics.com.ar – E-mail: info@funtics.com.ar

Dirección: Presidente CPN. Sergio Eduardo Arias

Registro: Mgr. Daniel A. Arias Figueroa

Impresión: FUNTICs

Queda hecho el Depósito que marca la ley 11.723
Impreso en Argentina – Printed in Argentina

Agradecimientos

Agradecemos la colaboración de todas aquellas personas que hicieron posible la realización de este material.

En especial queremos agradecer a la Fundación para la Investigación y Desarrollo en Nuevas Tecnologías – FUNTICs por el interés en publicar este material.

Sobre los autores

Sergio H. Rocabado Moreno

Es Postgrado en Redes de Datos por la Universidad Nacional de la Plata. Profesor en la Universidad Nacional de Salta desde el año 1991 a la fecha en las asignaturas Arquitectura y Organización de la Computadora y Sistemas Operativos de la Licenciatura en Análisis de Sistemas – Departamento de Informática – UNSa.

Daniel Arias Figueroa

Es Magíster en Redes de Datos y Postgrado en Ingeniería de Software por la Universidad Nacional de La Plata, Profesor en la Universidad Nacional de Salta desde el año 1989 a la fecha. Director del CIDIA – Centro de Investigación y Desarrollo en Informática Aplicada dependiente de la Facultad de Ciencias Exactas de la UNSa.

Estimado Lector

Este es un libro dirigido específicamente al estudiante universitario de aquellas carreras en las que la arquitectura de computadores es una materia fundamental. El libro se compone de dos partes bien diferenciadas: una primera de tipo teórico, en el que se describe la estructura de una computadora analizando sus diversos componentes, y una segunda eminentemente práctica, donde se presenta la programación en lenguaje ensamblador mediante ejemplos prácticos que facilitan el aprendizaje de este lenguaje. En los anexos se repasan conceptos de sistemas de numeración, microprogramación y codificación de las instrucciones.

INDICE GENERAL

LOS MICROPROCESADORES

1 CONCEPTOS BASICOS

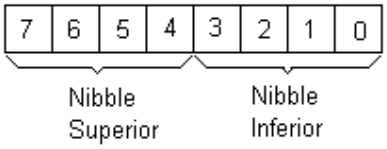
1.1 Estructura de la memoria de la computadora

La memoria se compone de unidades de almacenamiento llamadas bits, que tienen dos estados posibles (representados por 0 y 1), es decir, sirven para almacenar números expresados en sistema binario.

Los bits de la memoria se agrupan en bytes, a razón de 8 bits por byte. Un byte es realmente la unidad de direccionamiento, es decir podemos referirnos a un byte mediante un número que es su dirección. La cantidad de memoria de una computadora se mide en Kilobytes (Kb o k):

1 Kilobyte = 1Kb = 1k = 1024 bytes

La agrupación de los 4 bits (superiores o inferiores) de un byte se llama Nibble. Por tanto, un byte contiene 2 nibbles.



Dos bytes (16 bits) forman una palabra en el 8086.

1.2 Tipos de Datos en Memoria

1.2.1 Números Binarios

Pueden ser con signo y sin signo y ocupan desde 1 byte hasta 4 palabras, los números con signo reservan 1 bit para el signo. Los números máximos que se puede representar con signo y sin signo utilizando 1 byte son:

Sin signo: [0 - FFh] = [0 - 255]

Con signo: [-128 - 127]

1.2.2 Números decimales desempaquetados

Cada byte contiene un dígito BCD (Dígito decimal del 0 al 9) en los 4 bits inferiores y un código especial (generalmente nulo) llamado zona en los 4 bits superiores.

1.2.3 Números decimales empaquetados

Cada byte contiene dos dígitos BCD. El dígito menos significativo se almacena en el nibble inferior.

1.2.4 Caracteres ASCII

Además de representar valores numéricos, los bytes se usan para representar caracteres. Cada byte puede representar 256 caracteres posibles:

- Los 128 primeros (0-127) son los caracteres ASCII estándar
- Los 128 últimos (128-255) son los caracteres ASCII extendidos.

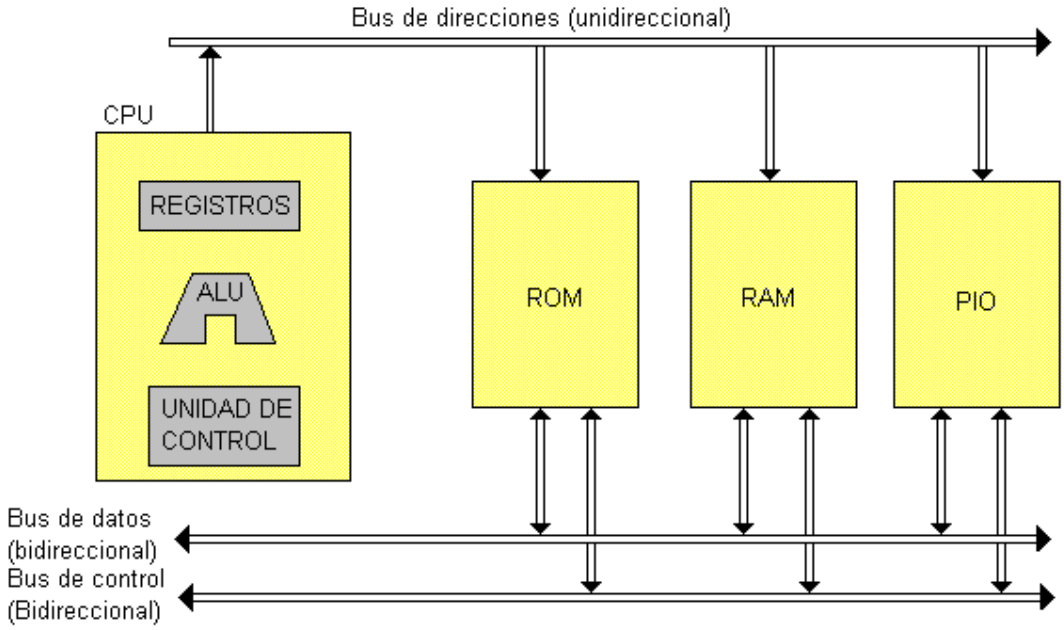
El código ASCII (American Standard Code for Information Interchange) es un convenio adoptado para asignar a cada carácter un valor numérico.

1.3 El Concepto de Computadora

Una computadora es una máquina de tratamiento automático de la información que manipula señales eléctricas binarias.

1.4 Componentes de una computadora

- La Unidad Central de Proceso (CPU).
- La Memoria.
- Los Controladores.
- Las Unidades de E/S.



Todos los componentes del circuito principal, los que hacen que la computadora funcione, están situados en la placa principal (Placa base o placa madre); otros elementos esta situados en placas de expansión, que pueden ser conectadas a la placa principal.

1.4.1 El Microprocesador

El microprocesador es el Chip que ejecuta los programas, lleva a cabo una gran variedad de cálculos, comparaciones numéricas y transferencia de datos como respuesta a las peticiones de los programas almacenados en memoria.

La CPU (Microprocesador) controla las operaciones básicas de la computadora enviando y recibiendo señales de control, direcciones de memoria y datos de un lugar a otro a través de un grupo de sendas electrónicas denominadas bus.

El microprocesador se compone de:

La unidad de control:	Interpreta las instrucciones y genera las señales de control para que se ejecuten.
La unidad de cálculo:	ALU (Unidad Aritmética y Lógica) siguiendo las órdenes de la unidad de control, recibe los datos de la memoria, opera con ellos y almacena el resultado en la memoria.

Registros: Son unidades de almacenamiento de alta velocidad que permiten almacenar información necesaria para el procesamiento.

El microprocesador esta conectado a un oscilador o reloj que genera impulsos igualmente espaciados en el tiempo, el microprocesador divide esa frecuencia base por una constante para implementar un ciclo de máquina. Cada instrucción que ejecuta el microprocesador consume un número determinado de ciclos de máquina.

1.4.2 El Bus

El bus es simplemente un canal de comunicación entre todas las unidades del sistema, donde cada una esta conectada. Un bus se compone en varias líneas o hilos (uno por bit) por el que circula un cierto tipo de información. El bus a su vez se divide en tres tipos, según la información que circula por el mismo:

Bus de datos: Se utiliza para transmitir datos entre los componentes de la computadora.

Bus de direcciones: Se utiliza para transmitir las direcciones de las posiciones de memoria y de los dispositivos conectados.

Bus de control: Serie de líneas que sirven básicamente para indicar el tipo de información que viaja por el bus de datos.

1.4.3 Puertos de Entrada/Salida

Son vías de comunicación con otros componentes de la computadora excepto la memoria. Se identifican mediante una dirección pudiéndose leer datos de un puerto y escribir sobre él.

1.4.4 Coprocesador Matemático

El microprocesador solo puede trabajar con números enteros. Durante la compilación de un programa el compilador genera para cada operación en punto flotante una larga y lenta serie de operaciones enteras.

El coprocesador permite hacer cálculos con números de punto flotante lo cual acelera la ejecución. Existe un zócalo en la placa base para este coprocesador.

1.4.5 ¿Cómo se comunica un microprocesador?

Un microprocesador se comunica con el exterior de tres maneras:

- Mediante acceso directo o indirecto a memoria; el acceso directo se logra a través del controlador DMA (Direct Memory Access). Las unidades de disco y las puertas de comunicación serie pueden acceder a la memoria directamente a través de su controlador. El acceso indirecto se logra a través de los registros internos que se utilizan para transferir datos hacia/desde la memoria.

- A través de puertas de entrada/salida.

- Mediante interrupciones; las interrupciones son señales que se le envían a la CPU para que interrumpa la ejecución de la instrucción en curso y atienda una petición determinada, al terminar de atender la petición ejecutara la instrucción que le correspondía.

1.4.6 Los Chips de Apoyo

El microprocesador no puede controlar toda la computadora sin ayuda. Al delegar ciertas funciones de control a otros chips, la CPU queda liberada para atender su propio trabajo. Estos chips de apoyo pueden ser responsables de los siguientes procesos:

- Controlar el flujo de información a través de circuitos internos (como el controlador de interrupciones y el controlador DMA).

- Controlar el flujo de información de uno a otro dispositivo (como un monitor o una unidad de disco) conectada a la computadora. Estos son llamados controladores de dispositivos y generalmente se hallan en placas separadas que se conectan en uno de los canales o ranuras de expansión.

1.4.6.1 El controlador programable de interrupciones (PIC)

Cuando un componente de Hardware necesita la atención de la CPU genera una interrupción para poder ser atendido. Ej. Cada vez que pulsamos una tecla se genera una interrupción de teclado para informarle a la CPU cual fue la tecla pulsada.
La CPU no puede dedicarse solamente a verificar si tiene solicitudes de interrupción ya que tiene otras tareas. Para ello la computadora tiene un circuito PIC que comprueba las interrupciones y las presenta una a una a la CPU.

1.4.6.2 El controlador DMA

Algunas partes de la computadora son capaces de transferir datos hacia y desde la memoria sin pasar por la CPU. Esta operación se denomina acceso directo a memoria y la lleva a cabo un chip conocido como el controlador DMA. El propósito principal del controlador DMA es el de permitir a las unidades de disco leer o escribir datos prescindiendo del microprocesador.

1.4.6.3 Controladores de entrada/salida

Subsistemas de entrada/salida con circuitos de control especializados que proporcionan una interfase entre la CPU y el hardware de E/S. Por ejemplo, el teclado tiene un chip controlador propio que transforma las señales eléctricas producidas por las pulsaciones de teclas en un código de 8 bits que representa la tecla pulsada. Todas las unidades de disco disponen de circuitos independientes que controla directamente la unidad; la CPU se comunica con el controlador a través de una interfase coherente.

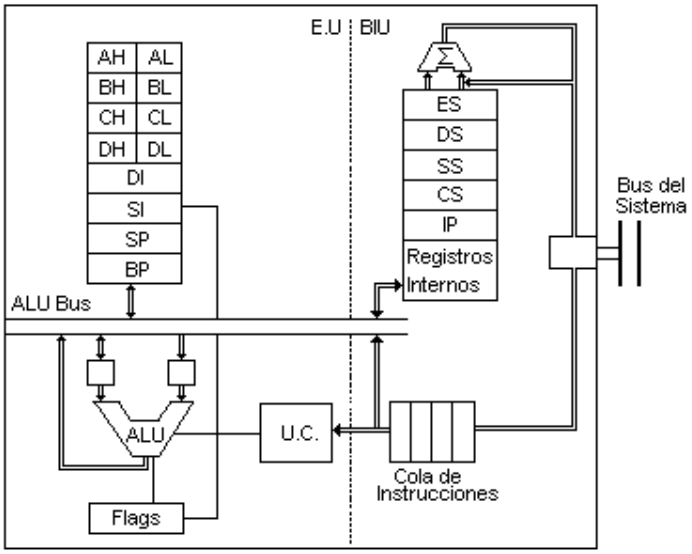
1.4.7 La memoria

Los chips de memoria al contrario que los chips de apoyo, no controlan directamente el flujo de información a través de la computadora, se dedican meramente a almacenar la información hasta que se la necesita.

1.4.8 Filosofía de diseño

Las PC's compatibles son funcionalmente iguales a las originales pese a tener circuitos diferentes, la compatibilidad se logra a través del BIOS o sistema básico de entrada y salida que esta formado por un conjunto de rutinas que manipulan la entrada y salida de la computadora. El fabricante de PC debe proporcionar el BIOS adecuado para el hardware de sus PC's, así el programador no necesitará saber los detalles técnicos de hardware sino simplemente conocer la forma de llamar a las rutinas del BIOS que debe ser la misma para cualquier BIOS.
Las rutinas del BIOS se cargan en memoria al arrancar la computadora y pueden ser llamadas desde cualquier programa.
La filosofía básica de la familia PC es “Deje que la ROM BIOS lo haga, no pierda tiempo con el control directo del hardware de E/S”.
La utilización de las rutinas de la BIOS facilita las buenas prácticas de programación y garantiza que los programas funcionen en cualquier PC original o compatible.

2 EL MICROPROCESADOR 8086



2.1 Direccionamiento de la memoria en el 8086

El 8086 tiene registros de 16 bits y un bus de direcciones que le permite direccionar 1Mb de RAM. Pero debe usar una técnica especial para poder direccionar un mega ya que el máximo número que puede guardar un registro de 16 bits es 64k. Para superar este límite, se utilizan dos registros para hacer referencia a una dirección:

- Segmento.
- Desplazamiento (Offset).

Y por lo tanto la dirección completa se calcula como:

(16 x Segmento) + Desplazamiento = (10h x Segmento) + Desplazamiento

Realmente no se multiplica por 10h, sino que se desplazan 4 bits a la izquierda del registro de segmento.

Segmento	=	XXXX0 (Hex)
Desplazamiento	=	YYYY (Hex) +

Dirección	=	ZZZZZ (Hex)

La dirección completa es de 20 bits, que es la longitud del bus de direcciones. De esta manera es posible direccionar entre las direcciones:

[0 – FFFFF Hex]

Cada segmento puede ser de hasta 64Kb de longitud y comienza en una posición que es múltiplo de 16. Esta dirección se llama también párrafo del segmento.

Una dirección completa con sus dos componentes se expresa de la siguiente manera:

[Segmento:Desplazamiento]

Y también

Segmento:[Desplazamiento]

Este método de direccionamiento se denomina direccionamiento segmentado.

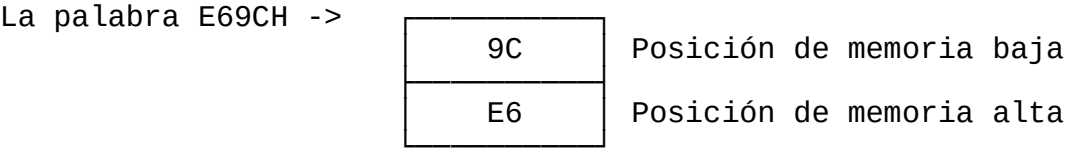
Una dirección en memoria se puede expresar de distintas maneras. Por ejemplo, las siguientes direcciones (en hexadecimal) son equivalentes:

100:50 = 1050 = 105:0 = 0:1050 = 10:950

Lo que implica que puede haber un gran número de posibilidades de solapamiento de direcciones segmentadas. Algunas direcciones físicas pueden ser representadas por hasta (2^{12}) direcciones segmentadas.

2.2 Almacenamiento inverso de palabras

A pesar de que la memoria del 8086 esta direccionada en unidades de bytes de 8 bits, muchas operaciones introducen palabras de 16 bits. En la memoria, una palabra de 16 bits se almacena en dos bytes adyacentes de 8 bits. El byte menos significativo de la palabra se almacena en la posición de memoria más baja, y el más significativo en la posición de memoria más alta. Debido a la apariencia inversa de este esquema de almacenamiento es denominado algunas veces como "almacenamiento inverso de palabras". Por ejemplo:



2.3 Recuperación y ejecución de instrucciones en el 8086

El ciclo de ejecución de las instrucciones de un programa consta de dos fases. La primera fase consiste en la recuperación de una instrucción. La segunda fase es la ejecución en si misma. Con objeto de optimizar estos dos procesos, el microprocesador posee dos unidades separadas:

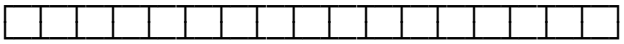
- Para recuperación de instrucciones: BIU (Bus Interface Unit).
- Para ejecución de instrucciones: EU (Execution Unit).

Existe lógicamente una comunicación entre la BIU y la EU, es decir que cada instrucción recuperada por la BIU pasa a la EU para que la ejecute. Mientras se ejecuta la instrucción la BIU recupera la siguiente en la memoria, que será la que se ejecute a continuación (si no hay bifurcación).

La BIU lleva una cola de las 4 instrucciones que le siguen a la que se ejecuta en la EU, la cola se va cargando mientras la EU esta ejecutando una instrucción. Cuando la EU termina de ejecutar la instrucción pasa a leer la primera de la cola y la ejecuta, pero si la instrucción que ejecuto es una instrucción de salto la cola se vacía y la BIU recupera la primera instrucción de la dirección especificada en la instrucción de salto y se la pasa a la EU para que la ejecute, mientras tanto prepara la cola con las instrucciones que siguen.

2.4 Los registros internos del microprocesador 8086

Los registros internos son 14, todos de 16 bits (Una palabra). Los bits se numeran de derecha a izquierda. El bit 0 es el menos significativo.



