Apunte Assembler Intel 8086

Log de Cambios:

Versión	Fecha	Comentarios
1.0	Abril 2011	Versión Inicial
1.1	Octubre 2011	Se agrega int 21h, serv 0Ah para lectura de cadenas. Ejemplos en comparaciones y copias de cadenas. Uso de la pila. Referencia al uso de OFFSET en lugar de LEA. Rutinas Internas. Agregado de log de cambios.
2.0	Mayo 2012	Cambios orientados a programacion con Netwide Assembler
3.0	Abril 2019	Cambios de x86 a 32 bits

Introducción

Estructura de un programa

Un programa estará dividido en 3 secciones según el siguiente esquema

```
section     .data
section     .bss
section     .text
```

.data define el inicio del bloque destinado a colocar las variables con contenido inicial a utilizar dentro del programa.

.bss define el inicio del bloque destinado a colocar las variables sin contenido inicial a utilizar dentro del programa.

.text define el inicio del bloque destinado a colocar las instrucciones del programa. En otras palabras, define el inicio del código

Un ejemplo completo de la estructura de un programa se muestra a continuación:

```
global      _main
extern      _printf

section     .data
mensaje     db    'Hola Mundo!',0   ;campo con el string a imprimir.
                                    ;Debe finalizar con 0 binario
section     .bss

section     .text
_main:
                        ;acá va el código del programa
      push  mensaje     ;Parametro 1:direccion del mensaje a imprimir
      call  _printf
      add   esp,4       ;Coloco puntero a la pila donde estaba

      ret
```

Definición de variables

Una variable es un espacio que se reserva en la memoria para guardar datos que serán usados en el programa.
Para reservar espacio es necesario saber la cantidad de bytes que vamos a necesitar para el dato a guardar.
Es posible en algunos casos que necesitemos inicializar ese espacio con un contenido en algún formato ya definido (numérico, carácter, array, etc)

Para definir variables inicializadas se usan las siguientes pseudo instrucciones

DB	Define Byte (1 byte)
DW	Define Word (1 palabra; 2 bytes)
DD	Define Double (doble palabra; 4 bytes)
DQ	Define Quad (cuadruple palabra; 8 bytes)
DT	Define Ten (10 bytes)

Para indicar su contenido se coloca luego de la directiva de dato el valor de distinta forma según el tipo. Por ejemplo:

miVariable db 'A'	Define 1 byte y le asigna el carácter ASCII A

Para definir variables sin contenido se usan las siguientes directivas:

resb	Reserve Byte (1 byte)
resw	Reserve Word (1 palabra; 2 bytes)
resd	Reserve Double (doble palabra; 4 bytes)
resq	Reserve Quad (cuadruple palabra; 8 bytes)
rest	Reserve Ten (10 bytes)

Por ejemplo:

miVariable resb 1	Reserva 1 byte sin contenido inicial

Variables Numéricas

Al definir un campo numérico el mismo será almacenado en formato binario de punto fijo con signo.

La expresión del contenido inicial se puede indicar en base 10 y además en base 2, base 8 o base 16 agregando las letras *b*, *o* y *h* respectivamente. También puede indicarse el signo.

Veamos a continuación algunos ejemplos:

decimal db 11	1 byte con valor numérico 11 expresado en base 10
decimal2 dw 12	2 bytes con valor numérico 12 expresado en base 10
decimal3 dd 12345	4 bytes con valor numérico 12345 expresado en base 10
hexa1 db 0Bh	1 byte con valor numérico 11 expresado en base 16 (*)
hexa2 dw 0Ch	2 bytes con valor numérico 12 expresado en base 16 (*)
hexa3 dd 10h	4 bytes con valor numérico 16 expresado en base 16 (*)

octal1	db	13o	1 byte con valor numérico 11 expresado en base 8
octal2	dw	71o	2 bytes con valor numérico 57 expresado en base 8
octal3	dd	2322o	4 bytes con valor numérico 1234 expresado en base 8
binario	db	1011b	1 byte con valor numérico 11 expresado en base 2
binario	dw	1011b	2 bytes con valor numérico 11 expresado en base 2
binario	dd	1011b	4 bytes con valor numérico 11 expresado en base 2
nega	db	-1	1 byte con valor numérico -1 expresado en base 10

(*) Para definir valores numéricos en base 16 deben comenzar con un digito numérico.

Variables caracteres

Al definir un campo para guardar letras, caracteres numéricos u otro tipo de símbolo los mismos serán almacenados en formato ASCII.

La expresión del contenido inicial deberá encerrarse entre comillas simples o dobles

Veamos a continuación algunos ejemplos:

letra	db	“A”	1 byte con valor ascii letra A
letra2	db	‘a’	1 byte con valor ascii letra a
numero	db	“1”	1 byte con valor ascii número 1
cadena	db	“Hola mundo”	10 bytes con valores ascii

Arrays

También podemos definir estructuras de array.
Para ello se utiliza la palabra reservada *times* para indicar la cantidad de repeticiones.

Veamos a continuación algunos ejemplos:

vecVacio_b times 10 resb 1	Vector de 10 bytes sin contenido inicial
vecVacio_w times 10 resw 1	Vector de 20 bytes sin contenido inicial (10 posiciones de 2 bytes c/u)
vecVacio_d times 5 resd 2	Vector de 20 bytes sin contenido inicial (5 posiciones de 4 bytes c/u)
vecLleno times 10 db ‘A’	Vector de 10 bytes con contenido inicial ‘A’

Detalle de la Arquitectura de Programación/ISA (Instruction Set Architecture)

Registros

Generales

Arquitecturas			Descripcion
16 bits	**32 bits**	**64 bits**	
AX (16 bits) AH (8 bits parte alta) AL (8 bits parte baja)	**EAX** (32 bits) AX (16 bits) AH (8 bits parte alta) AL (8 bits parte baja)	**RAX** (64 bits) EAX (32 bits) AX (16 bits) AL (8 bits parte baja)	**Acumulator**. Se usa como operando de instrucciones aritméticas y lógicas
BX (16 bits) BH (8 bits parte alta) BL (8 bits parte baja)	**EBX** (32 bits) BX (16 bits) BH (8 bits parte alta) BL (8 bits parte baja)	**RBX** (64 bits) EBX (32 bits) BX (16 bits) BL (8 bits parte baja)	**Base**. Se usa como registro base en operaciones que direccionan indirectamente al operando.
CX (16 bits) CH (8 bits parte alta) CL (8 bits parte baja)	**ECX** (32 bits) CX (16 bits) CH (8 bits parte alta) CL (8 bits parte baja)	**RCX** (64 bits) ECX (32 bits) CX (16 bits) CL (8 bits parte baja)	**Counter**. Se usa en instrucciones que requieren el uso de contadores, por ejemplo en loops o instrucciones que manejan strings
DX (16 bits) DH (8 bits parte alta) DL (8 bits parte baja)	**EDX** (32 bits) DX (16 bits) DH (8 bits parte alta) DL (8 bits parte baja)	**RDX** (64 bits) EDX (32 bits) DX (16 bits) DL (8 bits parte baja)	**Data**. Se usa en operaciones que requieren duplas de registros

Los registros generales tienen la particularidad de que pueden ser accedidos a cada una de sus mitades como registros independientes, es decir cada uno de ellos se divide en una parte llamada Alta (High) y otra llamada Baja (Low), según se especifica en el siguiente esquema:

Indice

Arquitecturas			Descripcion
16 bits	**32 bits**	**64 bits**	
SI (16 bits)	**ESI** (32 bits) SI (16 bits)	**RSI** (64 bits) ESI (32 bits) SI (16 bits) SIL (8 bits)	**Source**. Se usa para operaciones de manejo de cadenas para apuntar al operando “origen”
DI (16 bits)	**EDI** (32 bits) DI (16 bits)	**RDI** (64 bits) EDI (32 bits) DI (16 bits) DIL (8 bits)	**Destination**. Se usa para operaciones de manejo de cadenas para apuntar al operando “destino”

Pila

Arquitecturas			Descripcion
16 bits	**32 bits**	**64 bits**	
BP (16 bits)	**EBP** (32 bits) BP (16 bits)	**RBP** (64 bits) EBP (32 bits) BP (16 bits) BPL (8 bits)	**Base**. Apunta a la base de la pila
SP (16 bits)	**ESP** (32 bits) SP (16 bits)	**RSP** (64 bits) ESP (32 bits) SP (16 bits) SPL (8 bits)	**Stack**. Apunta al tope de la pila

Segmento

Arquitecturas			Descripcion
16 bits	**32 bits**	**64 bits**	
DS	DS	DS	Data. Apunta al inicio del segmento de datos
CS	CS	CS	Code. Apunta al inicio del segmento de código
SS	SS	SS	Stack. Apunta al inicio del segmento de pila
ES	ES	ES	Extra. Apunta al inicio de algún segmento cuando es requerido.
	FS	FS	Idem ES
	GS	GS	Idem ES