F E D C A B A 9 8 7 6 5 4 3 2 1 0 Nro. de bit

Existen:

2.4.1 Cuatro registros de datos o almacenamiento temporal

AX = Acumulador
Es el registro principal utilizado en las operaciones aritméticas.
BX = Base
Se utiliza para indicar un desplazamiento (Offset).
CX = Contador

Se utiliza para contador en los ciclos y en las operaciones de tipo repetitivo.

DX = Dato (Se usa también en operaciones aritméticas)

Es posible referirse al byte superior (mas significativo) o al byte inferior (menos significativo) en los registros AX, BX, CX, DX:

Registro	Byte Superior (Bits 15 a 8)	Byte Inferior (Bits 7 a 0)
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

2.4.2 Cuatro registros de segmento

Contienen la dirección de comienzo de ciertos segmentos de memoria.

CS = Registro de segmento de código (code segment).
Contiene la dirección del segmento de código, es decir, las instrucciones del programa.

DS = Registro de segmento de datos (data segment).
Contiene la dirección del segmento de datos, es decir el área de memoria donde se encuentran los datos del programa.

SS = Registro de segmento de pila (stack segment)
Contiene la dirección del segmento de pila. La pila es un espacio de memoria temporal que se utiliza para almacenar valores de 16 bits (palabras).

ES = Registro de segmento extra (extra segment).
Contiene la dirección del segmento extra, que es un segmento de datos adicional que se utiliza para superar la limitación de los 64 KB del segmento de datos y para hacer transferencia de datos entre segmentos.

2.4.3 Dos registros punteros de pila

SS = Puntero de pila (stack pointer).
Contiene la dirección relativa al segmento de pila.

BP = Puntero base (base pointer)
Se utiliza para acceder a los elementos de la pila.

2.4.4 Dos registros índices

Se utilizan como desplazamiento relativo a un campo de datos.

SI = Índice fuente (source index).
DI = Índice destino (destination index).

2.4.5 Un registro puntero de instrucciones

IP = Puntero de instrucción (Instruction pointer).
Contiene el desplazamiento de la próxima instrucción a ejecutarse.
En conjunción con el registro CS, indica la dirección completa de la siguiente instrucción a ejecutar, es decir: [CS:IP].

2.4.6 Un registro de banderas (flags)

Contiene información de estado y de control de las operaciones del microprocesador.

De los 16 bits del registro se utilizan solo 9 y cada uno de estos representa a una bandera. Existen:

2.4.7 Seis banderas de estado

Registran el estado del procesador, normalmente asociado a una comparación a o una instrucción aritmética:

CF - (Carry Flag) Bandera de acarreo. Indica acarreo en las instrucciones aritméticas.
OF - (Overflow Flag) Bandera de desbordamiento aritmético.
ZF - (Zero Flag) Bandera de resultado cero o comparación igual.
SF - (Sign Flag) Bandera de resultado o comparación negativa.
PF - (Parity Flag) Bandera de paridad, utilizada en la verificación de la transferencia de bytes entre dos componentes de la computadora.
AF - (Auxiliar Flag) Bandera auxiliar. Indica si hay necesidad de ajuste en las operaciones aritméticas con números BCD.

2.4.8 Tres banderas de control

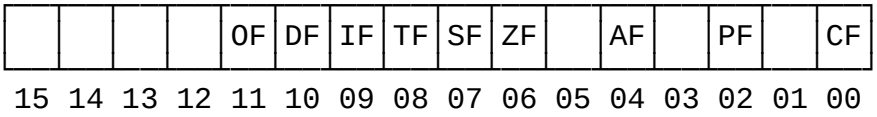
Registran el modo de funcionamiento de la computadora.

DF (Direction Flag) Bandera de dirección. Controla la dirección (hacia adelante o hacia atrás) en las operaciones con cadenas de caracteres incrementando o decrementando automáticamente los registros índices (SI y DI).

IF (Interrupt Flag) Bandera de interrupciones. Indica si están disponibles o no las interrupciones de los dispositivos externos.

TF (Trap Flag) Bandera de atrape. Controla la operación modo paso a paso (usada por el programa DEBUG).

Las posiciones de las banderas dentro del registro son:



2.5 La unidad de control

La unidad de control es la encargada de decodificar las instrucciones almacenadas en la cola de instrucciones y generar las señales de control necesarias para su ejecución.

Ejemplo: Instrucción AND AX, CX realiza un AND lógico entre los registros AX y CX almacenando el resultado en el registro AX.

Señales de control:

Poner el contenido del registro AX en el bus de datos de la ALU.
Poner el contenido del bus de datos de la ALU en el primer registro temporario de la ALU.
Poner el contenido del registro CX en el bus de datos de la ALU.
Poner el contenido del bus de datos de la ALU en el segundo registro temporario de la ALU.
Realizar la operación AND en la ALU.
Poner el contenido del bus de datos de la ALU en el registro AX.

2.5.1 La cola de instrucciones

Mientras la unidad de ejecución ejecuta las instrucciones, la unidad de interfaz de bus esta buscando la siguiente instrucción y las va colocando en una cola de 4

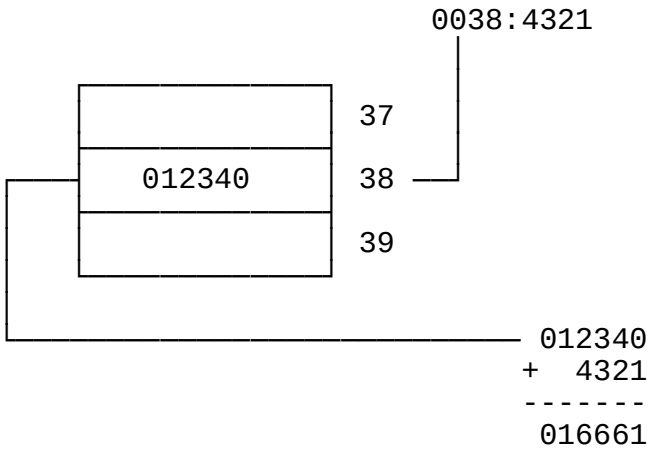
instrucciones, de esta manera cuando la unidad de control termina la ejecución de una instrucción no hace falta que la siguiente instrucción se busque en memoria, sino se la toma de la cola de instrucciones. Este mecanismo no es eficiente si tenemos instrucciones de salto.

3 OTROS MICROPROCESADORES

3.1 Direccionamiento de la memoria en el 80286

El 80286 también utiliza el direccionamiento segmentado, pero funciona en modo protegido, las direcciones se decodifican de manera distinta que en el 8086 o que en el mismo 80286 funcionando en modo real.

El 80286 decodifica las direcciones segmentadas a través de una tabla de descriptores de segmento. Cada descriptor de la tabla contiene una dirección base de 24 bits que indica el comienzo real de un segmento en memoria. La dirección resultante es la suma de la dirección base de 24 bits y del desplazamiento de 16 bits especificado en la dirección segmentada. De esta manera en modo protegido el 80286 puede acceder hasta (2^{24}) bytes de memoria; o sea, las direcciones físicas son de un tamaño de 24 bits.



Este esquema proporciona al 80286 grandes posibilidades sobre el control de la utilización de la memoria. Además de la dirección base de 2 bits, cada descriptor de segmento proporciona unos atributos de segmento (Código ejecutable, datos de programa, solo lectura, etc.), así como un nivel de privilegio que permite al sistema restringir el acceso al segmento.

3.2 Direccionamiento de la memoria en el 80386

El 80386 soporta el direccionamiento tanto del 8086 como del 80286 en modo protegido. El 80386 desarrolla el esquema de direccionamiento de modo protegido permitiendo direcciones base de 32 bits y desplazamientos de 32 bits. De esta manera,

una dirección segmentada simple que consiste en un selector de 16 bits y un desplazamiento de 32 bits, puede especificar cualquiera de (2^{32}) direcciones físicas.

El 80386 también proporciona un modo de direccionamiento "8086" virtual, en el cual el direccionamiento es el usual del 8086, pero las direcciones físicas corresponden a 1Mb del espacio de direccionamiento del 8086 mapeado en cualquier parte de los 4GB del espacio de direccionamiento del 80386. Esto permite a un sistema operativo que ejecute varios programas del 8086, cada uno en su propio espacio de direcciones de 1Mb compatible con el 8086.

3.3 Los registros internos del microprocesador 80386

Existen 32 registros que pueden ser divididos en 7 categorías principales:

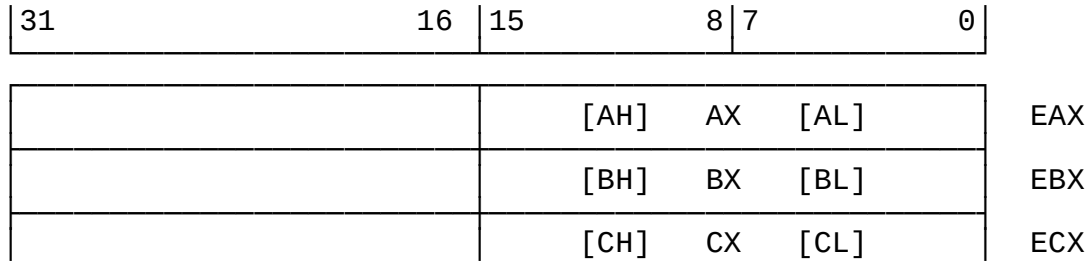
- Registros de propósito general.
- Registros de segmentos.
- Registros de instrucción y banderas.
- Registros de control.
- Registros de direcciones del sistema.
- Registros de test.

Todos los registros de 16 bits del 8086/80286 están contenidos en el microprocesador 80386.

3.3.1 Ocho Registros de propósito general

Los registros de propósito general son capaces de soportar operandos de 1, 8,16 ,32 bits y campos de bits de 1 a 32 bits. Estos registros también soportan operandos de direcciones de 16 y 32 bits.
Los 16 bits de orden inferior pueden ser accedidos utilizando la denominación que tenían en el 8086/80286.
Ejemplo:

```
EAX (32 Bits) ———> AX (16 bits) ———> AH,AL (8 bits)
```



	[DH] DX [DL]	EDX

	SP	ESP
	BP	EBP
	SI	ESI
	DI	EDI

3.3.2 Seis Registros segmentos

Son registros de 16 bits que contienen el valor para entrar a la tabla de descriptores de segmento.

Además de los 4 registros del 8086/80286 (CS, DS, SS, ES), incluye dos registros segmento que son FS y GS y que el programador puede usar para referenciar a cualquier segmento.

3.3.3 Un registro puntero de instrucción

Es un registro de 32 bits denominado IP, que con CS forma la dirección de la siguiente instrucción a ejecutar: [CS:EIP]. Los 16 bits inferiores de EIP pueden se accedidos separadamente, a esos bits se les denomina IP.

3.3.4 Un registro de banderas (Flags)

El registro EFLAGS del 80386, ha sido también extendido a 32 bits, se utiliza para controlar ciertas operaciones e indicar el status del mismo 80386.
Los 16 bits de orden inferior del registro EFLAG se denominan FLAG y tienen el mismo contenido que el registro de banderas del 8086/80286. De los otros 16 bits sólo se utilizan dos para dos nuevas banderas.

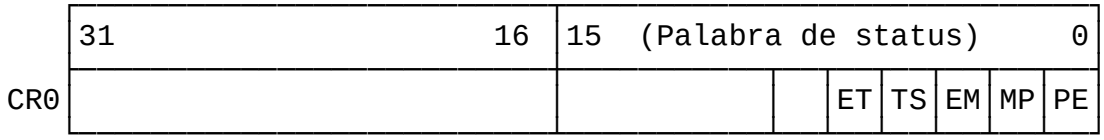
Bit 16: Flag VM (Virtual Mode) indica si el procesador esta en modo virtual.

Bit 17: Flag RF (Resume Flag) para ayudar al TF en la depuración paso a paso.

3.3.5 Cuatro registros de control. (CR0, CR1, CR2, CR3)

Son registros de 32 bits que contienen información sobre el status no dependiente de la tarea de la máquina y se accede a ellos a través de instrucciones de carga y almacenamiento especiales.

El registro CR0 contiene cinco señalizadores predefinidos usados para el control del microprocesador y propósitos de status. Los bits del 0 al 15 de este registro son conocidos como palabra de status de la máquina.



Los señalizadores tienen el siguiente propósito:

PE: Se utiliza para activar el modo protegido del ordenador.

MP: Indica si se tiene instalado un coprocesador.

EM: Se utiliza para indicar que emule coprocesador.

TS: Indica si se esta conmutando una tarea.

3.3.6 Cuatro registros de direcciones del sistema.

Estos registros de 32 bits son usados para referenciar las tablas o segmentos necesarios para soportar el modo protegido del 80386. Generalmente es el microprocesador el que utiliza esta información, pero puede ser accedida por el programador.

3.3.7 Seis registros de depuración y test

Son los registros DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7 y permiten tener un control completo de la ejecución paso a paso de un programa.

4 EL LENGUAJE ENSAMBLADOR

El programa ensamblador convierte los nombres simbólicos de las instrucciones (Nemonicos) en código máquina. A diferencia de un compilador de lenguaje de alto nivel que genera una serie de instrucciones de máquina por cada instrucción de alto nivel, el ensamblador genera una instrucción de máquina por cada nemonico.

4.1 Estructura de un programa COM

A continuación se muestra la estructura de un programa COM, que puede ser aplicada a cualquier programa.

```
; Definición de constantes

CR equ 0dh
LF equ 0ah

; Definición de segmentos

CSEG Segment para public 'CODE'
     Org 100h
     Assume  CS:Cseg;DS:Cseg;ES:Cseg;SS:Dseg

; Definición de variables

Mensaje db CR,LF,'Esto es un mensaje',CR,LF,'$'
Num dw 0

; Código del programa

Principal Proc near
     ....
     .... ————————————> Llamada cercana
     mov ah,4CH
     int 21H
Endp Principal

Otro Proc Near
     ....
     ....
     Ret ————————————> Retorno de la llamada
End Otro

Cseg Ends
End Principal  ; Punto de entrada del programa
```

Observamos que el procedimiento principal termina con una llamada a la interrupción de fin de programa, mientras que los demás procedimientos finalizan con la instrucción RET, pues deben devolver el control al procedimiento por el cual fueron llamados.

La directiva ORG 100H le indica que comience el programa en la base del PSP. La directiva ASSUME no tiene efecto directo alguna sobre el contenido de los registros de segmento, solo afecta a la operación del ensamblador.

4.2 Como generar un programa COM

Para generar un programa .COM se deben seguir los siguientes pasos:

1) Crear el código fuente con algún editor de texto.
2) Utilizar el MASM.EXE para crear un objeto a partir del fuente:

```
MASM Nombre_fuente.ASM;
```

3) Utilizar el LINK.EXE para crear un EXE a partir del objeto:

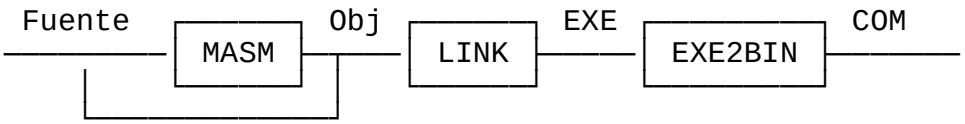
```
LINK Nombre_fuente.OBJ
```

4) Utilizar el EXE2BIN.EXE para crear un COM a partir del EXE:

```
EXE2BIN Nombre_fuente.EXE  Nombre_destino.COM
```

5) Eliminar el objeto y el EXE:

```
DEL Nombre_fuente.OBJ
DEL Nombre_fuente.EXE
```



La figura ilustra los pasos a seguir, observamos una flecha de la salida de MASM a fuente, esto ocurre si es que el programa tiene errores y es necesario modificarlo; en ese caso el ensamblador no generara el programa OBJ.

4.3 Estructura de un programa EXE

A continuación se muestra la estructura de un programa EXE, que puede ser aplicada a cualquier programa.

```
; Definición de constantes
CR equ 0dh
LF equ 0ah
; Definición de segmentos
; Segmento de datos, con el atributo Data
DSEG segment para 'DATA'
; Definición de variables
Mensaje db CR,LF,'Esto es un mensaje',CR,LF,'$'
Num dw 0
DSEG ends

; Segmento de pila, con el atributo Stack

SSEG segment para stack 'STACK'
     dw 64 dup(?)
SSEG ends

CSEG Segment para public 'CODE'
Assume  CS:Cseg;DS:Cseg;ES:Dseg;SS:Dseg

; Codigo del programa

Principal Proc Far
     MOV AX,DSEG     ;    DS al comenzar apunta al PSP
     MOV DS,AX       ;    Se debe hacer que apunte al seg-
                          mento de datos
     ....
     .... ————————————>  llamada lejana
     mov ah,4CH
     int 21H
Endp Principal


Otro Proc Far
     ....
     ....
Ret  ————————————>   Retorno lejano
End Otro

Cseg Ends
End Principal  ; Punto de entrada del programa
```

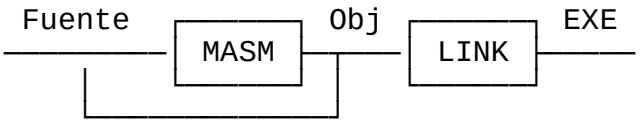
4.4 Como generar un programa EXE

Para generar un programa .EXE se deben seguir los siguientes pasos:

1) Crear el código fuente con cualquier editor de texto.
2) Utiliar el MASM.EXE para crear un objeto a partir del fuente:
   MASM Nombre_fuente.ASM;

3) Utilizar el LINK.EXE para crear un EXE a partir del objeto:
   LINK Nombre_fuente.OBJ

4) Eliminar el objeto:
   DEL Nombre_fuente.OBJ



4.5 Segmentos y registros asociados a un programa

Un programa consta de 4 tipos de segmentos. Cada segmento se direcciona mediante un determinado tipo de registro de segmento.

Segmento de código	Cada instrucción se direcciona mediante: Registro de segmento: CS. Registro de desplazamiento: IP.
Segmento de datos	Los datos se direccionan mediante: Registro de segmento: DS. Registro de desplazamiento: BX, SI o DI.
Segmento de pila	Los datos se direccionan mediante: Registro de segmento: SS Registro de desplazamiento: SP, BP.
Segmento Extra	Igual que el de datos sustituyendo ES por DS.

Estas asignaciones por defecto se pueden modificar en el programa.

4.6 Sentencias fuentes

Las sentencias fuentes de un programa fuente ensamblador pueden ser:

Instrucciones - También denominadas nemonicos, son representaciones simbólicas del juego de instrucciones del microprocesador.

[Etiqueta] Nombre [Operandos]

Entre corchetes las partes opcionales.