Control

<table>
<tr><th colspan="4">Arquitecturas</th></tr>
<tr><th>16 bits</th><th>32 bits</th><th>64 bits</th><th>Descripcion</th></tr>
<tr><td>IP</td><td>EIP</td><td>RIP</td><td>Instruction Pointer. Contiene la dirección de la instrucción corriente en ejecución.</td></tr>
<tr><td>FLAGS</td><td>EFLAGS</td><td>RFLAGS</td><td>Flags. Se usa para almacenar el estado general de la CPU. Cada bit tiene un significado en particular y algunos específicamente no se usan:
<table>
<tr><th colspan="3">FLAGS</th></tr>
<tr><td>0</td><td>C</td><td>Carry flag</td></tr>
<tr><td>1</td><td>-</td><td>Reserved</td></tr>
<tr><td>2</td><td>P</td><td>Parity flag</td></tr>
<tr><td>3</td><td>-</td><td>Reserved</td></tr>
<tr><td>4</td><td>A</td><td>Adjust flag</td></tr>
<tr><td>5</td><td>-</td><td>Reserved</td></tr>
<tr><td>6</td><td>Z</td><td>Zero flag</td></tr>
<tr><td>7</td><td>S</td><td>Sign flag</td></tr>
<tr><td>8</td><td>T</td><td>Trap flag (single step)</td></tr>
<tr><td>9</td><td>I</td><td>Interrupt enable flag</td></tr>
<tr><td>10</td><td>D</td><td>Direction flag</td></tr>
<tr><td>11</td><td>O</td><td>Overflow flag</td></tr>
<tr><td>12-13</td><td>IOPL</td><td>I/O privilege level (286+ only), always 1 on 8086 and 186</td></tr>
<tr><td>14</td><td>NT</td><td>Nested task flag (286+ only), always 1 on 8086 and 186</td></tr>
<tr><td>15</td><td>0</td><td>Reserved, always 1 on 8086 and 186, always 0 on later models</td></tr>
<tr><th colspan="3">EFLAGS</th></tr>
<tr><td>16</td><td>RF</td><td>Resume flag (386+ only)</td></tr>
<tr><td>17</td><td>VM</td><td>Virtual 8086 mode flag (386+ only)</td></tr>
<tr><td>18</td><td>AC</td><td>Alignment check (486SX+ only)</td></tr>
<tr><td>19</td><td>VIF</td><td>Virtual interrupt flag (Pentium+)</td></tr>
<tr><td>20</td><td>VIP</td><td>Virtual interrupt pending (Pentium+)</td></tr>
<tr><td>21</td><td>ID</td><td>Able to use CPUID instruction (Pentium+)</td></tr>
<tr><td>22</td><td>0</td><td>Reserved</td></tr>
<tr><td>23</td><td>0</td><td>Reserved</td></tr>
<tr><td>24</td><td>0</td><td>Reserved</td></tr>
<tr><td>25</td><td>0</td><td>Reserved</td></tr>
<tr><td>26</td><td>0</td><td>Reserved</td></tr>
<tr><td>27</td><td>0</td><td>Reserved</td></tr>
<tr><td>28</td><td>0</td><td>Reserved</td></tr>
<tr><td>29</td><td>0</td><td>Reserved</td></tr>
<tr><td>30</td><td>0</td><td>Reserved</td></tr>
</table>
</td></tr>
</table>

			31	0	Reserved
			RFLAGS		
			32-63	0	Reserved

Tipos de Direccionamiento

Implcícito

El dato está implícito en el código de operación.
Ej: CBW

Registro

El dato está en un registro.
Ej: MOV EAX,EBX

Inmediato

El dato está dentro de la instrucción
Ej: MOV EAX,5

Directo

El dato está en memoria apuntado por un campo que contiene la dirección
Ej: MOV EAX,[VARIABLE]
MOV EAX,[VARIABLE+2]

Registro Indirecto:

El dato está en memoria apuntado por un registro base o índice.
Ej: MOV EAX,[EBX]
MOV EAX,[ESI]

Registro Relativo

El dato está en memoria apuntado por un registro base o índice más un desplazamiento.
Ej: MOV EAX,[EBX+4]
MOV EAX, [VECTOR+EBX]
MOV [EDI+3],EAX

Base + Indice

El dato está en memoria apuntado por un registro base más un registro índice.
Ej: MOV [EBX+EDI],CL

Base Relativo + índice

El dato está en memoria apuntado por un registro base más un registro índice más un desplazamiento.
Ej: MOV EAX,[EBX+EDI+4]
MOV EAX,[VECTOR+EBX+EDI]

Tipos de Dato

Numéricos enteros: Binario de punto fijo con Signo de 1 y 2 bytes

Numérico decimal: Binario de punto Flotante IEEE de 1 y 2 bytes

Caracteres: Ascii

Memoria

Celda de Memoria:
Una celda de memoria ocupa 1 Byte (8 bits)

Palabra:
Una palabra está formada por 2 bytes (16 bits)

Endiannes

Es el método aplicado para almacenar datos mayores a un byte en una computadora respecto a la dirección que se le asigna a cada uno de ellos en la memoria.
Existen 2 métodos:
Big-Endian: determina que el orden en la memoria coindice con el orden lógico del dato. Podríamos traducirlo a “el dato final en la mayor dirección”
Little-Endian: es a la inversa, el dato inicial para la lógica se coloca en la mayor dirección y el dato final en la menor. Podríamos traducirlo a “el dato final en la menor dirección”.