Ejemplos:

```
ETI1:  MOV AX,CX   ; Aparece una etiqueta y dos operandos.
       INC CX      ; Aparece un solo operando.
       STI         ; Instrucción sin operandos.
```

Directivas - Son indicadores para el ensamblador, le indican que debe hacer con los datos y las instrucciones.

4.7 Tipos de operandos

Los operandos de las instrucciones pueden ser:

Registro	De 8 o 16 bits en el 8086/80286, también de 32 bits en el 80386. Por ejemplo: AX -> Nombre del registro.
Memoria	Puede ser un byte de memoria o una palabra de memoria. Se especifica mediante una dirección, Por ejemplo: DS:[DI] -> El operando se halla en esa dirección.
Valor Inmediato (Dato)	Un número o una constante, Por ejemplo: 0FF0H -> Por defecto son decimales a no ser que se le coloque la H después del número para indicarle que es hexadecimal.

4.8 Modos de direccionamiento en el 8086

Las instrucciones que tienen operandos obligan al procesador a recuperar estos, para ello el procesador debe direccionar de alguna manera (Modo de direccionamiento) el operando, lo que no implica que necesariamente deba acceder a la memoria.

Según el operando que tenga la instrucción los modos de direccionamiento pueden ser de tres tipos.

Inmediato: La instrucción tiene un operando que es un dato o una constante.
Registro: La instrucción tiene un operando que es un registro.
Memoria: La instrucción tiene un operando que se halla en la memoria.

Existen instrucciones de dos operandos, un operando, o ningún operando. En las instrucciones de dos operandos por lo menos uno debe ser un registro y en las de un operando este no puede ser un valor inmediato.

La siguiente tabla ilustra todas las posibles maneras de direccionar que tiene el 8086:

#	MODO	OPERANDO	REG	EJEMPLOS
1	Registro	Registro	---	MOV AX,BX
2	Valor	Valor	---	MOV AX,500H
3	Directo	Variable	DS	MOV AX,TABLA

4	Indirecto mediante registro	[SI]
		[DI]
		[BX]
		[BP]
			DS
			DS
			DS
			SS
				MOV AX,[SI]
				MOV AX,[DI]
				MOV AX,[BX]
				MOV AX,[BP]

5	Relativo a base	[BX]+Desp [BP]+Desp	DS SS	MOV AX,VALNUM[BX] MOV AX,[BP]+4
6	Directo indexado	[DI]+Desp [SI]+Desp	DS SS	MOV AX,TABLA[DI] MOV AX,VALNUM[SI]

7	Indexado a base	[BX][SI]+Desp
		[BX][DI]+Desp
		[BP][SI]+Desp
		[BP][DI]+Desp
			DS
			DS
			SS
			SS
				MOV AX,TABLA[BX][SI]
				MOV AX,TABLA[BX][DI]
				MOV AX,TABLA[BP][SI]
				MOV AX,TABLA[BP][DI]

Observaciones:

- La columna REG se refiere al registro de segmento por defecto.
- Los modos 3 al 7 se refieren a direcciones de memoria.
- En el modo 3 TABLA se toma como variable en memoria no como constante.
- Desp: Puede ser una constante definida con la directiva EQU, una variable definida con la directiva DB o DW o un número.

```
VALNUM EQU 0FFFH
TABLA  DB  0200H
```

las siguientes instrucciones son equivalentes:

MOV AX,[BX]0FFFH MOV AX,0FFFH[BX] MOV AX,[BX]+0FFFH
MOV AX,[BX+0FFFH]

4.9 Los prefijos de segmento

Los modos de direccionamiento 3 a 7, es decir, los que se refieren a direcciones de memoria, pueden estar precedidos por un registro de segmento. Por ejemplo:

MOV AX,ES:TABLA[SI]

Esto quiere decir que el registro de segmento correspondiente al operando TABLA[SI], que por defecto es DS, será sustituido por el registro de segmento ES. El código que se genera para la instrucción esta precedido por un byte, que llama código de prefijo de segmento.

La tabla siguiente relaciona las posibles combinaciones entre los cuatro registros de segmento y los seis registros de desplazamiento.

	CS	SS	DS	ES
IP	SI	NO	NO	NO
SP	NO	SI	NO	NO
BP	PREFIJO	POR DEFECTO	PREFIJO	PREFIJO
BX	PREFIJO	PREFIJO	POR DEFECTO	PREFIJO
SI	PREFIJO	PREFIJO	POR DEFECTO	PREFIJO
DI	PREFIJO	PREFIJO	POR DEFECTO	POR DEFECTO Solo cadenas

La directiva ASSUME que asocia un registro de segmento con un determinado segmento de código o datos, de modo que en las instrucciones cuyos operandos hagan referencia a campos de ese segmento les incluye automáticamente el prefijo de segmento, a menos que coincida con el registro de segmento por defecto, en cuyo caso no se incluye.

ASSUME le indica al ensamblador que registro de segmento se va a utilizar para direccionar cada segmento dentro del modulo. Para el segmento de código se debe usar siempre CS y para los segmentos de datos se pueden utilizar DS, ES y SS.

ASSUME sigue normalmente a la sentencia SEGMENT. Si no se usa ASSUME, se debe especificar explícitamente el registro de segmento de las instrucciones.

Con ASSUME al comienzo del segmento de código, el ensamblador genera automáticamente un código de un byte como prefijo de la instrucción, para indicar el registro de segmento a utilizar en la instrucción, en lugar del registro de segmento por defecto, según el modo de direccionamiento. Si el registro definido en ASSUME coincide con el registro de segmento por defecto, no se genera el código de sustitución del segmento.

Por ejemplo, supongamos que TABLA ha sido definido en el segmento de datos. Si en el segmento de código aparece una instrucción con el operando TABLA[DI], se utiliza por defecto el registro de segmento DS, de acuerdo con el tipo de direccionamiento, y no se genera código de prefijo de registro de segmento.

Pero si TABLA se ha definido dentro del segmento de código, es necesario especificar como operando CS:TABLA[DI]. En este caso, el ensamblador genera el código correspondiente (un byte como prefijo de la instrucción) para indicar el segmento a utilizar (CS) en lugar del segmento por defecto (DS).

4.10 Constantes en ensamblador

Existen cinco tipos de constantes:

Binarias	1011b
Decimales	3129d (la letra d es opcional).
Hexadecimales	0E23h (No puede empezar con una letra sino con un número).
Octal	1477q
Carácter	'ABC' toma el código ASCII de esos caracteres.

Las constantes no ocupan lugar en memoria, cuando uno define una constante con la directiva EQU:

```
CR EQU 13H
```

Le esta indicando al ensamblador que cada vez que encuentre una referencia a esta constante en el programa la reemplazara por el valor inmediato 13H.

Se deben definir como constantes todos los números o caracteres que se utilizan mucho en el programa y que son difíciles de memorizar, así al programar bastara con recordar el nombre de la constante. La definición de constantes se hace al inicio del programa fuente fuera de los segmentos.

4.11 Variables en ensamblador

Las variables ocupan lugar en memoria y cada vez que se las referencia el ensamblador reemplaza el nombre de la variable por su dirección. Las variables entonces tendrán el valor que tenga la posición de memoria que le fue asignada. Para definir variables se debe usar la directiva DB o la directiva DW.

DB: Permite definir variables de 1 byte.
DW: Permite definir variables de una palabra (2 bytes).

Ejemplos:

NUM1 DB 100H Define la variable NUM1 de un byte y le asigna el valor 100h.

NUM2 DB 0FFA0100H Define la variable NUM2 de una palabra y le asigna el valor 0FFA0100H.

4.12 Definición de áreas de memoria

Puede ocurrir que un programa necesite manejar áreas de datos de más de una palabra, para definirlas podemos usar la directiva DB o la directiva DW.

Ejemplos:

AREA1 DB 100 DUP(0) Define un área de 100 bytes con el valor 0 y el primer byte del área es la variable AREA1.

AREA2 DW 100 DUP(0) Define un área de 100 palabras(200 bytes) con el valor 0 y la primera palabra del área es la variable AREA2.

Es importante entender claramente el significado del nombre de variable que se le da al primer byte del área, cada vez que referenciamos a esa variable tendremos el valor del primer byte del área y NO el offset del área:

MOV AL,AREA1 Pone en AL el valor del primer byte del área.

MOV AL,[0FF0] La instrucción ya ensamblada.

Pero si referenciamos a esa variable en una dirección compuesta, el ensamblador reemplazará el nombre de la variable por el offset de la misma en el offset que le corresponda:

MOV AL,AREA1[DI]

MOV AL,[0FF0+DI] La instrucción ya ensamblada.

Otra forma de conocer el offset del primer byte del área es utilizar la siguiente definición:

DIRAREA1 DW AREA1 Define la variable DIRAREA1 de 2 bytes y le asigna el offset de la variable AREA1.

La definición de variables se debe incluir en algún segmento.

La palabra reservada DUP(Nro) le indica al ensamblador el número con el que debe rellenar el área. Si se coloca DUP(?) el área de memoria no toma ningún valor.

Ejemplos más complejos de definición de variables:

```
VALORES DW 300,150,2000
```

Define un área de 3 palabras cuya primera palabra es la variable VALORES y vale 300, la segunda palabra vale 150 y la tercera 2000.

```
MENSAJE DB CR,LF,'Se produjo un error ! ',CR,LF
        DB CR,LF,'Pulse una tecla para continuar ',CR,LF
        DB '$'
```

Define un área de caracteres ASCII, note que el primer elemento del área es la variable MENSAJE y el último es el carácter '$' que indica fin de una cadena de caracteres ASCII.

```
TABLA DW 0,0,0,0,0,0,0,0
TABLA DW 8 DUP(0)
```

Ambos son equivalentes.

```
VALOR DW 128*12
```

```
DIREC DW TABLA+14
```

La variable DIREC tiene la dirección de la última palabra del área cuyo primer elemento es la variable TABLA.

4.13 Ejemplo de manejo de áreas en ensamblador

En el siguiente programa (PRUEBA.ASM) se definen 3 áreas en el segmento de datos, luego se direcciona a las mismas.

```
SSEG SEGMENT PARA STACK 'STACK'
```

```
 DW 40 DUP(?)
SSEG ENDS

DSEG SEGMENT PARA 'DATA'
    AREA0 DW 50 DUP(?)
    AREA1 DW 100 DUP(?)
    DIRAREA1 DW AREA1
DSEG ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'
    ASSUME CS:CSEG;DS:DSEG;SS:SSEG

PRINC:
  MOV AX,DSEG
  MOV ES,AX
  MOV DI,0
  MOV AX,ES:DIRAREA1[DI]
  MOV AX,ES:AREA1[DI]
  MOV ES:AREA0[DI],AX
  MOV AH,4CH
  INT 21H
CSEG ENDS
END PRINC
```

Una vez compilado se genera el ejecutable (PRUEBA.EXE) y utilizando el DEBUG de DOS observamos el código que genero:

```
C:\PROG>DEBUG PRUEBA.EXE
-U
11F6:0000   B8E311          MOV AX,11E3
11F6:0003   8EC0        MOV ES,AX
11F6:0005   BF0000          MOV DI,0000
11F6:0008   26          ES:
11F6:0009   8B852C01    MOV AX,[DI+012CH]
11F6:000D   26          ES:
11F6:000E   8B856400    MOV AX,[DI+0064H]
11F6:0012   26          ES:
11F6:0013   89850000    MOV [DI+0000H],AX
11F6:0017   B44C        MOV AH,4CH
11F6:0019   CD21        INT 21H
```

Se debe tener en cuenta las siguientes consideraciones:

11F6 Es el valor asignado por la función EXEC del DOS al segmento de código.

La longitud de las instrucciones es variable.

La referencia a un prefijo de segmento (ES), ocupa un byte.

En el programa se utilizo la variable AREA1 para direccionar, **en el código generado se sustituye esa variable por el offset de la misma** , este offset es conocido por el ensamblador (conoce todas las longitudes definidas en el segmento de datos) y es igual a 64H o 100D porque antes de definir esta área se definió una de 50 palabras o 100 bytes.

La variable DIRAREA1 contiene el offset de AREA1, pero cuando se la utiliza para direccionar, el ensamblador no hace referencia a su contenido sino a su desplazamiento.

También es bueno recordar que las instrucciones que tengan operandos variables y dato son aceptadas por el ensamblador. Por ejemplo:

```
MOV NUMERO,0004H
MOV AREA1[DI],09H
```

4.14 Operadores en sentencias fuente

Un operador es un modificador que se usa en el campo de operandos de una sentencia ensamblador. Se puede utilizar mas de un operador y combinaciones entre si en una sentencia.

Hay cinco clases de operadores:

Aritméticos	Operan sobre valores numéricos
Lógicos	Operan sobre valores binarios de bit a bit.
Relacionales	Compara dos valores numéricos o direcciones de memoria del mismo segmento y produce: 0 - Si la relación es falsa. FFFF - Si la relación es verdadera.
De retorno de valores	Son operandos pasivos que suministran información acerca de las variables y de las etiquetas del programa.
De atributos	Permiten redefinir el atributo de una variable o de una etiqueta. Los atributos para variables son: BYTE = Byte. WORD = Palabra. DWORD = Doble palabra (4 bytes). TBYTE = 10 Bytes. Los atributos de etiquetas son:

NEAR = Se puede referenciar desde dentro del segmento en que esta definida.

FAR = Se puede referenciar desde fuera del segmento en que esta definida.

4.15 Operadores aritméticos

Operador	Función	Ejemplo
+	Suma dos constantes	SUM EQU NUM1+NUM2
-	Resta dos constantes	RESTA EQU NUM1-NUM2
*	Multiplica dos constantes	PROD EQU NUM1*NUM2
/	Divide dos constantes	DIV EQU NUM1/NUM2
SHL	Desplaza a izquierda una cantidad de bits	NUM3 EQU NUM1 SHL 2
SHR	Desplaza a derecha una cantidad de bits	NUM3 EQU NUM2 SHR 3

4.15.1 Operadores lógicos

Operador : AND
Formato : Valor1 AND Valor2
Función : Producto lógico de Valor1 y Valor2
Ejemplos :

```
BINARIO EQU 00110100b AND 11010111b ; 00010100b
HEXA    EQU 34H AND 0D7H            ; 14H
```

Operador : OR
Formato : Valor1 OR Valor2
Función : Suma lógica de Valor1 y Valor2
Ejemplos :

```
BINARIO EQU 00110100b OR 11010111b ; 11110111b
HEXA    EQU 34H OR 0D7H            ; F7H
```

Operador : NOT
Formato : NOT Valor
Función : Obtiene el complemento de Valor
Ejemplos :

```
BINARIO EQU NOT 00110100b       ; 11001011b
HEXA EQU NOT 34H                ; 0CBH
```

4.15.2 Operadores relacionales

Operador : EQ (Equal)
Formato : Operando1 EQ Operando2
Función : Verdad (FFFF) si los operandos son iguales
Ejemplo :

```
VALOR EQU 20H
MOV AX,VALOR EQ 20H  ; AX=FFFFH
```

Operador : NE (Not Equal).
Formato : Operando1 NE Operando2.
Función : Verdad si los dos operandos son distintos.
Ejemplo :

```
VALOR EQU 20H
MOV AX,VALOR NE 20H  ; AX=0
```

4.15.3Operadores de retorno de valores

Operador : SEG
Formato : SEG variable o SEG etiqueta
Función : Devuelve el valor del segmento de la variable o de la etiqueta
Ejemplo :

```
MOV AX,SEG TABLA     ; AX=Segmento de TABLA
```

Operador : OFFSET
Formato : OFFSET variable u OFFSET etiqueta
Función : Devuelve el desplazamiento de la variable o de la etiqueta
Ejemplo :

```
MOV DI,OFFSET TABLA  ; DI=Desplazamiento de TABLA
```

Operador : LENGTH
Formato : LENGTH variable
Función : Devuelve el número de unidades (bytes o palabras) reservadas a partir de variable.
Ejemplo :

```
NUM DW 20 DUP(0)
MOV AX,LENGTH NUM   ; AX=20
```

Solo se aplica si la variable se definió con DUP.

4.15.4Operadores de atributos

Operador : PTR
Formato : Tipo PTR expresión
Función : Redefine el atributo de tipo (BYTE, WORD, DWORD, QWORD, TBYTE) o el atributo de distancia (NEAR o FAR) de un operando en memoria.
Tipo : Nuevo atributo.
Expresión : Identificador cuyo atributo se va a sustituir.

Ejemplo 1 :

```
TABLA DW 100 DUP(?)
PRI_BYTE EQU BYTE PTR TABLA ;       Asigna nombre al primer byte.

QUINTO_BYTE EQU BYTE PTR PRI_BYTE+4
                    ;       Asigna nombre al segundo byte.
```

Ejemplo 2 :

```
Primer segmento:

EMPEZAR:    MOV AX,100

            JMP EMPEZAR     ;  Salto en el mismo segmento.

Segundo segmento:

LEJANO EQU FAR PTR EMPEZAR

        JMP LEJANO      ;   Salto al primer segmento a la etiqueta empe-
                            zar.
```

Operadores : DS:, ES:, SS:

Formato : Reg_segmento:etiqueta o Reg_segmento:variable o Reg_segmento:expresión de dirección.

Función : Sustituye el atributo de segmento de una etiqueta, variable o expresión de dirección. Genera como código un prefijo de un byte.

El microprocesador supone:

DS si el desplazamiento se expresa con BX, SI o DI.
SS si el desplazamiento se expresa con SP o BP.

Si el registro especificado coincide con el registro de segmento por defecto, no se genera el byte de prefijo.

Con la directiva ASSUME se puede especificar el registro de segmento asociado a un determinado segmento, que se insertara como prefijo en las instrucciones cuyos operandos hagan referencia al segmento, a menos que coincida con el registro de segmento por defecto (en cuyo caso no se incluye).

Ejemplos:

```
MOV AX,ES:[BP]      ; El ES sustituye al ES para calcular la dirección de
                      memoria.

MOV AL,[17H]            ; Genera error pues necesita el prefijo de
                          segmento.
```

4.16 El juego de instrucciones

Para aprender el funcionamiento de cada una de ellas, vamos a dividirlas en siete grupos:

- Instrucciones de transferencia de datos.
- Instrucciones aritméticas.
- Instrucciones de manejo de bits.
- Instrucciones de transferencia de control.
- Instrucciones de manejo de cadenas.
- Instrucciones de interrupción.
- Instrucciones de control del microprocesador.

A continuación se describen las principales instrucciones dentro de cada grupo:

4.16.1 Instrucciones de transferencia de datos

MOV
LEA
PUSH
POP

Instrucción: MOV
Formato: MOV destino,fuente
Descripción: Transfiere un byte o una palabra desde el operando fuente el operando destino.
Lógica: destino=fuente
Ejemplo: MOV AX,[DI]
Observaciones: No se puede utilizar el registro CS como destino.
No se puede mover un valor inmediato a un registro de segmento. Debe utilizar un registro intermedio.
Instrucción: LEA
Formato: LEA destino,fuente
Descripción: Transfiere el desplazamiento del operando fuente al operando destino. El operando destino debe ser un registro de 16 bits pero no un registro de segmento.
Lógica: destino=desplazamiento de fuente
Ejemplo: LEA DI,MENSAJE

Instrucción: PUSH
Formato: PUSH fuente
Descripción: Decrementa el puntero de la pila en 2 y luego transfiere la palabra especificada en el operando fuente a lo alto de la pila.
El registro CS no se puede especificar como operando fuente.

Lógica: SP=SP-2
SS:[SP]=fuente
Ejemplo: PUSH DX

Instrucción: POP
Formato: POP destino
Descripción: Transfiere el elemento que esta en lo alto de la pila al operando destino (palabra) y luego incrementa en dos el puntero de pila.
El registro CS no se puede especificar como destino.
Lógica: destino=SS:[SP]
SP=SP+2
Ejemplo: POP DX

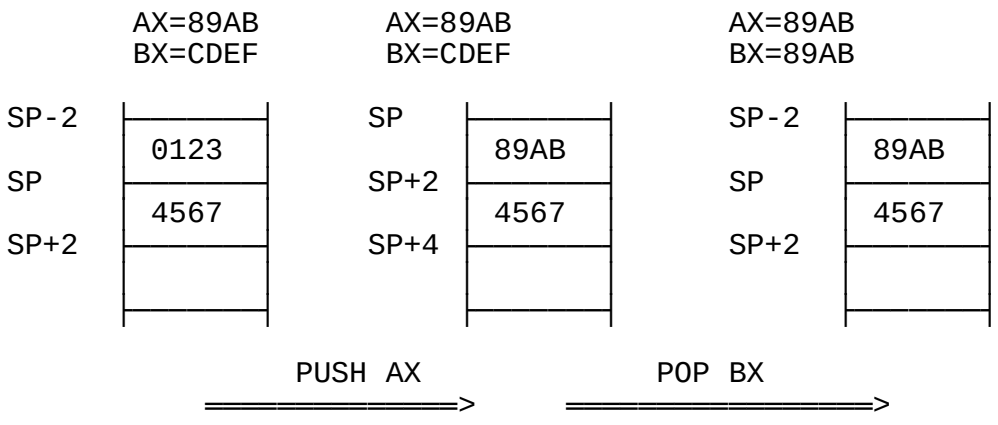
4.16.1.1 Funcionamiento de la pila

La pila es un área de memoria que se utiliza como espacio temporal para almacenar direcciones y datos. Cada elemento de la pila es una palabra (16 bits).

La pila se direcciona mediante SS:[SP]. El registro SP apunta siempre al último elemento depositado en la pila (el elemento más alto). El crecimiento de la pila es en el sentido decreciente de memoria.

Al hacer PUSH, SP debe apuntar a la posición inmediatamente superior (para lo cual hace SP=SP-2) y a continuación mueve el contenido de la palabra especificada por el operando fuente a SS:[SP].

Al hacer POP, como SP apunta al último elemento depositado sobre la pila, se realiza el movimiento de la palabra SS:[SP] al operando destino. A continuación, actualiza el puntero de la pila, para indicar el ultimo elemento de la pila (SP=SP-2). La figura siguiente ilustra el funcionamiento de estas instrucciones.



Las instrucción PUSHF pone en el stack el contenido del registro de flags y la instrucción POPF coloca la palabra apuntada por SS:[SP] en el registro de flags.

Si deseo recuperar en AX el registro de flags debo hacer:

```
PUSHF
P0P  AX
```

Si deseo colocar en el registro de flags del registro AX debo hacer:

```
PUSH  AX
P0PF
```

4.16.2Instrucciones de manejo de bits

AND
OR
NOT
SHL
SHR
ROR
ROL

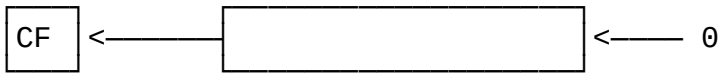
Instrucción: AND
Formato: AND destino,fuente
Descripción: Realiza un AND lógico bit a bit entre los operandos.
Lógica: destino=(destino AND fuente)

Instrucción: OR
Formato: OR destino,fuente
Descripción: Realiza un OR lógico bit a bit entre los operandos.
Lógica: destino=(destino OR fuente)

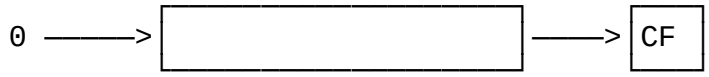
Instrucción: NOT
Formato: NOT destino
Descripción: Realiza un complemento bit a bit del operando destino.
Lógica: si destino es tipo byte
destino=FFh-destino
si destino es tipo palabra
destino=FFFFh-destino

Instrucción: SHL
Formato: SHL destino,contador
Descripción: Desplaza a izquierda la cantidad de bits de destino especificada en el segundo operando.

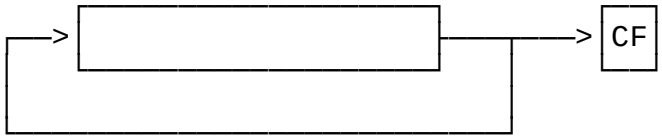
Lógica: temp=contador
Mientras temp<>0
CF=bit superior de destino
destino=destino*2
temp=temp-1



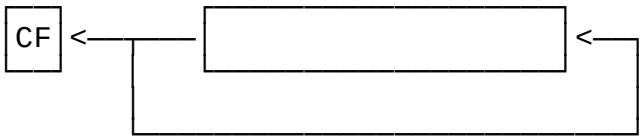
Instrucción: SHR
Formato: SHR destino,contador
Descripción: Desplaza a derecha la cantidad de bits de destino especificada en el segundo operando.
Lógica: temp=contador
Mientras temp<>0
CF=bit inferior de destino
destino=destino/2
temp=temp-1



Instrucción: ROR
Formato: ROR destino,contador
Descripción: Rota a derecha la cantidad de bits especificada en el contador
Lógica: temp=contador
Mientras temp<>0
CF=bit inferior de destino
destino=destino/2
bit superior de destino=CF
temp=temp-1



Instrucción: ROL
Formato: ROL destino,contador
Descripción: Rota a derecha la cantidad de bits especificada en el contador
Lógica: temp=contador
Mientras temp<>0
CF=bit superior de destino
destino=destino*2 + CF
temp=temp-1



4.16.3Instrucciones aritméticas

Aunque todas las operaciones se realizan en binario, las instrucciones aritméticas operan sobre:

- Números binarios (Con signo o sin signo) de 8 y 16 bits.
- Números decimales sin signo (empaquetados o no).

  Empaquetados: Dos dígitos BCD por byte.
  Desempaquetados: Un dígito BCD por byte.

4.16.3.1 Operaciones aritméticas

Las instrucciones aritméticas del microprocesador están limitadas en cuanto a precisión (número de bits):

OPERACION	OPERANDO 1	OPERANDO 2	RESULTADO
suma	8/16 bits	8/16 bits	8/16 bits
resta	8/16 bits	8/16 bits	8/16 bits
multiplicación	8/16 bits	8/16 bits	16/32 bits
división	16/32 bits	8/16 bits	8/16 bits

El número máximo que puede obtenerse como resultado de una multiplicación es:

FFFFFFFFh = 4294967295

Se plantea entonces el tema de la posibilidad de realizar cálculos aritméticos sin limitar el número de bits o signos, para ello existen tres tipos de alternativas:

- Trabajar con números binarios
- Trabajar con números decimales empaquetados
- Trabajar con números decimales desempaquetados

De estas tres formas de representación numérica, la más cómoda a efectos de cálculos aritméticos es la de los números desempaquetados por dos razones:

- La programación es más sencilla

- El resultado de la operación esta prácticamente preparado para ser escrito en la pantalla. Para ello basta sumar 30H a cada dígito antes de escribirlo y así se consigue el código ASCII del dígito.

Dígito	ASCII
00	30
01	31
02	32
03	33
05	35
06	36
07	37
08	38
09	39

Los números decimales de muchos dígitos se construyen encadenando juntos varios números de 1 byte, donde cada byte se considera como un dígito.

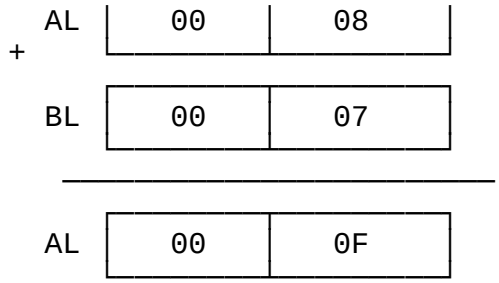
En este material solamente veremos la operación de suma, para más información acerca de las otras operaciones consulte la bibliografía.

4.16.4Instrucciones de suma

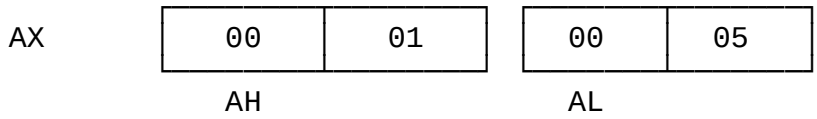
Instrucción: ADD
Formato: ADD destino,fuente
Descripción: Suma los dos operandos y almacena el resultado en destino. Realiza una suma bit a bit.
Lógica: destino=destino+fuente

Instrucción: AAA
Formato: AAA
Descripción: Corrige el resultado en AL de una suma de dos números decimales desempaquetados, convirtiéndolo en un valor decimal desempaquetado.
Lógica: Si nibble inferior de AL >9
AL=AL+6
AH=AH+1
CF=1
nibble superior de AL=00h

Ejemplo 1: Se desea sumar dos dígitos decimales desempaquetados, el primero se encuentra en AL y es igual a 8, el segundo se halla en BL y es igual a 7.



Se obtiene un byte que no representa a un dígito decimal desempaquetado, este nuevo número debe representarse en dos bytes y para ello se utiliza la instrucción AAA. Luego de ejecutar AAA el registro AX tendrá los siguientes valores:



Se debe tener en cuenta que AAA solo trabaja con el registro AX y pone CF=1 si es que desempaqueto y CF=0 si no lo hizo.

Ejemplo: Se desea sumar dos áreas de memoria de N bytes (N en CX), cada área contiene N dígitos decimales desempaquetados. Colocar el resultado en una tercer área.

```
(* inicio de la rutina *)
        mov dh,0      (*Acarreo inicial)
        mov cx,N
        lea di,area1
        lea si,area2
        lea bx,area3
@inicio:    mov dl,dh     (* Guardo el ultimo carry *)
        mov al,[di]
        add al,[si]
        aaa
        jc @carry
        mov dh,0
        add al,dl
        aaa
        jc @carry1
        jmp @seguir
carry:  mov dh,1
        add al,dl
        jmp @seguir
```

```
carry1:     mov dh,1    (* Al sumar el carry de la anterior etapa, se puede
generar un nuevo carry *)
seguir:     mov [bx],al
        inc di
        inc si
        inc bx
        dec cx
        cmp cx,0
        jne @inicio
        mov [bx],dh
 (* fin de la rutina *)
```

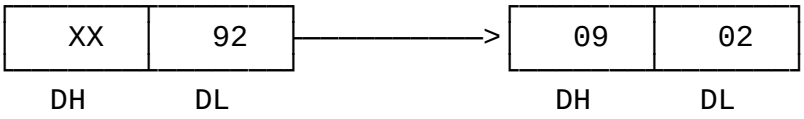
Si se tratara de números empaquetados tendremos que:

1. Desempaquetar el número
2. Efectuar la operación
3. Empaquetar el número

Supongamos que en DL tenemos almacenados dos dígitos empaquetados y deseamos desempaquetarlos en DX, tendremos que hacer:

```
MOV DH,DL
OR  DL,00001111b
SHR DH,4
```

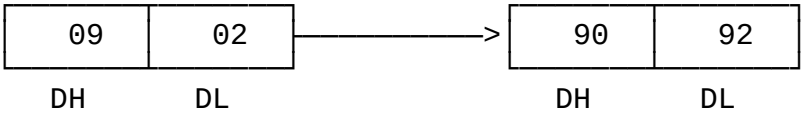
Ejemplo:



Y para empaquetar:

```
SHL DH,4
ADD DL,DH
```

Ejemplo:



4.16.5Instrucciones de transferencia de control

Transfieren el control a otro punto del programa y se las puede dividir en tres grupos:

INCONDICIONALESTransfieren el control de modo incondicional.

JMP
CALL
RET

Instrucción: JMP
Formato: JMP dirección
Descripción: Transfiere el control a la dirección especificada por el operando. La bifurcación puede ser directa (si operando es una etiqueta) o indirecta (si operando es una dirección).

Lógica:
```
si bifurcación a distinto segmento
  CS=segmento
  IP=desplazamiento
si no
  si bifurcación directa
    IP=desplazamiento
  si no
    IP=IP+desplazamiento  (con signo)
```

Instrucción: CALL
Formato: CALL destino
Descripción: Salta a un procedimiento fuera de línea, salvando previamente en la pila la dirección de la instrucción siguiente, para poder volver a esta instrucción una vez ejecutado el procedimiento.
El procedimiento llamado puede estar en el mismo segmento (NEAR) o en otro segmento (FAR).

Lógica:
```
Si distinto segmento
  SP=SP-2
  CS -> pila
  CS=segmento destino
SP=SP-2
IP -> pila
IP=desplazamiento destino
```

Instrucción: RET
Formato: RET
Descripción: Retorna de un procedimiento previamente invocado por CALL, utilizando como retorno la dirección salvada en la pila por CALL, que corresponde a la instrucción siguiente a dicha sentencia CALL.
Lógica: IP <- pila
SP=SP+2
Si distinto segmento
CS <- pila
SP=SP+2

CONDICIONALES Transfieren el control si se cumple una condición determinada.

Nombre	Descripción	Condición
JE	Salte si es igual	ZF=1
JNE	Salte si no es igual	ZF=0
JL	Salte si es menor	ZF<>OF
JG	Salte si es mayor	ZF=0 y SF=OF
JLE	Salte si es menor o igual	SF<>OF o ZF=1
JGE	Salte si es mayor o igual	SF=OF

En todos los casos de la tabla el formato de llamada es el mismo:

Ejemplo: JE desplazamiento

Donde desplazamiento puede ser una etiqueta o dirección a la cual salta si se cumple la condición (ZF=1 en este caso)

ITERATIVAS Permiten la ejecución de bucles, el número de iteraciones se determina mediante el valor del registro CX (Contador).

Instrucción: LOOP
Formato: LOOP desplazamiento
Lógica: CX=CX-1
si CX<>0
IP=IP+desplazamiento
si no
IP=offset de la siguiente instrucción

4.16.6Instrucciones de interrupción

Una interrupción es similar a la llamada a un procedimiento. Bifurca a la dirección de la rutina correspondiente. La dirección se obtiene del área de los vectores de interrupción (Ver carga del DOS).

INT
IRET

Instrucción: INT
Formato: INT Nro
Descripción: Activa el procedimiento de interrupción correspondiente a Nro. La dirección del procedimiento se consigue en la tabla de vectores de interrupción. En el área de memoria [0 - 400h].
El vector de interrupción asignado a Nro se compone de dos palabras, la primera es el offset y la segunda el segmento del procedimiento de interrupción.
Lógica: SP=SP-2
banderas -> pila
IF=0
TF=0
SP=SP-2
CS -> pila
CS=(Nro*4)+2
SP=SP-2
IP -> pila
IP=(Nro*4)

Instrucción: IRET
Formato: IRET
Descripción: Devuelve el control a la dirección de retorno salvada en la pila por una operación de interrupción previa y restaura los registros de banderas.
IRET se utiliza para finalizar un procedimiento de interrupción. Es equivalente a la instrucción RET en un procedimiento, solo que IRET también recupera las banderas.
Lógica: IP <- pila
SP=SP+2
CS <- pila
SP=SP+2
Banderas <- pila
SP=SP+2

4.16.7Instrucciones de control del microprocesador

Permiten modificar el estado de las banderas CF, IF, DF del registro de estado.

Nombre	Descripción
CLC	Borrar bandera de acarreo (CF=0)
CLD	Borrar bandera de dirección (DF=0)
CLI	Borrar bandera de interrupción (IF=0)
CMC	Complementar bandera de acarreo
STC	Poner bandera de acarreo a 1
STD	Poner bandera de dirección a 1
STI	Poner bandera de interrupción a 1

4.16.8Instrucciones de cadena

Las instrucciones de cadena no se ven en este material, para conseguir información de estas consulte la bibliografía.

Consideración:

Las instrucciones que tienen el operando destino en memoria y el operando fuente inmediato merecen una consideración especial, por ejemplo:

MOV [DI],04H Mueve a la dirección especificada por [DI] un byte con el valor 04H.

Para poder mover una palabra debo utilizar el operador WORD PTR, entonces la instrucción quedaría:

MOV WORD PTR [DI],04H Mueve a la dirección especificada por [DI] la palabra 0004H

Esto no es necesario si aparece un operando registro pues es el tamaño del registro el que da la cantidad de bytes a utilizar. Por ejemplo:

MOV AX,04h Correcto, muevo un byte a un registro de dos bytes, la parte alta del registro queda con el valor 00h.
MOV AH,04H Correcto, muevo un byte a un registro de un byte.
MOV AH,0F40H Es erróneo pues muevo una palabra a un registro de un byte.

5 ANEXO: MICROPROGRAMACION

5.1 Introducción

Como sabemos, un sistema se subdivide en distintos niveles de trabajo. Estos son:

Nivel de Lenguaje de Alto Nivel
ò
Nivel de Lenguaje Ensamblador
ò
Nivel de Sistema Operativo
ò
Nivel de Lenguaje Máquina
ò
Nivel de Microprogramación
ò
Nivel de Lógica Digital

Una instrucción de lenguaje ensamblador tiene una relación uno a uno con el nivel de lenguaje máquina y a su vez una instrucción en lenguaje máquina es interpretada por el procesador y ejecutada.