La familia de procesadores de Intel 80x86 usa el método **Little-Endian**

Veamos mediante algunos ejemplos como es el manejo de memoria mediante el método de Little-Endian:

- **Caso 1**: Definición de un área de memoria con contenido inicial definido en formato carácter

```
msg   db    'HOLA'
```

Aquí la posición de memoria que toma cada carácter recibido es la que por intuición uno asume, o sea, la letra 'H' en la dirección menor:

Carácter	H	O	L	A
Código Ascii (hexa)	48	4F	4C	41
Desplazamiento (en bytes) dentro del segmento de datos	0	1	2	3

- **Caso 2**: Definición de un área de memoria con contenido inicial definido en formato numérico

```
numdw      dw    012Fh
numdd      dd    01234567h
```

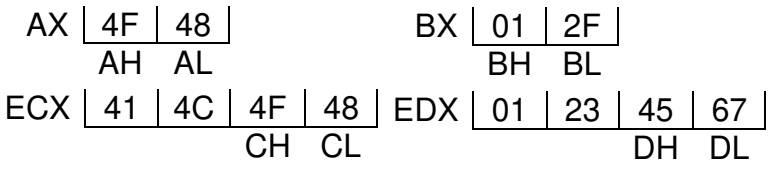
Aquí es donde se observa la ubicación de los bytes con el método Little-Endian, el byte menos significativo se ubica en la dirección de memoria menor que el byte más significativo

numdw	2F	01		
numdd	67	45	23	01
Desplazamiento (en bytes) dentro del segmento de datos	0	1	2	3

- **Caso 3**: Se ejecuta una copia de 2 y 4 bytes de memoria a registro

```
mov   AX,[msg]
mov   BX,[numdw]
mov   ECX,[msg]
mov   EDX,[numdd]
```

El contenido final de los registros AX, BX, ECX y EDX son:



La parte alta del registro contiene el byte de orden superior de memoria, y la parte baja del registro contiene el byte de orden inferior.

Instrucciones

Transferencia y Comparacion

MOV

Copia bytes de un operando a otro.
Las combinaciones posibles de operando y ejemplos se muestran a continuación:

Combinaciones	Ejemplos

MOV <reg>	MOV AH,BL
	MOV AX,BX
	MOV CX,DS

MOV <reg>,<memoria> (*1)	MOV CH,[AA55H] / MOV CH,[VARIABLE]
MOV <reg>,<memoria> (*1)	MOV ECX,[VARIABLE]
MOV <reg>,<inmediato> (*2)	MOV CX,2450h
MOV <reg>,<inmediato> (*2)	MOV EAX,0h
MOV <reg de segmento>,<reg 16 bits>	MOV SS,AX
MOV <reg de segmento>,<memoria> (*3)	MOV SS,[4584H] / MOV SS,[VARIABLE]
MOV <memoria>,<reg> (*1)	MOV [DI],AX / MOV WORD[VARIABLE],AX MOV DWORD[VARIABLE],EAX
MOV <memoria>,<inmediato> (*4)	MOV BYTE [DI],234h / MOV BYTE [VARIABLE],2Ah MOV DWORD[VARIABLE],1234
MOV <memoria>,<reg de segmento>(*1) (*3)	MOV [DI],DS / MOV [VARIABLE],DS

(*1) Se copiaran tantos bytes segun la longitud del registro
(*2) El campo inmediato conviene que esté definido con longitud menor o igual a la del operando 1. Si es mayor se perderan bytes en la copia. Si es menor se alinea el contenido a la derecha del op 1.
(*3) El campo en memoria conviene que esté definido de 2 bytes (dw). De ser menor compiará 2 bytes a partir de la direcicon apuntada por el operando 2
(*4) Debe indicarse la longitud del operando en memoria para que sea tomada en cuenta para la cantidad de bytes a copiar. En el ej se especifica con la palabra BYTE, podria ser WORD en caso de 2 bytes y DWORD en caso de 4 bytes

XCHG

Intercambia el contenido de 2 operandos
Las combinaciones posibles de operando y ejemplos se muestran a continuación:

Combinaciones	Ejemplos
XCHG <reg>	XCHG AH,BL XCHG AX,BX XCHG EAX,ECX
XCHG <reg>,<memoria> (*1)	XCHG CH,[AA55H] / XCHG CH,[VARIABLE]
XCHG <memoria>,<reg> (*1)	XCHG [DI],AX / XCHG [VARIABLE],AX

(*1) Se intercambiaran tantos bytes según la longitud del registro. Conviene que el campo en memoria esté definido de la misma longitud para evitar sobreescribir el dato a continuación en caso de ser de menor longitud.

LEA

Carga en el operando 1 la dirección efectiva (desplazamiento dentro del segmento de datos) del operando 2 (un campo en memoria). Ej.

```
LEA  BX,[VARIABLE]
```

Notar que la variable se encuentra encerrada con corchetes.