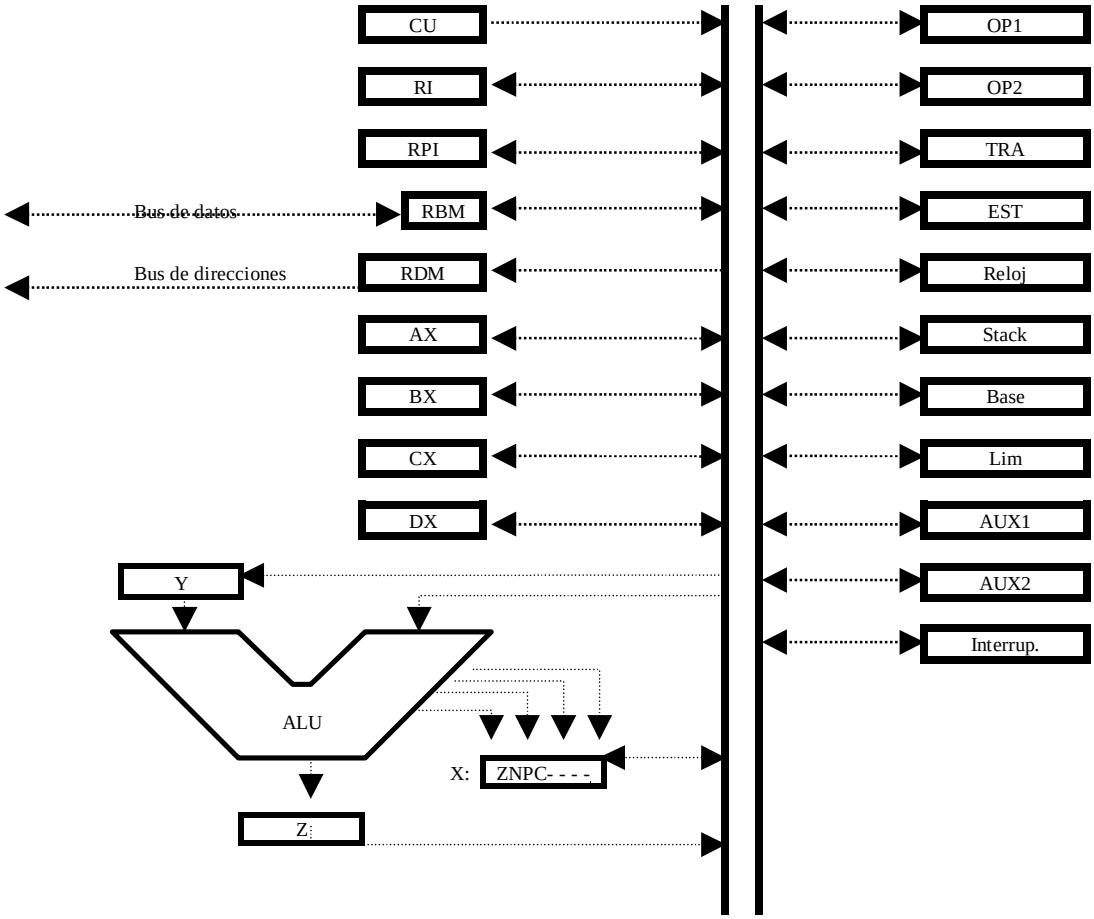
Para ello ira a buscar a la memoria principal una instrucción cargada previamente, la examinará, la ejecutará e irá a buscar la siguiente instrucción, a menos que la ejecución de la instrucción haya modificado la dirección de búsqueda de la próxima instrucción (bifurcación de control).

Para ejecutar una instrucción, el procesador podría descomponer la operación que debe ejecutar en tareas o pasos más pequeños; al ser menor cada tarea es más fácil realizarla. Si la Unidad de Control usa esta técnica, a cada una de las tareas en que se descompone una instrucción se la llamará microinstrucción. Un conjunto de microinstrucciones combinadas lógicamente para implementar una instrucción se llama microprograma.

En este apartado se intentará dar nociones sobre el nivel de microprogramación. Comenzaremos describiendo los dispositivos internos de una CPU convencional y su funcionamiento.

5.2 Esquema en bloque de una CPU de un bus

Se definirá la arquitectura y organización de la Unidad de Control microprogramada basada en un bus. Si analizamos el contenido de la figura podemos encontrar los siguientes componentes:



CU : Unidad de control
RI : Reg. de instrucción
RPI : Reg. de próxima instrucción
EST: Reg. de estado
AUX1, AUX2: Reg. auxiliares

RBM: Reg. buffer de memoria
RDM: Reg. direc. de memoria
Y, Z, X: Reg. auxiliares internos
OP1, OP2: Reg. para operandos
ALU: Unidad aritmét. y lógica

TRA: Reg. de trabajo
AX: Reg. de uso general
BX: Reg. de uso general
CX: Reg. de uso general
DX: Reg. de uso general

5.2.1 Registros

Un registro es un dispositivo en donde se almacena información. Las operaciones que se pueden hacer con registros son dos: Ingresar una nueva información y Poner a disposición la información almacenada.

La característica más importante del registro es la longitud de palabra, que coincide con la cantidad de bits que es capaz de almacenar simultáneamente. En general, trabajaremos con registros de 16 bits (2 palabras).

5.2.2 Bus

Es el medio físico mediante el cual se transmite la información. Un bus es un conjunto de cables donde cada uno de ellos lleva la información de un bit. El valor de cada bit será 0 o 1 de acuerdo a que haya o no tensión.

Hay dos formas para transmitir la información: en Serie o en Paralelo. En Serie se transmite un bit a continuación del otro a través de una sola vía de comunicación. Por lo tanto un byte de ocho bits tardará ocho veces el tiempo de transmisión de un bit en completarse. En Paralelo se transmiten los *n* bits al mismo tiempo mediante *n* vías de comunicación; como se puede ver, un byte tardará en enviarse el tiempo de transmisión de un bit. Por lo tanto la transmisión en paralelo es más rápida que en serie. En el caso de los buses la transmisión se realiza en paralelo.

Otra característica de los buses es que pueden ser unidireccionales o bidireccionales. Esta condición se representará con la siguiente notación:

Bus unidireccional
═══════Ø

Bus bidireccional
×═══════Ø

5.2.3 Comunicación entre registros y bloques

Todos los dispositivos se comunican entre sí por medio de *n* buses. Si todos los dispositivos estuvieran conectados al bus en forma directa, estarían enviando y recibiendo información al mismo tiempo. Este envío simultáneo de información por el bus produciría la mezcla de las señales eléctricas, provocando la pérdida de la información incluyendo el posible cortocircuito o sobretensión de algún dispositivo.

Para evitar esto es necesario administrar el envío de la información en el bus, y esto se lleva a cabo utilizando compuertas que se conectan entre los dispositivos y el bus. Dichas compuertas son llaves de paso que serán habilitadas en el momento preciso, eliminando la posibilidad de pérdida de la información.

5.2.4 Compuertas

Existen distintos tipos de compuertas, que permiten o no la circulación de la información. Su funcionamiento es el siguiente: Supongamos que se quiere que la información pase de A a B. Ponemos una señal eléctrica en C, y de acuerdo con su

valor, dejará pasar la información de A a B. Es decir que C comanda cuando va a pasar la información y cuando no.

Los registros son capaces de recibir y enviar información, por ello serán necesarias dos líneas de conexión, una para cada tarea.

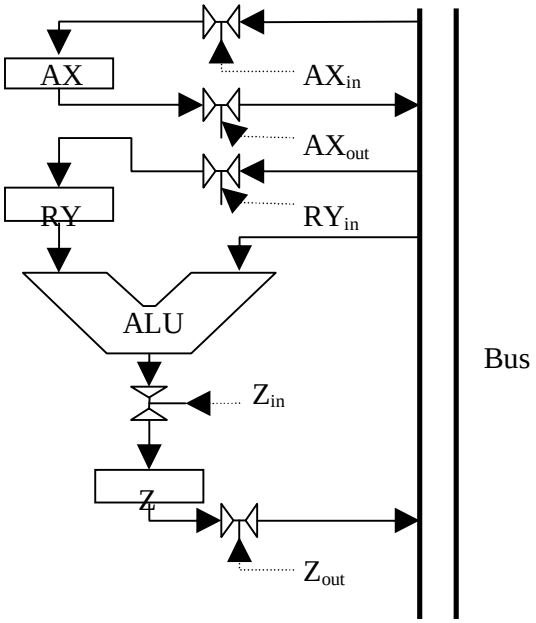
Cada línea de transmisión tendrá una compuerta asociada, la cual a menos que se la invoque, estará siempre desconectada. A los efectos de indicar que una compuerta es invocada, se utilizará la siguiente nomenclatura: para señalar a que dispositivo corresponde se usará la sigla del registro (por ejemplo RPI, AX, RBM), y para establecer el tipo de operación agregaremos un subíndice 'in' para el caso de ingreso de información y 'out' para el envío de la misma.

Por ejemplo:

PRI_{out} : El contenido del registro RPI estará disponible en el bus.

AX_{in} : El contenido del bus se almacenará en el registro AX.

En todos los casos se transmitirá simultáneamente la información de todos los bits, entre los registros y el bus (y viceversa).



5.2.5 Unidad de control microprogramada (CU)

La tarea de la Unidad de Control (Control Unit - CU) es reconocer primero la instrucción a ejecutar y luego ejecutar un microprograma asociado a esa instrucción. Dicho microprograma está conformado por una secuencia de microinstrucciones, cada una de las cuales está compuesta a su vez por microoperaciones. (tales como: habilitar una o varias compuertas, enviar señales de control como ser READ, WRITE, ADDs, AND, etcétera)

Cada microinstrucción requerirá un tiempo para ejecutarse, en ese tiempo se llevarán a cabo todas las microoperaciones. El tiempo de ejecución de una instrucción será el tiempo de ejecución de todas las microinstrucciones que componen la instrucción.

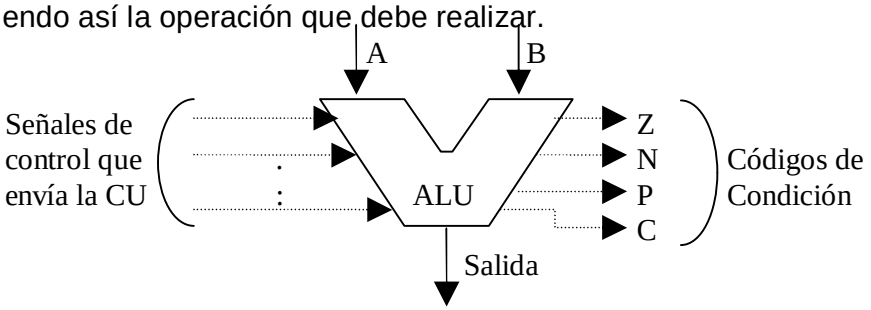
La Unidad de Control hará las siguientes tareas: ir a buscar una instrucción a memoria, decodificar la instrucción y ejecutarla. El proceso de búsqueda de una instrucción (o dato) a memoria es común a todas las instrucciones, es decir a cada microprograma.

Se puede esquematizar cada instrucción como una secuencia de microinstrucciones que a su vez están formadas por microoperaciones:

Instrucción
- $\text{microinstrucción}_1$ {microoperación_1, microoperación_2, ...}
- $\text{microinstrucción}_2$ {microoperación_1, microoperación_2, ...}
- :
- :
- $\text{microinstrucción}_n$ {microoperación_1, microoperación_2, ...}

5.2.6 Unidad aritmética y lógica (ALU)

La función de la Unidad Aritmética y Lógica (Arithmetic & Logic Unit - ALU) es la de realizar las operaciones aritméticas y lógicas, como ser la SUMA, RESTA, MULTIPLICACIÓN, DIVISIÓN, AND, OR, XOR, etcétera. Para efectuar todas las operaciones, salvo el NOT y el SHIFT, utiliza dos entradas y una salida. Es decir, realiza las operaciones sólo con dos operandos a la vez (los de entrada) y devuelve un resultado. La ALU recibe las señales de control que envía la Unidad de Control, conociendo así la operación que debe realizar.



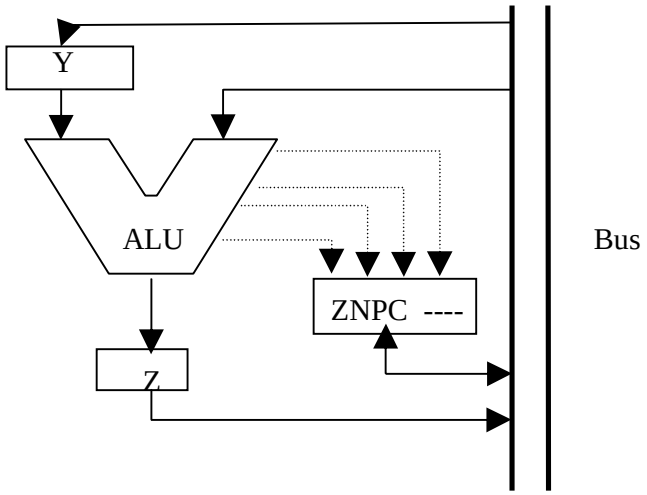
Además del resultado, la ALU retorna un código de condición de la operación realizada. Dicho código de condición está compuesto por cuatro bits y puede ser:

Z	N	P	C	Significado
1	0	0	0	El resultado de la operación aritmética es cero o el resultado de la operación lógica tiene todos bits en cero.
0	0	1	0	El resultado de la operación aritmética es positivo y sin acarreo o el resultado de la operación lógica tiene todos los bits en 1
0	0	1	1	El resultado de la operación aritmética es positivo y con acarreo
0	1	0	0	El resultado de la operación aritmética es negativo y sin acarreo o el resultado de la operación lógica contiene bits en 0 y en 1
0	1	0	1	El resultado de la operación aritmética es negativo y con acarreo

El tiempo que demanda una operación es, a lo sumo, el de una microinstrucción. Debido a que la ALU no puede retener el resultado de la operación efectuada, se la utiliza en combinación con un registro de operación auxiliar (registro Z) al cual se le habilita la compuerta de ingreso en el momento de efectuar la operación.

Para poder trabajar con dos operandos, uno de ellos tendrá que ser previamente almacenado en un registro auxiliar (Registro Y), mientras que el otro estará presente en el bus. Para realizar una operación se habilitará la salida del registro Y y la del otro operando que estará almacenado en otro registro.

Para almacenar los códigos de condición usaremos un registro auxiliar (registro X) de 8 bits, en el cual los primeros 4 bits se destinarán a los códigos Z, N, P, C respectivamente.



Supongamos que queremos realizar una suma, y que los sumandos están almacenados uno en el registro Y y el otro en el registro AX, el resultado de dicha operación quedará en el registro Z. Para ello la microinstrucción a utilizar sería:

$$Z = RY + AX \quad \equiv \quad ADDs, AX_{out}, Z_{in}$$

Nota: las señales de control como ADDs que utilizan dos operandos habilitan la compuerta de salida de RY, es decir, dan simultáneamente las órdenes a la ALU y a RY_{out}.

Vemos que la anterior es una microinstrucción compuesta por tres microoperaciones que se realizan simultáneamente que son:

ADDs: Señal de control a la ALU y habilitación de la salida de información a Y_{out}.

AX_{out}: Habilitación de la salida de información del registro AX.

Z_{in}: Habilitación de la entrada de información al registro Z.

A continuación indicaremos las operaciones que puede realizar la ALU:

Operación	Microoperación	Resultado
Suma	ADDs, ADDc, ADDsa, ADDca	Bus + [Y]
Resta	SUBs, SUBc, SUBsa, SUNca	Bus - [Y]
Multiplicación	MULTs, MULTc	Bus * [Y]
División	DIVs, DIVc	Bus / [Y]
Decalaje	$SHIFT_l$, $SHIFT_r$	Decala Bus
AND Lógico a bit	AND	Bus ^ [Y]
OR Lógico a bit	OR	Bus v [Y]
XOR Lógico a bit	XOR	Bus v [Y]
NOT Lógico a bit	NOT	~ Bus

donde los subíndices indican :

s	Operación aritmética sin signo y sin considerar o modificar el acarreo
c	Operación aritmética en notación complemento y sin considerar o modificar el acarreo
sa	Operación aritmética sin signo y considerando y modificando el acarreo
ca	Operación aritmética en notación complemento y considerando y modificando el acarreo

En una operación de suma con acarreo, si el bit de acarreo está en uno, se incrementa el resultado y en una de resta, con las mismas condiciones, se decrementa.

5.2.7 Bus de direcciones y de datos

Hay dos registros que contendrán la información para comunicarse con el medio externo, estos son:

RBM : Registro base de memoria
RDM : Registro de dirección de memoria

El primero contendrá la información que fue recibida o que va a ser transmitida y el segundo contendrá la dirección desde donde la información será recibida o a donde será enviada. El RBM estará conectado al bus de datos y el RDM al de direcciones.

Para efectuar una lectura de memoria deberá primero cargar la dirección de memoria en el RDM y luego enviar una señal de pedido de lectura a memoria y esperar a que se realice la operación. Una vez que se recibe la señal de que la operación se ha completado el contenido de la lectura solicitada estará almacenado en el RBM.

Como el acceso a memoria se hará de a un byte (una palabra), dicho bus contendrá 8 “cables” y el RBM almacenará sólo 8 bits. Debido a que estamos trabajando con un mapa de memoria de 0000h a FFFFh necesitamos 2 bytes para poder representar todas las direcciones, por lo cual el bus de direcciones será de 16 bits y el registro RDM también.

5.2.8 Señales

Las señales brindan información acerca del control de lo que se debe hacer. Como vimos antes, la Unidad de Control envía señales a la ALU sobre qué operación debe efectuar. Además envía señales a la memoria sobre si debe hacer una operación de lectura o escritura. Hay otras señales externas como ser la que indica la finalización de la operación con la memoria.

5.3 Funcionamiento de la CPU

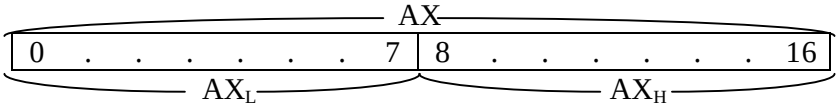
En esta sección analizaremos el funcionamiento de la CPU en cuanto a la ejecución de instrucciones.

5.3.1 Transferencia de la Información

La transferencia se puede hacer de dos formas: en una se transfiere el contenido de un registro entero (16 bits) mientras que en la otra se puede transmitir medio

registro (8 bits); en este caso se podrá trabajar con los primeros 8 bits o con los últimos 8.

En el primer caso nos referiremos al registro como fue expresado anteriormente y en el segundo se agregarán los subíndices 'L' y 'H' según si se quiere operar con los primeros 8 bits o con los últimos ('L' = Low, 'H' = High).



Veamos cómo se puede hacer para copiar información de un registro a otro, por ejemplo del AX al BX. Se habilita la compuerta de salida de AX, haciéndose de esta manera la información presente en el bus, y se abre la compuerta de entrada de BX, permitiendo de esta manera que la información del bus quede almacenada.

Todo esto debe ocurrir al mismo tiempo debido a que en caso contrario la información de AX no se mantendrá en el bus y BX tomará cualquier valor. El tiempo en el cual se efectuarán estas dos microoperaciones es el de una microinstrucción. En este caso la microinstrucción a realizar será:

AX_{out} , BX_{in}

Como vemos, todas las microoperaciones que componen una microinstrucción se escriben en una misma línea sin importar el orden dentro de la microinstrucción, por lo cual es equivalente escribir: AX_{out}, BX_{in} a BX_{in}, AX_{out}.

Ahora analicemos el caso de copiar primero de AX a BX y luego de AX a CX:

AX_{out}, BX_{in}
AX_{out}, CX_{in}

Como vemos para realizar esta tarea se necesitan dos microinstrucciones. Otras formas de obtener el mismo resultado son:

AX_{out}, BX_{in} BX_{out}, CX_{in}	Equivalente a	AX_{out}, CX_{in} AX_{out}, BX_{in}	Equivalente a	AX_{out}, CX_{in} CX_{out}, BX_{in}

Supongamos que se quisieran ahorrar microoperaciones, se podría haber planteado:

AX_{out}, BX_{in}
CX_{in}

¿Que pasaría en este caso? BX tendría el contenido de AX, pero CX tendría un valor indefinido. ¿Por que? La respuesta es que el contenido de AX se mantendrá en el bus únicamente por el tiempo que tarde la primer microinstrucción, una vez que esta haya concluido, nadie puede asegurar cual es el contenido del bus, por lo tanto el contenido final de CX.

Veamos ahora como es posible reducir efectivamente la cantidad de microinstrucciones a realizar (y a la vez de microoperaciones). Aprovechando el momento en que la información de AX está disponible en el bus, nada nos impide almacenar esta información en BX y en CX a la vez, de la siguiente manera:

$$AX_{out},\ BX_{in},\ CX_{in}$$

Como se puede observar, la diferencia principal entre la primera forma de copiar AX a BX y CX y esta última es que en esta se ahorra el tiempo de una microinstrucción.

5.3.2 Ejemplo de una instrucción sin acceso a memoria

Aquí nos referiremos sólo a una instrucción que en su ejecución no hará acceso a memoria. La instrucción es copiar el contenido de un registro a otro.

MOV AX, BX es equivalente a [AX] ß [BX]

Primero se debe buscar la instrucción a memoria, luego decodificarla y por último ejecutarla. A nivel de microprograma será:

1. RPI_{out}, RDM_{in}
2. READ
3. $WAIT_{fmc}$
4. CLEAR Y
5. RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
6. Z_{out}, RPI_{in}
7. RBM_{out}, RI_{Lin}
8. RPI_{out}, RDM_{in}
9. READ
10. $WAIT_{fmc}$
11. CLEAR Y
12. RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
13. Z_{out}, RPI_{in}
14. RBM_{out}, RI_{Hin}

======== Decodificación ========

15. BX_{out}, AX_{in}
16. END

1. RPI_{out}, RDM_{in}

Se pasa la dirección de búsqueda de la próxima instrucción que está en el RPI al registro de dirección de memoria, el cual está conectado a la memoria por medio del bus de direcciones. En una operación de lectura/escritura la memoria va a buscar la dirección de trabajo al bus de direcciones.

2. READ
Efectúa la lectura de la memoria. La operación READ lo que hace es enviar una señal de lectura a la memoria, abrir la compuerta de salida que está entre el RDM y el bus de direcciones permitiendo así que el contenido del RDM esté en dicho bus, y abrir la compuerta de entrada que está entre el bus de datos y el RBM permitiendo que el contenido del bus de datos se almacene en el RBM.
La memoria al recibir la señal efectuará la operación de lectura tomando como dirección de búsqueda la del bus de direcciones y hará presente el contenido de dicha dirección en el bus de datos.

3. $WAIT_{fmc}$
Hasta que la operación con la memoria no se haya terminado se mantendrán las compuertas abiertas de salida del RDM y de entrada del RBM. El WAIT bloquea la CPU hasta recibir la señal que envía la memoria y entonces cierra las compuertas abiertas anteriormente. (fmc = función de memoria completada).

4. CLEAR Y
De esta manera se ponen todos los bits del registro Y en cero.

5. RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
El bit asociado al carry se pone en uno, esto implica que en una operación de suma o resta con acarreo modificará en uno el resultado. El ADDsa es una señal de control que le indica a la ALU la operación que debe realizar. La ALU tomará el contenido del registro Y y del bus, por eso simultáneamente se hace RPIout. Por lo tanto va a sumar:

Registro Y	0
Bus [RPI]	RPI
Carry	1 .
Registro Z	[RPI]+1

El resultado de la operación se almacenará en el registro Z, si la compuerta del entrada del mismo estaba habilitada. Notar que si no se abre esta compuerta el resultado de la operación se pierde.

6. Z_{out}, RPI_{in}
Se pasan el resultado de la operación anterior al RPI.

7. RBM_{out}, RI_{Lin}
El contenido del RBM se carga en la parte baja del registro de instrucción (RI_L), es decir en el primer byte.

Lo que hacen los pasos 4, 5 y 6 es incrementar el RPI en uno. El resto de los pasos efectúan la operación de búsqueda de la instrucción en memoria para luego cargarla en el registro de instrucción. Con lo que hicimos hasta ahora se cargó el primer byte del RI, por lo que falta cargar el segundo byte. Para esto se repiten los siete pasos anteriores con la diferencia que en el último de ellos se carga la parte alta del RI.

8. RPI_{out}, RDM_{in}
9. READ
10. $WAIT_{fmc}$
11. CLEAR Y
12. RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
13. Z_{out}, RPI_{in}
14. RBM_{out}, RI_{Hin}

======== Decodificación ========
En este punto se terminó la operación de búsqueda de la instrucción a memoria y se realiza la decodificación o interpretación del código de operación que dará el punto de comienzo del microprograma que ejecutará la instrucción.

15. BX_{out}, AX_{in}
Se copia el contenido de BX a AX.

16. END
Esta microoperación avisa que terminó de ejecutarse la instrucción, y la Unidad de Control arranca de nuevo para buscar la próxima instrucción.

Nota: En este ejemplo no se tuvieron en cuenta los códigos de condición para no hacer compleja la comprensión del microprograma.

5.3.3 Ciclo de Fetch

Las primeras 14 instrucciones del microprograma anterior conforman en Ciclo de Fetch de instrucciones. Como estas microinstrucciones se ejecutan por cada instrucción que la CPU ejecutará, es importante tratar de optimizar este ciclo cuanto sea posible.

Analizando nuevamente el ejemplo anterior vemos que se puede mejorar el ciclo de búsqueda (disminuir la cantidad de microinstrucciones). Las microinstrucciones 2 y 3 se pueden hacer simultáneamente ya que se puede hacer el pedido de lectura y esperar a que se complete. De la misma manera se puede poner el Registro Y en cero y en la misma microinstrucción realizar la operación de suma. Estos pasos se repiten en las microinstrucciones 9, 10, 11 y 12. Con estos cambios es ciclo de FETCH sería:

1. RPI_{out}, RDM_{in}
2. READ, $WAIT_{fmc}$
3. CLEAR Y, RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
4. Z_{out}, RPI_{in}
5. RBM_{out}, RI_{Lin}
6. RPI_{out}, RDM_{in}
7. READ, $WAIT_{fmc}$
8. CLEAR Y, RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
9. Z_{out}, RPI_{in}
10. RBM_{out}, RI_{Hin}

======== Decodificación ========

Las microinstrucciones 2 y 3 se pueden realizar al mismo tiempo debido a que no hay conflicto entre la señal de lectura, la suma y el uso del bus, quedando el ciclo de búsqueda así:

1. RPI_{out}, RDM_{in}
2. READ, $WAIT_{fmc}$, CLEAR Y, RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
3. Z_{out}, RPI_{in}
4. RBM_{out}, RI_{Lin}
5. RPI_{out}, RDM_{in}
6. READ, $WAIT_{fmc}$, CLEAR Y, RPI_{out}, $CARRY_{in=1}$, ADDsa, Z_{in}
7. Z_{out}, RPI_{in}
8. RBM_{out}, RI_{Hin}

======== Decodificación ========

Tratemos ahora de unir las microinstrucciones 2 y 3. Vemos que no hay un conflicto de de señales, pero si lo hay en el uso del bus ya que uno pretendía que en el bus estuviera el contenido del RPI y el contenido de Z, lo que carece de sentido. Un caso análogo sería tratar de juntar la microinstrucción 3 con la 4 o la 2 con la 4.

Intentaremos unir las microinstrucciones 1 y 2. En ambas el contenido del bus es el mismo, por lo tanto no hay incompatibilidad.

1. RPI_{out}, RDM_{in}, READ, $WAIT_{fmc}$, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
2. Z_{out}, RPI_{in}
3. RBM_{out}, RI_{Lin}
4. RPI_{out}, RDM_{in}, READ, $WAIT_{fmc}$, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
5. Z_{out}, RPI_{in}
6. RBM_{out}, RI_{Hin}

======== Decodificación ========

En este punto notamos que no se pueden superponer más microinstrucciones debido a que cada una utiliza una información distinta en el bus. Este microprograma demoraría 6 tiempos (de ejecución de una microinstrucción) en llevarse a cabo, pero como tiene dos operaciones con la memoria, la microoperación WAIT retardará su ejecución el tiempo que tarde la memoria en responder.

Podríamos hacer que ese retraso afecte lo menos posible y esto lo conseguiremos reubicando la microoperación $WAIT_{fmc}$ en la microoperación anterior a que se necesite la información pedida. Realizando esta última modificación el Ciclo de Fetch quedará:

1. RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
2. Z_{out}, RPI_{in}, $WAIT_{fmc}$
3. RBM_{out}, RI_{Lin}
4. RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
5. Z_{out}, RPI_{in}, $WAIT_{fmc}$
6. RBM_{out}, RI_{Hin}

======== Decodificación ========

Con esto hemos alcanzado una versión muy optimizada del Ciclo de Fetch, punto fundamental dado que el mismo se ejecutará cada vez que la CPU ejecute cualquier instrucción.

Es importante tratar de optimizar todas las instrucciones que se ejecutan (disminuyendo la cantidad de microinstrucciones necesarias) juntando microinstrucciones y/o reubicándolas, pero se debe tener el mayor de los cuidados para no cambiar el sentido de la lógica de la instrucción original.

5.3.4 Contador de microprograma

Para llevar el control sobre la microinstrucción del microprograma que se debe ejecutar, hay un contador (mPC : MicroProgram counter) dentro de la Unidad de Control que dirá el número de microinstrucción que se debe efectuar. Del análisis del Fetch se desprende que siempre comienza con un valor inicial que se irá incrementando de uno en uno con cada microinstrucción que se ejecute.

¿Que pasa una vez que se ejecutó la microinstrucción 6? Debe ejecutar la primer microinstrucción del microprograma correspondiente a la instrucción que levantó. Dicha microinstrucción tendrá un número dentro de la Unidad de Control, el cual estará relacionado con el código de operación de la instrucción.

Esto significa que después de ejecutar la microinstrucción 6, se podrá ejecutar la microinstrucción 213, o la 65, o la 187, ... de acuerdo a la instrucción levantada.

Nota: Debido a las características de nuestro assembler y para no hacer complejo el mapa de microinstrucciones de la Unidad de Control adoptaremos la notación que el comienzo de cada microprograma coincida con la microinstrucción siguiente al Fetch. Para poder efectuar bifurcaciones incondicionales o saltos tenemos la microinstrucción BRANCH *n* donde '*n*' es un número que indica la microinstrucción a ejecutar.

Después de la ejecución de un microprograma debe comenzar el Fetch de la siguiente instrucción, para lo cual habrá que poner el mPC en cero y esto lo hacemos con la microoperación END.

5.3.5 Instrucciones

Se mostrará aquí el microprograma de algunos tipos de instrucciones como ejemplo.

5.3.5.1 Instrucciones sin operandos en memoria

Las instrucciones que no usan operandos en memoria son aquellas que trabajan sólo con registros. La mayoría de las instrucciones modifican los códigos de condición.

Luego de cada operación de la ALU se modifican los códigos de condición del registro AX; cuando interese retener su valor debe ser transferido al registro EST. Como no se puede transferir medio byte (4 bits) primero se cargará el registro AX, con la parte alta del registro EST; luego se opera con la ALU, y por último se copia el contenido de AX en la parte alta de EST.

ADD CX, BX (Suma de registros)

ADD	CX BX

FETCH
======== Decodificación ========
6) EST_{Hout}, AX_{in}
7) CX_{out}, Y_{in}
8) BX_{out}, ADDsa, Z_{in}
9) AX_{out}, EST_{Hin}
10) Z_{out}, CX_{in}, END

5.3.5.2 Instrucciones con un operando en memoria

En la ejecución de este tipo de instrucciones se debe leer un operando de la memoria, para lo cual se harán dos lecturas (pasos 6 a 11) que se almacenarán en el registro OP1. Una vez obtenido el operando, se lo debe resolver, para lo cual se utilizará una microrutina, que trabajará con el registro auxiliar TRA (del que tomará el operando y en el cual dejará la dirección obtenida).

Para llamar a la microrutina se deberá hacer una bifurcación a su microinstrucción de comienzo (si suponemos que la misma empieza en la dirección 300, será

BRANCH 300), y para retornar de la misma se debe almacenar el número de la próxima microinstrucción a ejecutar en el contador de microprograma auxiliar (por ejemplo: PC_{aux}=60). Para volver, la microrutina bifurca a PC_{aux} (BRANCH PC_{aux}).

MOV ROT, DX (Carga de registro con memoria)

MOV	DX	Dir. ROT

FETCH
======== Decodificación ========
6) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
7) Z_{out}, RPI_{in}, $WAIT_{fmc}$
8) RBM_{out}, $OP1_{Lin}$
9) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
10) Z_{out}, RPI_{in}, $WAIT_{fmc}$
11) RBM_{out}, $OP1_{Hin}$
12) $OP1_{out}$, TRA_{in}, PC_{aux}=50
13) BRANCH 300
50) TRA_{out}, RDM_{in}, READ, $WAIT_{fmc}$
51) RBM_{out}, DX_{Lin}
52) TRA_{out}, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
53) Z_{out}, RDM_{in}, READ, $WAIT_{fmc}$
54) RBM_{out}, DX_{Hin}, END

5.3.5.3 Instrucciones de bifurcación

Lo que hacen estas instrucciones es modificar el RPI con el contenido de un operando.

JMP ROT (Bifurcación incondicional)

JMP	Dir. ROT

FETCH
======== Decodificación ========
6) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
7) Z_{out}, RPI_{in}, $WAIT_{fmc}$
8) RBM_{out}, $OP1_{Lin}$
9) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
10) Z_{out}, RPI_{in}, $WAIT_{fmc}$
11) RBM_{out}, $OP1_{Hin}$
12) $OP1_{out}$, TRA_{in}, PC_{aux}=80, BRANCH 300
80) TRA_{out}, RPI_{in}, END

5.3.5.4 Instrucciones con dos operandos en memoria

En la siguiente instrucción, los pasos 6 a 11 y 12 a 17 respectivamente almacenan los operandos ROT1 en OP1 y ROT2 en OP2. Los pasos 43 a 54 forman un ciclo que decrementa la cantidad de bytes a mover, leyendo y grabando el byte especificado en las direcciones OP1 y OP2. Debido a que el código de condición resultante de la ejecución de la instrucción depende del contenido de cada byte, será necesario mantener el código de cada byte.

En la microinstrucción 53 se llamará a una microrutina que está en la microinstrucción 200, la cual efectúa la tarea de almacenar el código de condición de cada byte. Esta rutina toma como parámetro el registro AUX1 y operará con el registro AUX2 en donde dejará almacenado el código de condición acumulado.

En la microinstrucción 80 se invoca a otra microrutina la cual está en la microinstrucción 250 y cuya tarea es la de tomar el contenido del registro AUX2 y determinar su código de condición, almacenándolo en la parte alta de EST.

MOVS ROT1, ROT2 (Mover strings)

MOVS	ROT1	ROT2

FETCH

======== Decodificación ========

6) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
7) Z_{out}, RPI_{in}, $WAIT_{fmc}$
8) RBM_{out}, $OP1_{Lin}$
9) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
10) Z_{out}, RPI_{in}, $WAIT_{fmc}$
11) RBM_{out}, $OP1_{Hin}$
12) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
13) Z_{out}, RPI_{in}, $WAIT_{fmc}$
14) RBM_{out}, $OP2_{Lin}$
15) RPI_{out}, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
16) Z_{out}, RPI_{in}, $WAIT_{fmc}$
17) RBM_{out}, $OP2_{Hin}$
18) $OP1_{out}$, TRA_{in}, PC_{aux}=30, BRANCH 300
30) TRA_{out}, $OP1_{in}$
31) $OP2_{out}$, TRA_{in}, PC_{aux}=40, BRANCH 300
40) TRA_{out}, $OP2_{in}$
41) CLEAR TRA, CLEAR AUX1, CLEAR AUX2
42) RI_{Hout}, TRA_{Hin}
43) CLEAR Y, TRA_{out}, $CARRY_{in=1}$, SUB_{ca}, Z_{in}
44) IF N, BRANCH 80
45) Z_{out}, TRA_{in}
46) $OP2_{out}$, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
47) Z_{out}, $OP2_{in}$, $WAIT_{fmc}$
48) $OP1_{out}$, RDM_{in}, READ, CLEAR Y, $CARRY_{in=1}$, ADDsa, Z_{in}
49) Z_{out}, $OP1_{in}$, $WAIT_{fmc}$

50) RBM_{out}, Y_{in}
51) CLEAR RBM
52) RBM_{out}, SUB_s
53) X_{out}, $AUX1_{Lin}$, PC_{aux}=54, BRANCH 200
54) BRANCH 43
80) PC_{aux}=81, BRANCH 250
81) END

La microoperación IF (línea 44) funciona de la siguiente manera. Consulta el bit de los códigos de condición del registro AX, y en el caso de coincidir hace una bifurcación a donde le indique el BRANCH. En caso contrario, continuará con la siguiente microinstrucción. El IF usa los códigos de condición Z, P, ~P, N, ~N, C y ~C. Esta microoperación no se puede ejecutar conjuntamente con otras.