Otra manera de hacerlo es usando la instrucción MOV

```
MOV  BX,VARIABLE
```

CMP

Compara 2 operandos.
Las combinaciones posibles de operando y ejemplos se muestran a continuación:

Combinaciones	Ejemplos

CMP <reg>	CMP AH,BL
	CMP AX,BX
	CMP EAX,EBX

CMP <reg>,<memoria> (*1)	CMP CH,[AA55H] / CMP CH,[VARIABLE] CMP EAX,[VARIABLE]
CMP <reg>,<inmediato> (*2)	CMP CX,2450h / CMP CX,2Ah CMP EAX,1000
CMP <memoria>,<reg> (*1)	CMP [DI],AX / CMP [VARIABLE],AX CMP [VARIABLE],EAX

CMP <memoria>,<inmediato> (*2)	CMP [DI],245h / CMP BYTE [VARIABLE],2Ah
	CMP BYTE[VARIABLE],'A'
	CMP WORD[VARIABLE],2ah
	CMP DWORD[VARIABLE],2ah

(*1) Se compararán tantos bytes segun la longitud del registro. Es aconsejable que el campo en memoria tenga la misma longitud que el registro.
(*2) Debe indicarse la longitud del operando en memoria para que sea tomada en cuenta para la cantidad de bytes a comparar. En el ej se especifica con la palabra BYTE, podria ser WORD en caso de 2 bytes y DWORD en caso de 4 bytes

Saltos

JMP

Provoca un salto incondicional al punto del programa indicado en el operando (un operando inmediato). Ej:

```
     JMP  otraParte
     .......

otraParte:
```

Jxx

Las instrucciones del tipo Jxx provocan saltos condicionales hasta +127 o -128 bytes y podemos separarlas en 3 secciones:

1) Saltos generales:

JE	Salta si es igual
JNE	Salta si es distinto
JZ	Salta si es cero
JNZ	Salta si no es cero
JCXZ	Salta si el contenido de reg CX es cero
JC	Salta si carry flag distinto de cero

2) Saltos con signo

JG	Salta si es mayor
JGE	Salta si es mayor o igual
JL	Salta si es menor
JLE	Salta si es menor o igual
JNG	Salta si no es mayor
JNGE	Salta si no es mayor o igual
JNL	Salta si no es menor
JNLE	Salta si no es menor o igual

3) Saltos sin signo

JA	Salta si es mayor
JAE	Salta si es mayor o igual
JB	Salta si es menor
JBE	Salta si es menor o igual
JNA	Salta si no es mayor
JNAE	Salta si no es mayor o igual
JNB	Salta si no es menor
JNBE	Salta si no es menor o igual

LOOP

Resta uno al contenido del registro ECX y si el resultado es distinto de 0 salta al punto del programa indicado en el operando. Ej:

```
            MOV  ECX,100
ROTULO:
      . . . . . .
            LOOP ROTULO
```

Otra manera de realizar un ciclo sin usar la instrucción LOOP es combinando las instrucciones DEC y JUMP

```
            MOV  ECX,100
ROTULO:
      . . . . . .
            DEC ECX
            JNZ ROTULO
```

CALL – RET (rutinas internas)

Para invocar una rutina interna usamos la instrucción CALL, que internamente guarda en el stack la dirección de la siguiente instrucción y ejecuta un salto al rotulo indicado en el operando.
Una vez finalizada la rutina interna, debe retornarse a la siguiente instrucción del programa llamador usando la instrucción RET, que internamente retorna a la dirección almacenada en el stack por la instrucción CALL..
Ej:

```
CALL RUT_INT
...
...

RUT_INT:
  ...
        RET
```

Aritméticas

ADD

Realiza la suma del binario de punto fijo con signo del operando uno con el del operando 2 y deja el resultado en el operando 1.

Las combinaciones posibles de operando y ejemplos se muestran a continuación:

Combinaciones	Ejemplos

ADD <reg>,<reg>	ADD AH,BL
	ADD AX,BX
	ADD ECX,EAX

ADD <reg>,<memoria> (*1)	ADD CH,[AA55H] / ADD CH,[VARIABLE] ADD ECX,[VARIABLE]
ADD <reg>,<inmediato> (*2)	ADD CX,2450h ADD ECX,1234
ADD <memoria>,<reg> (*1)	ADD [DI],AX / ADD WORD[VARIABLE],AX ADD DWORD[VARIABLE],ECX
ADD <memoria>,<inmediato> (*2)(*3)	ADD [DI],245h / ADD BYTE[VARIABLE],2Ah ADD DWORD[VARIABLE],1234

(*1) Es aconsejable que el campo en memoria tenga la misma longitud que el registro.

(*2) El valor numérico del campo inmediato debe poder representarse en una cantidad de bytes menor o igual a la del operando 1.
(*2) Debe indicarse la longitud del operando en memoria para. En el ej se especifica con la palabra BYTE, podria ser WORD en caso de 2 bytes y DWORD en caso de 4 bytes

SUB

Realiza la resta del binario de punto fijo con signo del operando uno con el del operando 2 y deja el resultado en el operando 1.