6 ANEXO: SISTEMAS DE NUMERACIÓN

6.1 Sistemas de numeración y representación de la información

Las computadoras procesan datos, que pueden representar desde el saldo de una cuenta corriente, el apellido de una persona hasta el valor de la temperatura interior del transbordador espacial. De alguna manera estos datos se guardan en memoria: es decir, debe existir una forma de representar el valor $128,3; una de representar PEREZ y una de representar el valor -145,398. Estos temas son los que se tratarán a continuación. En primer lugar se dará una introducción teórica a los sistemas de numeración y luego se estudiarán las diferentes convenciones existentes para representar la información.

6.2 Sistemas de numeración no posicionales

En los sistemas de numeración no posicionales, el valor de una cifra no depende de la posición que ésta ocupe dentro del número. El ejemplo más conocido es el de los números romanos. La representación del número decimal 221 en el sistema de números romanos es:

CCXXI

La cifra C aparece dos veces en el número y siempre vale lo mismo (100), al igual que la cifra X que en sus dos apariciones vale 10.

6.3 Sistemas de numeración posicionales

En los sistemas de numeración posicionales, el valor de una cifra depende del lugar que ésta ocupe dentro del número. El más conocido es el sistema decimal. En el número decimal 221 el dígito 2 figura dos veces, pero el de la derecha representa 2 decenas mientras que el de la izquierda representa dos centenas.

Generalizando, en un sistema de numeración posicional de base *b,* la representación de un número se define a partir de la regla:

$$(\ldots a_3\, a_2\, a_1\, a_0.a_{-1}\, a_{-2}\, a_{-3} \ldots)_b = \ldots + a_2\, b^2 + a_1\, b^1 + a_0\, b^0 + a_{-1}\, b^{-1} + a_{-2}\, b^{-2} + a_{-3}\, b^{-3} + \ldots \textbf{(1)}$$

Por ejemplo, $(423.1)_6 = 4 \cdot 6^2 + 2 \cdot 6^1 + 3 \cdot 6^0 + 1 \cdot 6^{-1}$. Cuando *b* es diez y los a_i se eligen del conjunto de dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 se trata del sistema decimal; y en este caso el subíndice *b* de la expresión (1) puede omitirse.

Las generalizaciones más simples del sistema decimal se obtienen cuando *b* es un entero no negativo mayor a 1 y cuando los a_i pertenecen al conjunto de enteros en el rango $0 \leq a_i < b$. Así, cuando *b* es 2 se obtiene el sistema de numeración binario, cuando *b* es 8 el octal y cuando *b* es 16 el hexadecimal. Pero en general, se podría elegir cualquier *b* distinto de cero, y los a_i de cualquier conjunto de números, obteniendo sistemas muy interesantes (ver [1]Knuth para más información).

El punto que aparece entre los dígitos a_0 y a_{-1} se denomina punto fraccionario. Cuando *b* es 10 se lo llama punto decimal y cuando *b* es 2, punto binario.

Los a_i se denominan *dígitos* de la representación. Un dígito a_j se dice que es más significativo que un dígito a_k si $j > k$. Así, el dígito del extremo izquierdo es comunmente llamado el *dígito más significativo* y el del extremo derecho el *dígito menos significativo.* En el sistema binario estándar los dígitos binarios, usualmente llamados *bits* (ver definición en Representación de la Información), son el 1 y el 0*;* y en el sistema hexadecimal estándar, los dígitos 0 a 15 se representan con el conjunto:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F.

Si $w = (a_k \ldots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3} \ldots a_{-l})_b$, se llamará parte entera de w y se notará w_e al número formado por los dígitos que se encuentran a la izquierda del punto fraccionario.

$$w_e = (a_k \ldots a_3 a_2 a_1 a_0)_b$$

Se llamará parte fraccionaria de w y se la notará w_f al número formado al número formado por el punto fraccionario y los dígitos a su derecha.

$$w_f = (.a_{-1} a_{-2} a_{-3} \ldots a_{-l})_b$$

Por definición, $w = w_e + w_f$

6.3.1 Aritmética de base b

Las operaciones entre números de base *b* se lleva a cabo conforme las tablas de adición y multiplicación correspondientes a dicha base.

Ejemplos:

Aritmética de base 4:

+	0	1	2	3
0	0	1	2	3
1	1	2	3	10
2	2	3	10	11
3	3	10	11	12

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	10	12
3	0	3	12	21

123_4
$+ \underline{201_4}$
330_4

123_4
$x \underline{201_4}$
$123_4 + 31200_4 = 31323_4$

6.3.2 Cambio de base

Suponer que se quiere convertir un número de base *b* a base *p.* La mayoría de los algoritmos existentes se basan en multiplicaciones y divisiones y utilizan alguno de los cuatro esquemas que se describirán a continuación. Dos de ellos se utilizan para convertir números enteros y los otros dos para convertir fracciones.

Dado un número $w_b = (w_e)_b + (w_f)_b$, si se desea encontrar su representación en base *p* habrá que hallar $(w_e)_p$ mediante alguno de los métodos de cambio de base para enteros y $(w_f)_p$ mediante alguno de los métodos para fracciones. Finalmente,

$$w_p = (w_e)_p + (w_f)_p$$

6.3.3 Conversión de números enteros

Existen dos métodos para cambiar de base números enteros. Uno que utiliza aritmética de base *b* y otro que utiliza aritmética de base *p*.

6.3.3.1 Método de división por p usando aritmética de base b

Dado un número entero *u*, su representación en base *p,* $(v_m \ldots v_1 v_0)_p$ se puede obtener de la siguiente manera:

$v_0 = u \bmod p$ (mod devuelve el resto de la división entera)
$v_1 = \lfloor u / p \rfloor \bmod p$ ($\lfloor u / p \rfloor$ devuelve la parte entera de la división real)
$v_2 = \lfloor \lfloor u / p \rfloor / p \rfloor \bmod p$

…

terminando cuando (… ⌊⌊*u* / *p*⌋ / *p*⌋… /*p*)= 0

Ejemplo: Suponer que se quiere convertir el número decimal 478 a base 8. En este caso *b* = 10 y *p* = *8.* Se usa aritmética decimal y se genera una tabla de dos columnas con los cocientes y restos de las sucesivas divisiones por *p* (en este ejemplo, divisiones por 8):

Cociente	**Resto**
478	
59	***6***
7	***3***
0	***7***

Entonces, 478 = $(736)_8$

6.3.3.2 Método de multiplicación por b usando aritmética en base p

Dado un número *u,* si su representación en base *b* es $(u_m \ \ldots \ u_1u_0)_b$, su representación en base *p* se puede hallar evaluando el polinomio $u_mb^m + \ldots + u_1b^1 + u_0$ usando aritmética de base *p*. La evaluación del polinomio puede resultar tediosa si *m* es grande, pero se puede simplificar usando la regla de Horner para evaluación de polinomios:

$$u_mb^m + \ldots + u_1b^1 + u_0 = ((\ldots (u_mb + u_{m-1})b + \ldots)b + u_1)b + u_0)$$

Ejemplo: Convertir el número decimal 478 a base 8. En este caso *b* = 10 y *p* = *8.* Ahora utilizaremos aritmética de base 8. Para ello, primero deberemos expresar *b* en base 8, y luego comenzar con la evaluación.

10 = $(12)_8$

Para llevar a cabo la evaluación del polinomio mediante el método de Horner, se pueden escribir los dígitos del número original en una columna, comenzando por el más significativo en la primera fila. Cada uno de los dígitos será escrito en base *p*. Como el ejemplo pide convertir a base 8, el dígito 8 se escribirá como $(10)_8$. En una columna “resultado” se construirá la representación en base *p*. Para comenzar, en la primer fila de la columna “resultado” se copia el dígito de esa fila. Luego, el resultado de la fila *n* se obtendrá sumando al dígito de la fila *n* el producto del resultado de la fila *n-1* por *b.* Finalmente, el resultado de la última fila guardará el número buscado.

Nota: En la siguiente tabla se han omitido los subíndices. Los números de la primer columna se expresan en el sistema decimal. Los restantes se expresan en base 8. Recordar que las operaciones se llevan a cabo en aritmética de base 8.

Dígito	En base 8	Operación	Resultado
4	4		*4*
7	7	*4 x 12 + 7*	*57*
8	10	*57 x 12 + 10*	***736***

Entonces, $478 = (736)_8$

6.3.4 Conversión de la parte fraccionaria

Notar que en algunos casos es imposible expresar una fracción finita en base *b* $(0.u_{-1}\ u_{-2}\ \dots\ u_{-m})_b$ como otra fracción finita en base *p* de manera exacta. Para que una fracción *f* en base *b* tenga una representación exacta en base *p*, *f* deberá ser múltiplo de alguna potencia negativa de *p.* Por ejemplo, la fracción decimal 0,1 tiene una representación binaria infinita $(0.0001100110011\ \dots)_2$ ya que no existen enteros *n* y *e* tal que $0{,}1 = n \cdot 2^{-e}$. Por lo tanto, en algunos casos será necesario establecer un criterio para terminar el algoritmo de conversión. Los criterios más frecuentemente usados son dos: El primero consiste en fijar una cantidad máxima *m* de dígitos para representar a la fracción; el segundo consiste en fijar una cota máxima para la diferencia entre el número original *u* y el número en base *p, v.*

6.3.4.1 Método de multiplicación por *p* usando aritmética en base *b*

Dado un número fraccionario *u*, los dígitos de su representación en base *p* $(v_{-1}\ v_{-2}\ \dots)_p$ se pueden obtener de la siguiente manera:

$v_{-1} = \lfloor u.p \rfloor$
$v_{-2} = \lfloor \{u \times p\} \times p \rfloor$
$v_{-3} = \lfloor \{ \{u \times p\} \times p\} \times p \rfloor$

donde $\{x\}$ denota $x \bmod 1 = x - \lfloor x \rfloor$.

Si se ha fijado un máximo de *m* dígitos para la representación de la fracción, el algoritmo se puede terminar una vez calculado v_{-m}. Luego, si se desea redondear el número para obtener la mejor aproximación se deberá sumar uno a v_{-m} si

{ … { { *u.p* }.*p … p* } es mayor a $\lfloor p/2 \rfloor$.

{ … { { *u.p* }.*p … p* }. p^{-m} es la diferencia entre *u* y *v* (error de representación). Por lo tanto, si se ha fijado una cota ε para la diferencia entre *u* y *v* como criterio de terminación, el algoritmo se puede terminar cuando

$$|\{ \ldots \{ \{ u \$ p \} \$ p \ldots p \} \times p^{-m}| < \varepsilon$$

Ejemplo: Suponer que se quiere convertir la fracción $(0,63)_{10}$ a binario.

Dígito	Operación	Resultado	Parte entera	Parte fraccionaria	Error de representación
v_{-1}	0,63 x 2	1,26	1	0,26	0,26 x 2^{-1} = 0,13
v_{-2}	0,26 x 2	0,52	0	0,52	0,52 x 2^{-2} = 0,13
v_{-3}	0,52 x 2	1,04	1	0,04	0,04 x 2^{-3} = 0,005

Finalmente, $(0,63)_{10} = (0,101)_2$ con un error de representación de $(0,005)_{10}$.

6.3.4.2 Método de división por b usando aritmética en base p

Dado un número *u,* si su representación en base *b* es $(.u_{-1}u_{-2} \ldots u_{-m})_b$, su representación en base *p* se puede hallar evaluando el polinomio $u_{-1}b^{-1} + u_{-2}b^{-2} + \ldots + u_{-m}b^{-m}$ usando aritmética en base *p.* Al igual que en el método 4.1.2, se puede usar la regla de Horner para simplificar la evaluación del polinomio:

$$u_{-1}b^{-1} + u_{-2}b^{-2} + \ldots + u_{-m}b^{-m} = ((\ldots (u_{-m} / b + u_{m-1)} / b + \ldots + u_{-2}) / b + u_{-1}) / b.$$

Se debe tener el cuidado de controlar los errores que ocurran debido al redondeo o truncamiento al dividir por *b;* éstos son en general despreciables, pero no siempre.

Ejemplo: Convertir la fracción $(0,63)_{10}$ a binario.

En este caso se debe operar con aritmética de base 2. Al igual que en el ejemplo del método para enteros de multiplicación por *b* usando aritmética de base *p*, se puede construir una tabla para llevar a cabo la evaluación del polinomio según el método de Horner. Se escriben los dígitos en una columna comenzando con el menos significativo en la primer fila, y terminando con el cero que se encuentra a la izquierda del punto fraccionario en la última. En una columna "resultado" se construirá la representación en base *p*. El resultado de la fila *n* se calcula sumando al dígito de esa fila la división del resultado de la fila *n-1* por *p*. Para la primer fila, suponer que el resultado de la anterior es cero. Finalmente, el resultado de la última fila es el número buscado.

Nota: En la tabla siguiente se han omitido los subíndices. Los números de la primer columna se expresan en el sistema decimal. Los restantes se expresan en base 2.

Dígito	En base 2	Operación	Resultado
3	011	*0 / 1010 + 011*	*011*
6	110	*011 / 1010 + 110*	*110,01000*
0	000	*110,01000/1010*	***0,101***

6.3.5 Casos especiales de cambio de base

Es fácil ver que existe una relación entre los números en base *b* y los números en base $b^{k:}$

$$(\ldots a_3\, a_2\, a_1.\, a_{-1}\, a_{-2}\, \ldots)_b = (\ldots A_3\, A_2\, A_1.\, A_{-1}\, A_{-2}\, \ldots)_b^k ,$$

donde $A_j = (a_{kj+k-1} \ldots a_{kj+1}\, a_{kj})_b$.

El cambio de base en estos casos es directo.

Ejemplo: sistema binario, octal y decimal.

La conversión entre números octales o hexadecimales a binarios es sencilla. Para convertir un número binario a octal, habrá que dividirlo en grupos de 3 bits, con los 3 bits inmediatamente a la izquierda (o derecha) del punto fraccionario (o también llamado punto binario) formando un grupo, los 3 bits inmediatamente a su izquierda formando otro grupo y así sucesivamente. Cada grupo de 3 dígitos binarios se puede convertir directamente a un dígito octal, 0 a 7, de acuerdo a las equivalencias dadas en las primeras líneas de la **Tabla 1.** Puede ser necesario agregar uno o dos ceros para llenar un grupo de 3 bits. La conversión de octal a binario es también fácil. Cada dígito del sistema octal es simplemente reemplazado por su correspondiente representación binaria de 3 dígitos.

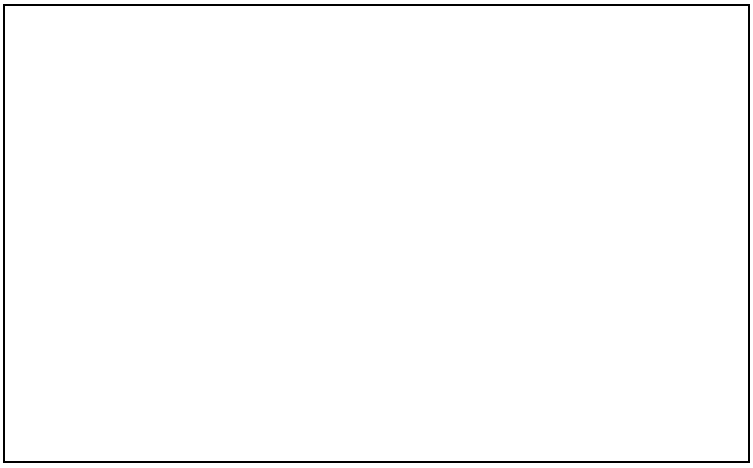
Decimal	Binario	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E

15	1111	17	F
16	10000	20	10

Tabla 1. Representación de los primeros 17 números enteros en los sistemas decimal, binario, octal y hexadecimal.

La conversión hexadecimal a binario es básicamente idéntica a la conversión de octal a binario, con la excepción de que a cada dígito hexadecimal le corresponde un grupo de 4 bits en lugar de uno de 3.

Ejemplos de conversión de octal-binario y hexadecimal-binario



6.4 Representación de la información

Los circuitos electrónicos de una computadora tienen solamente dos configuraciones posibles. Los dígitos “1” y “0” se usan para representar cada una de estas dos configuraciones. Por esta razón, todos los datos dentro de la computadora deben ser representados por secuencias de 1´s y 0´s. La representación de datos mediante 1´s y 0´s se denomina representación binaria.

Definiciones importantes:

Bit: (Binary Digit) Un bit es un dígito binario. Como tal, puede tener 2 valores posibles, 1 y 0. Como los circuitos de una computadora pueden asumir 2 estados, los bits se utilizan para representar el estado de los circuitos. Y siendo uno de estos circuitos la unidad mínima de almacenamiento que posee una computadora, el bit será la mínima unidad de representación.

Byte: En términos generales, un byte es un conjunto de bits. En el presente, se entiende como byte al conjunto de 8 bits.

Palabra: Una palabra es el conjunto de bits que pueden ser accedidos por la CPU en un requerimiento de lectura/escritura.

¿ Qué tipo de datos representarán estas secuencias de 1´s y 0´s ?

1. Números: enteros positivos y negativos, y fracciones.
2. Letras: Todas las letras del alfabeto (mayúsculas y minúsculas), símbolos de puntuación, símbolos matemáticos, etc.
3. Caracteres de control: caracteres para limpiar la pantalla, saltar una línea, etc.
4. Instrucciones de programa.
5. Direcciones de memoria.

La memoria no es más que un conjunto de bits. Si se quisiera mostrar su contenido, se podría mostrar el estado de cada uno de los bits, pero tal representación resultaría larga e ininteligible. En la práctica se usa el código hexadecimal para representar el contenido de la memoria, porque requiere únicamente de dos dígitos para representar cada byte. Es importante no confundir la representación hexadecimal con sistema de numeración en base 16. La representación hexadecimal muestra el contenido de unos bytes de memoria, pero esos bytes pueden representar un número, una cadena de caracteres, etc.

6.4.1 Códigos de representación

Para evitar ambigüedades al intercambiar información, se han definido reglas que sirven para interpretar grupos de bits. A estas reglas se las llama códigos de representación. Algunos ejemplos de éstos códigos son el código Baudot, ASCII y EBCDIC. La definición de estos códigos se puede consultar en el apéndice.

Códigos de caracteres

6.4.1.1 El código Baudot

El código Baudot es un código muy común utilizado en las transmisiones de telegramas y mensajes de telex. Es un código de 5 bits y consecuentemente, parecería de antemano que sólo permite representar 32 caracteres diferentes. No es así. Para salvar esta restricción existen dos caracteres especiales llamados caracteres de cambio. Estos caracteres cambian el alfabeto que se utiliza para interpretar la información recibida. Después de recibir un carácter de cambio, todos los caracteres que siguen corresponden al alfabeto especificado por ese carácter, hasta que se recibe otro carácter de cambio. Pero aún así, la cantidad de caracteres representables es muy limitada, siendo ésta una de las principales

desventajas del código Baudot. Otra falencia es la carencia de lógica en la asignación de los patrones de bits a sus caracteres correspondientes.

6.4.1.2 El código ASCII

El código ASCII (American Standard Code for Information Interchange), fue establecido por el Instituto Nacional Americano de Estándares (ANSI). Es un código de 7 bits que ha sido muy aceptado y es ahora de uso general; y en algunos casos se ha convertido en un código de 8 bits mediante la adición de un dígito de paridad.

Existen varios caracteres del código ASCII, llamados caracteres de control, que tienen significados especiales. Se usan para transmisiones de datos en serie de una máquina a otra, e indican cosas como fin de línea, avance de carro, etc.

6.4.1.3 El código EBCDIC

El código EBCDIC (Extended Binary Coded Decimal Interchange Code) es otro de los códigos más utilizados. Cada carácter se representa con un número de 8 bits. Este código fue inicialmente desarrollado por IBM para usar en sus computadoras, pero luego lo adoptaron otros fabricantes.

6.4.2 Representando números decimales en ASCII y EBCDIC

Los códigos decimal empaquetado y desempaquetado que se describen a continuación fueron muy utilizados en algunas computadoras mainframes, pero en la actualidad han caído en desuso.

6.4.2.1 Código decimal desempaquetado con signo “overpunch”

En este código, usado para representar números decimales, cada dígito decimal ocupa un byte de 8 bits: Los 4 bits más significativos se denominan bits de zona y se usan para indicar el código que se está usando (ASCII o EBCDIC), y los 4 bits menos significativos se usan para codificar al dígito decimal. El signo se representa con los bits de zona del byte menos significativo.

Entonces, un número decimal desempaquetado tendrá la forma:

ZD ZD ZD … SD

donde Z representa a los 4 bits de zona y su valor será:

0011 (3) si se trabaja con el código de caracteres ASCII
1111 (F) si se trabaja con el código de caracteres EBCDIC

D representa a los 4 bits del dígito decimal. Cada dígito decimal se representará con su correspondiente codificación binaria. Por ejemplo, los dígitos 6 y 8 se representarán por la secuencia de bits 0110 y 1000.

Finalmente, S representa a los 4 bits de signo y su valor será:

Al trabajar con el código ASCII
0011 (3) si el número es positivo
1101 (D) si el número es negativo

Al trabajar con el código EBCDIC
1111 (F) ó 1100 (C) si el número es positivo
1101 (D) si el número es negativo

Ejemplo: Representar el número +128 en ASCII y EBCDIC

ASCII: 00110001 00110010 00111000 (31 32 38)
EBCDIC: 11110001 11110010 11111000 (F1 F2 F8)
ó 11110001 11110010 11001000 (F1 F2 C8)

Ejemplo: Representar el número -128 en ASCII y EBCDIC

ASCII: 00110001 00110010 11011000 (31 32 D8)
EBCDIC: 11110001 11110010 11011000 (F1 F2 D8)

6.4.2.2 Código decimal empaquetado

El código decimal empaquetado también se utiliza para representar números enteros. A diferencia del código decimal desempaquetado, se representan 2 dígitos decimales por byte, con la excepción del byte menos significativo que contiene un dígito y el signo.

Si se divide cada byte en 2 conjuntos de 4 bits, cada conjunto representará un dígito decimal. Al igual que en código decimal desempaquetado, cada dígito decimal se representará por su correspondiente codificación binaria. En el caso del byte menos significativo, los 4 bits más significativos representarán un dígito y los otros 4 el signo.

Entonces, un número decimal empaquetado tendrá la forma:

DD DD DD … DS

donde D representa a los 4 bits del dígito decimal y S al signo, que se representa usando las mismas reglas que se aplican para el código decimal desempaquetado:

Al trabajar con el código ASCII

0011 (3) si el número es positivo
1101 (D) si el número es negativo

Al trabajar con el código EBCDIC

1111 (F) ó 1100 (C) si el número es positivo
1101 (D) si el número es negativo

Ejemplo: Representar el número +128 en ASCII y EBCDIC

ASCII: 00010010 10000011 (12 83)
EBCDIC: 00010010 10001100 (12 8C)
ó 00010010 10001111 (12 8F)

Ejemplo: Representar el número -128 en ASCII y EBCDIC

ASCII: 00010010 10001101 (12 8D)
EBCDIC: 00010010 10001101 (12 8D)

6.4.3 Representando números en binario

Un conjunto de bytes de memoria se puede también interpretar como un número binario en notación complemento o sin signo, siendo el programador quien define y sabe cual de estas dos notaciones corresponde en cada caso.

Ejemplos:

1- Representar el número decimal +20 en formato binario ocupando dos bytes.

+20 equivale a: 0000000 00010100 (representación hexadecimal: 00 14)

tanto en notación complemento como sin signo.

2- Representar el número decimal -20 en binario. En este caso únicamente se puede utilizar notación complemento:

-20 equivale a: 11111111 11101100 (representación hexadecimal: FF EC)

La notación complemento es muy conveniente para llevar a cabo las operaciones aritméticas. Por ejemplo, para llevar a cabo 10 - 3, habrá que sumar 10 y (-3) y descartar el acarreo:

Decimal	Complemento a 2
10	00001010
+ (-3)	11111101
+7	1 00000111

se descarta el acarreo

6.4.4 Cadenas de caracteres

En una cadena de caracteres cada carácter ocupa 1 byte y se representa según el código de caracteres que se esté utilizando (ASCII o EBCDIC).

Ejemplos:

1- La cadena de caracteres de 4 bytes 48 4E 4C 41 representa, según la tabla de códigos ASCII, a la cadena HOLA.

Si se utiliza el código EBCDIC, HOLA se representa con la cadena de caracteres C8 D6 D3 C1.

2- Dado un conjunto de 3 bytes que contiene el número decimal +3420 en formato *decimal empaquetado* en un sistema de computación que trabaja con el código ASCII, qué pasa si lo queremos mostrar en una terminal como una cadena de caracteres ?

El número 3420 en formato *decimal empaquetado* se representa con los 3 bytes:

03 42 03.

Si queremos imprimir este campo, consultando la tabla ASCII:

03 representa al símbolo	ETX (End of Text)
42	B
03	ETX

Como ETX es un carácter de control no representable que se utiliza para la comunicación de datos, en la terminal se verá una expresión del tipo: ?B ?, donde ? es un carácter cualquiera.

Si en lugar de usar el formato decimal empaquetado para representar al número +3420 se hubiese usado el formato *decimal desempaquetado,* y asumiendo que el sistema con el que se está trabajando utiliza el código ASCII, entonces sí se verá en la terminal la cadena 3420:

> El número +3420 en formato decimal desempaquetado está representado por los bytes:
>
> 33 34 32 30
>
> que corresponden a los caracteres ASCII 3, 4, 2 y 0.

Una cadena de caracteres debe tener un principio y un fin. El principio (la dirección de memoria en la que empieza la cadena) es siempre conocido, no siendo este el caso del fin. Existen dos alternativas para indicar el final de una cadena de caracteres: se puede especificar la longitud de la cadena o colocar un carácter especial de terminación. Por ejemplo, el lenguaje de programación C utiliza el carácter ´\0' para indicar la terminación de una cadena.

6.5 Tablas de códigos

6.5.1 El código Baudot

1	2	3	4	5	Letras	Números
X	x				A	-
X			x	x	B	?
	x	x	x		C	:
x			x		D	quién es ud ?
x					E	3
x		x	x		F	
	x		x	x	G	
		x		x	H	
	x	x			I	8
x	x		x		J	bell
x	x	x	x		K	(
	x			x	L	)
		x	x	x	M	.
		x	x		N	,
			x	x	O	9
	x	x		x	P	0
x	x	x		x	Q	1
	x		x		R	4
x		x			S	!
				x	T	5
x	x	x			U	7

	x	x	x	x	V	=
x	x			x	W	2
x		x	x	x	X	/
x		x		x	Y	6
x				x	Z	+
					Blanco	
x	x	x	x	x	Cambio a letras	
x	x		x	x	Cambio a números	
		x			Espacio	
			x		Carriage return - Retorno de carro	
	x				Line Feed - Avance de línea	

6.5.2 El código ASCII

Hex	ASCII	Hex	ASCII	Hex	ASCII
00	NUL	2B	+	56	V
01	SOH	2C	‘	57	W
02	STX	2D	-	58	X
03	ETX	2E	.	59	Y
04	EOT	2F	/	5A	Z
05	ENQ	30	0	5B	[
06	ACK	31	1	5C	\
07	BELL	32	2	5D	]
08	BKSP	33	3	5E	↑
09	HT	34	4	5F	←
0A	LF	35	5	60	`
0B	VT	36	6	61	a
0C	FF	37	7	62	b
0D	CR	38	8	63	c
0E	SO	39	9	64	d
0F	SI	3A	:	65	e
10	DEL	3B	;	66	f
11	DC1	3C	<	67	g
12	DC2	3D	=	68	h
13	DC3	3E	>	69	i
14	DC4	3F	?	6A	j
15	NAK	40	@	6B	k
16	SYNC	41	A	6C	l
17	ETB	42	B	6D	m
18	S0	43	C	6E	n
19	S1	44	D	6F	o
1A	S2	45	E	70	p
1B	ESC	46	F	71	q
1C	S4	47	G	72	r
1D	S5	48	H	73	s
1E	S6	49	I	74	t
1F	S7	4A	J	75	u
20	SP	4B	K	76	v
21	!	4C	L	77	w
22	“	4D	M	78	x
23	#	4E	N	79	y
24	$	4F	O	7A	z
25	%	50	P	7B	{
26	&	51	Q	7C	|
27	´	52	R	7D	}
28	(	53	S	7E	~
29	)	54	T	7F	DEL
2A	*	55	U		

NUL	Nulo	HT	Horizontal Tab Tabulación horizontal	NAK	Negative Acknowledgement Reconocimiento negativo
SOH	Start of header Comienzo de encabezado	LF	Line Feed Avanzar línea	SYNC	Synchronous idle Carácter de sincronismo
STX	Start of text Comienzo de texto	VT	Vertical Tab Tabulación vertical	ETB	End of block Fin de bloque

ETX	End of text Fin de texto	FF	Form Feed Avance de página	S0-S7	Separator information
EOT	End of transmission Fin de transmisión	CR	Carriage return Retorno de carro	SP	Space Espacio
ENQ	Enquiry Consulta	SO	Shift out Corrimiento hacia afuera	ESC	Escape
ACK	Positive acknowledgement Reconocimiento	SI	Shft In Corrimiento hacia adentro	DEL	Delete Borrar
BELL	Audible signal Señal sonora	DLE	Data link escape Escape de enlace de datos		
BKSP	Backspace Retroceso	DC1-DC4	Device control Control de dispositivos		

6.5.3 El código EBCDIC

		Dígitos más significativos															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
		(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(A)	(B)	(C)	(D)	(E)	(F)
	0000 (0)	NUL				SP	&	-									0
D	0001 (1)							/		a	j			A	J		1
í	0010 (2)									b	k	s		B	K	S	2
g	0011 (3)									c	l	t		C	L	T	3
	0100 (4)	PF	RES	BYP	PN					d	m	u		D	M	U	4
m	0101 (5)	HT	NL	LF	RS					e	n	v		E	N	V	5
e	0110 (6)	LC	BS	EOB	UC					f	o	w		F	O	W	6
n	0111 (7)	DEL	IL	PRE	EOT					g	p	x		G	P	X	7
o	1000 (8)									h	q	y		H	Q	Y	8
s	1001 (9)									i	r	z		I	R	Z	9
	1010 (A)			SM			!		:								
s	1011 (B)					.	$	,	#								
i	1100 (C)					<		%	@								
g	1101 (D)					(	)	-	´								
	1110 (E)					+	;	>	=								
	1111 (F)					|		?	“								

NUL	Nul	NL	New line Nueva línea	PRE	Prefix Prefijo
PF	Punch off	BS	Backspace Retroceso	SM	Set mode Establecer modo
HT	Horizontal tab Tabulación horizontal	IL	Idle	PN	Punch on
LC	Lower case Minúscula	BYP	Bypass	RS	Reader stop Parar lectora
DEL	Delete Borrar	LF	Line feed Avanzar línea	UC	Upper case Mayúscula
RES	Restore Restaurar	EOB	End of block Fin del bloque	EOT	End of transmission Fin de transmisión
				SP	Space Espacio

7 ANEXO: CODIFICACIÓN DE LAS INSTRUCCIONES

Cada instrucción ensamblador (nemonico + operandos) esta asociada a una instrucción del 8086 que es un código binario. Para poder pasar del nemonico al código binario cada instrucción tiene asociado un formato de codificación también llamado máscara. Por ejemplo:

Instrucción: MOV AX,04h

Máscara: 1011WREG VAL
Código: 10111000 0400

Donde los campos W REG y VAL varían según el tipo de operandos que utilice la instrucción.

NOTACION

Para poder describir los formatos de codificación vamos a utilizar la siguiente notación:

reg indica registro de 8 o 16 bits
reg8 indica registro de 8 bits
reg16 indica registro

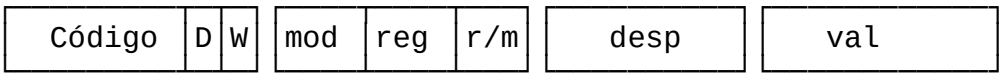
mem indica variable de memoria de 8 o 16 bits
mem8 indica variable de memoria de 8 bits
mem16 indica variable de memoria de 16 bits

val indica valor inmediato de 8 o 16 bits
val8 indica valor inmediato de 8 bits
val16 indica valor inmediato de 16 bits

desp indica desplazamiento de 8 o 16 bits
desp8 indica desplazamiento de 8 bits
desp16 indica desplazamiento de 16 bits

7.1 Formato general de una instrucción

El formato depende de la instrucción y puede contener a lo sumo la siguiente información:



El código de operación aparece en el primer byte. Es el único que existe siempre. Los demás campos pueden aparecer o no dependiendo del tipo de instrucción.

Los operandos de la instrucción se reflejan en los campos mod reg y r/m.

Un operando se especifica mediante mod y R/M y puede ser un registro o una dirección de memoria.

El otro operando se especifica mediante reg y debe ser un registro.

El campo desp es el componente desplazamiento de una dirección de memoria. Puede ser uno o dos bytes. Si ocupa dos bytes, el byte menos significativo se almacena primero.

El campo val es un valor inmediato. Puede ocupar uno o dos bytes. Como en el caso anterior, si ocupa dos bytes, el byte menos significativo se almacena primero.

Los campos que aparecen en la codificación de una instrucción ensamblador son:

W Bit que indica la longitud de los operandos.
Aparece dentro del byte de código.

W=0 byte
W=1 palabra (2 bytes)

D Bit que indica el destino.
Aparece dentro del byte de código.

D=0 El operando destino se especifica mediante los campos mod y r/m.
D=1 El operando destino se especifica mediante el campo reg.

reg dos o tres bits que indican el tipo de registro que se va a utilizar como operando, en el caso de instrucciones de dos operandos puede ser destino o fuente según el valor del bit D.

mod dos bits que indican el tipo de desplazamiento del operando.

r/m tres bits que indican el tipo de direccionamiento del operando.

Es importante recalcar que un formato de instrucción NO siempre debe contener TODOS los campos, esto varia según el numero de operandos y el modo de direccionamiento de estos.

Las tablas que se muestran en la siguiente página corresponde a los valores que pueden tomar los campos Reg. Mod y R/M.

7.2 Tablas de codificación

TABLA DEL CAMPO REG

reg	reg8	reg16
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

REG	REGISTRO SEGMENTO
00	ES
01	CS
10	SS
11	DS

TABLA DEL CAMPO MOD

MOD	DESPLAZAMIENTO
00	No tiene (0)
01	8 bits en el byte siguiente
10	16 bits en los dos bytes siguientes
11	indica que R/M es un registro y se debe buscar en la tabla de REG

TABLA DEL CAMPO R/M

R/M	REG. BASE	REG. INDICE
000	BX	SI
001	BX	DI
010	BP	SI
011	BP	DI
100	Ninguno	SI
101	Ninguno	DI
110	BP	Ninguno(*)
111	BX	Ninguno

(*) CASO ESPECIAL - Si mod=00, entonces la instrucción contiene el desplazamiento en dos bytes adicionales.

7.3 Ejemplos de codificación de instrucciones

Ejemplo 1: MOV AX,0256H

formato: 1011wreg | val

w= 1 Se transfiere un palabra
reg= 000 por el registro AX
val= 5602 Primero se almacena el byte menos significativo
código: 10111000 01010110 00000010

Ejemplo 2: MOV DS,AX

formato: 10001100 | mod 0reg r/m

mod= 11 indica que R/M es un registro
reg= 11 registro segmento DS
r/m= 000 Registro AX en tabla REG
código: 10001100 | 11011000

Ejemplo 3: SUB AX,BX

formato: 001010DW | mod reg r/m

D= 0 Cuando aparezcan dos registros de operandos el destino siempre en los campos mod y r/m.
W= 1 Los operandos son de 16 bits.
mod= 11 indica que R/M es un registro
r/m= 000 Por el registro AX
reg= 011 Por el registro BX
código: 00101001 11011000

Ejemplo 4: MOV BX,[BX]+0FFFH

formato: 100010DW | mod reg r/m | desp

D= 1 El destino es un registro
W= 1 Transferencia de 16 bits
reg= 011 Destino en el registro BX
mod= 10
r/m= 111 Se direcciona el origen con el registro BX
desp= FF00H
código: 10001011 | 11011111 | 11111111 00001111

Ejemplo 5: MOV [BP]+0FH,AX CASO ESPECIAL

formato: 100010DW | mod reg r/m | desp

D= 0 El destino en los campos mod y r/m
W= 1 Transferencia de 16 bits
reg= 000 origen en AX de la tabla reg
mod= 01 Desplazamiento de un byte
R/M= 110 Se direcciona el operando origen con BP
desp= 0F00 Desplazamiento de 16 bits se encuentra en dos bytes adicionales.

Código: 10001001 | 01000110 | 00001111 00000000

8 BIBLIOGRAFÍA

- Assembly Language for x86 Processors, 5th edition
  Kip Irvine, Florida International University
  ISBN: 0-13-602212-X
  Prentice-Hall (Pearson Education)

- The Art of Assembly Language Programming
  by Randall Hyde
  ISBN: 1886411972
  Free ebook

- IA-32 Intel® Architecture Software Developer's Manual
  Volume 1 - Basic Architecture
  INTEL Corp.