Las combinaciones posibles de operandos y ejemplos son iguales a las de la operación ADD.

INC

Incrementa en 1 al binario de punto fijo con signo del operando.

Combinaciones	Ejemplos
INC <reg>	INC AH INC AX INC EAX
INC <memoria> (*1)	INC BYTE[VARIABLE] INC WORD[VARIABLE] INC DWORD[VARIABLE]

(*1) Debe indicarse la longitud del operando en memoria para. En el ej se especifica con la palabra BYTE, podria ser WORD en caso de 2 bytes y DWORD en caso de 4 bytes

DEC

Decrementa en 1 al binario de punto fijo con signo del operando.
Las combinaciones posibles de operandos y ejemplos son iguales a las de la operación INC.

MUL – IMUL

Ambas instrucciones multiplican el contenido de un registro (multiplicando) con el contenido del operando (multiplicador)

Si el operando es de 8 bits se usa el registro AL como multiplicando, dejando el resultado en el registro AX.

Si el operando es de 16 bits se usa el registro AX como multiplicando, dejando el resultado en la dupla de registros DX:AX.
Si el operando es de 32 bits se usa el registro EAX como multiplicando, dejando el resultado en la dupla de registros EDX:EAX.

En el caso de la MUL, realiza una multiplicación sin signo, o sea, multiplicador y multiplicando se interpretan como binarios de punto fino SIN signo.

En el caso de la IMUL, realiza una multiplicación con signo, o sea, multiplicador y multiplicando se interpretan como binarios de punto fino CON signo. Multiplica el contenido del op1 por el op2 y almacena el resultado en el op1.
Si no hay overflow CF/OF = 0 sino CF/OF = 1

Combinaciones	Ejemplos
MUL <reg 8 bits>	MUL BH ;multiplica (AH)*(BH) y deja result en AX
MUL <memoria 8 bits>	MUL byte[var8] ;multiplica (AH)*(var8) y deja result en AX
MUL <reg 16 bits>	MUL BX ;multiplica (AX)*(BX) y deja result en DX:AX
MUL <memoria 16 bits>	MUL word[var16] ;multiplica (AX)*(var16) y deja result en DX:AX
MUL <reg, 32 bits>	MUL EBX ;multiplica (EAX)*(EBX) y deja result en EDX:EAX
MUL <memoria 32 bits>	MUL dword[var32] ;multiplica (EAX)*(var32) y deja result en EDX:EAX
IMUL <reg, inm>	IMUL ECX,4 ; multiplica (ECX) * 4 y deja el resultado en ECX
IMUL <reg, reg>	IMUL EAX,EBX ; multiplica (EAX) * (EBX) y deja el resultado en EAX
IMUL <reg>, <long><mem>	IMUL EAX,dword[var32] ; multiplica (EAX) * (var32) y deja el resultado en EAX

DIV – IDIV

Ambas instrucciones dividen el contenido de un registro (dividendo) por el contenido del operando (divisor)

Si el operando es de 8 bits se usa el registro AX (16 bits) como dividendo, dejando el cociente en el registro AL (8 bits) y el resto en el AH (8 bits).

Si el operando es de 16 bits se usa la dupla de registros DX:AX (32 bits) como dividendo, dejando el cociente en el registro AX (16 bits) y el resto en el DX (16 bits).

En el caso de la DIV, realiza una división sin signo, o sea, dividendo y divisor se interpretan como binarios de punto fino SIN signo.

En el caso de la IDIV, realiza una división con signo, o sea, dividendo y divisor se interpretan como binarios de punto fino CON signo.

Combinaciones	Ejemplos
DIV <reg 8 bits>	DIV BH ;divide (AX)/(BH) y deja coc en AL y resto en AH
DIV <memoria 8 bits>	DIV byte[var8] ;divide (AX)/(var8) y deja coc en AL y resto en AH
DIV <reg 16 bits>	DIV BX ;divide (DX:AX)/(BX) y deja coc en AX y resto en DX
DIV <memoria 16 bits>	DIV word[var16] ;divide (DX:AX)/(var16) y deja coc en AX y resto en DX
DIV <reg, 32bits>	DIV ECX ;divide (EDX:EAX)/(ECX) y deja coc en EAX y resto en EDX
DIV <memoria 32 bits>	DIV dword[var32] ;divide (EDX:EAX)/(var32) y deja coc en EAX y resto en EDX
Con IDIV los ej son idénticos	

Logicas

AND - Logical AND

AND op1,op2

Ejecuta la operación lógica AND entre *op1* y *op2*, dejando el resultado en *op1*.

Combinaciones	Ejemplos en NASM
AND <reg>,<reg>	AND AH,BL AND AX,BX AND ECX,EAX
AND <reg>,<long><mem>	AND AL,byte[VAR_8B] (*1) AND ECX,dword[VAR_32B]
AND <reg>,<inm>	AND CH,00h AND CX,250h AND CX,3456
AND <long><mem>,<inm>	AND word[VAR_16B],1010b AND dword[VAR_32B],FFCC00AAh
AND <long><mem>,<reg>	AND byte[VAR_8B],BL(*2) AND dword[VAR_32B],EAX

(*1) Se tomarán tantos bytes según la longitud del registro. Es aconsejable que el campo en memoria tenga la misma longitud que el registro.
(*2) Debe indicarse la longitud del operando en memoria para que sea tomada en cuenta para la cantidad de bytes a utilizar. En el ej se especifica con la palabra BYTE, podria ser WORD en caso de 2 bytes y DWORD em caso de 4 bytes.

OR - Logical Inclusive OR

OR op1,op2

Ejecuta la operación lógica OR entre *op1* y *op2*, dejando el resultado en *op1*.
Las combinaciones son idénticas que para la instrucción AND

XOR - Logical Exclusive OR

XOR op1,op2

Ejecuta la operación lógica Exclusive OR entre *op1* y *op2*, dejando el resultado en *op1*.
Las combinaciones son idénticas que para la instrucción AND

NOT - One's Complement Negation

NOT op

Ejecuta la operación lógica NOT en *op* (cambia los bits en 1 por 0 y viceversa)

Combinaciones	Ejemplos en NASM
NOT <reg>	NOT AH NOT AX NOT ECX
NOT <mem>	NOT byte[VAR_8B] (*2) NOT word[VAR_16B] NOT dword[VAR_32B]

(*2) Debe indicarse la longitud del operando en memoria para que sea tomada en cuenta para la cantidad de bytes a utilizar. En el ej se especifica con la palabra BYTE, podria ser WORD en caso de 2 bytes y DWORD en caso de 4 bytes

Corrimiento

Las instrucciones de corrimiento mueven a derecha o izquierda los bits contenidos del operando 1. La cantidad a correr deberá estar contenida en el operando 2 (registro CL o un operando inmediato de 8 bits).

SHL

Hace corrimiento a izquierda

Combinaciones	Ejemplos
SHL <reg>,<inm>	SHL AH,5
SHL <reg>,CL	SHL AX,CL
SHL <long><mem>,<inm>	SHL dword[VAR_32B],5
SHL <long><mem>,CL	SHL dword[VAR_32B],CL

SHR

Hace corrimiento a izquierda
Las combinaciones son igual a las de SHL

Manejo de la Pila (stack)

PUSH – POP

La pila es usada para almacenar transitoriamente datos o direcciones de retornos de subrutinas. El nombre de PILA no es otra cosa sino por su forma de uso, es decir, el ultimo que entra es el primero que sale (LIFO – Last In First Out)

La instrucción PUSH sirve para poner el dato de un registro en la pila, en cambio POP se usa para recuperar el dato de la pila y lo escribe en un registro
Ej:

```
     ..........
     PUSH ECX       ;Guardo el contenido de ECX en la pila
     MOV  ECX,100
     MOV  ESI,0
PUTDIG:             ;Rutina para llenar un vector números
     MOV  [VEC+ESI],ESI
     INC  ESI
     LOOP PUTDIG    ;Al finalizar el ciclo ECX contiene 0
     POP  ECX       ;Recupero el contenido original de ECX
     ..........
```

Manejo de cadenas

CMPS – CMPSB – CMPSW

Las instrucciones del tipo **CMPS comparan** el **contenido** de 2 áreas de memoria. Una de las áreas debe estas **apuntada** por **ESI** y la otra por **EDI**.

Para comparar áreas de memoria mayores a un byte debe indicarse en el registro **ECX** la **longitud** de las mismas y la instrucción debe ser precedida por el modificador **REPE** (repetir si es igual).

La instrucción incrementará en 1 los registros ESI y EDI y el modificador hará que la instrucción se repita hasta que las áreas apuntadas sean distintas o hasta que se cumpla la cantidad de repeticiones indicadas en ECX.

Si usamos **CMPSB** la comparación se hará de a **un byte**, y en **ECX** deberá cargarse la cantidad de **bytes** a comparar.
Si usamos **CMPSW** la comparación se hará de a **dos bytes**, y en **ECX** deberá cargarse la cantidad de **duplas** de **bytes** a comparar.
Ej:

```
     ..........
     MOV  ECX,4              ;Cantidad de bytes a comparar
     MOV  ESI,MSGUNO         ;Desplazamiento origen
     MOV  EDI,MSGDOS         ;Desplazamiento destino
     REPE CMPSB              ;Compara byte por byte
     JE   IGUALES
     ..........
```

MOVS – MOVSB – MOVSW

Las instrucciones del tipo **MOVS copian** el **contenido** de un área de memoria (origen) a otra área de memoria (destino). El área origen debe estar **apuntada** por los registros **ESI** y el área destino por **EDI**.

Para copiar áreas de memoria mayores a un byte debe indicarse en el registro **ECX** la **longitud** de las mismas y la instrucción debe ser precedida por el modificador **REP** (repetir).

La instrucción incrementará en 1 los registros ESI y EDI y el modificador hará que la instrucción se repita hasta que se cumpla la cantidad indicada en el registro ECX.

Si usamos **MOVSB** la copia se hará de a **un byte** y en **ECX** deberá cargarse la cantidad de **bytes** a copiar.
Si usamos **MOVSW** la copia se hará de a **2 bytes** y en **ECX** deberá cargarse la cantidad de **duplas de bytes** a copiar.
Ej:

```
..........
MOV  ECX,4          ;Cantidad de bytes a copiar
LEA  ESI,[MSGORI]   ;Desplazamiento origen
LEA  EDI,[MSGDES]   ;Desplazamiento destino
REP  MOVSB          ;Copia byte por byte
..........
```

Entrada / Salida

Impresión de mensaje en pantalla

Para imprimir un mensaje en la pantalla se hace uso de una rutina externa ***_printf***, de la cuál hace falta importar dentro del programa para ser utilizado.

Ej:

```
global      _main
extern      _printf

section     .data
      mensaje db 'Hola Mundo!',0 ;campo con el string a imprimir.
                                 ;Debe finalizar con 0 binario
section     .bss

section     .text
_main:
      push  mensaje     ;Parametro 1: direccion del mensaje a imprimir
      call  _printf
      add   esp,4       ;Coloco puntero a la pila donde estaba

      ret
```

Existe otra manera de usar _printf con parámetros.

Ej:

```
global      _main
extern      _printf

section     .data
      mensaje     db    'Imprimo con printf el numero %d',0
      numero      dd    1234
section     .bss

section     .text
_main:
      push  dword[numero];Parametro 2: dato formateado para imprimir
      push  mensaje3    ; Parametro 1: direccion del mensaje a imprimir
      call  _printf
      add   esp,8       ;Coloco el puntero a la pila donde estaba

      ret
```

Otra manera de imprimir un mensaje por pantalla es utilizar otra rutina externa llamada **_puts**, también es necesario importarlo dentro del programa.

Ej:

```
global      _main
extern      _puts

section     .data
      mensaje db 'Hola Mundo!',0 ;campo con el string a imprimir.
                                 ;Debe finalizar con 0 binario
section     .bss

section     .text
_main:
      push  mensaje ;Parametro 1: direccion del mensaje a imprimir
      call  _puts   ;Imprime hasta que llega al 0 y agrega fin de linea
      add   esp,4   ;Coloco puntero a la pila donde estaba

      ret
```

Lectura por teclado

Para leer datos mediante el teclado se hace uso de rutinas externas ***_gets*** y ***_scanf***.

Ej:

```
global      _main
extern      _printf
extern      _puts
extern      _gets
extern      _scanf

section  .data
      msgIngN     db    'Ingrese su nombre: ',0
      msgSalN     db    'Usted se llama ',0
      msgIngE     db    'Ingrese su edad ',0
      msgSalE     db    'Usted tiene %d años',0
      edadFormat  db    '%d'

section  .bss
      nombre      resb  20
      edad        resd  1 ;Se define como DOBLE-WORD porque se colocara
                          ;en el stack para imprimir x pantalla

section .text
_main:
      push  msgIngN
      call  _printf
      add   esp,4

      push  nombre ; Parametro 1:direccion de memoria del campo
                   ; donde se guarda lo ingresado por teclado
      call  _gets  ;Lee de teclado y lo guarda como string
                  ;hasta que se ingresa fin de linea.
                  ;Agrega un 0 binario al final
```

```
    add   esp,4

    push  msgSalN
    call  _printf
    add   esp,4

    push  nombre
    call  _puts
    add   esp,4

    push  msgIngE
    call  _printf
    add   esp,4

    push  edad       ;Parametro 2: direccion de memoria del campo
                     ;que guarda lo ingresado por teclado
    push  edadFormat ;Parametro 1: direccion de memoria del campo que
                     ;indica el formato en el que se almacena
                     ;lo ingresado
    call  _scanf     ;Lee de teclado y lo guarda en el formato
                     ;indicado hasta que se ingresa fin de linea
    add   esp,8

    push  dword[edad] ;Parametro 2: campo que se encuentra en el
                      ;formato indicado q se imprime por pantalla
    push  msgSalE     ;Parametro 1: direccion de memoria del campo a
                      ;imprimir
    call  _printf
    add   esp,8

  ret
```

Manejo de archivos

Apertura

Para abrir un archivo se usa la rutina externa ***_fopen***.
Y deben indicarse los siguientes datos:

- En el parámetro 1, la dirección del nombre de archivo.
- En el parámetro 2, el modo de la apertura del archivo

El resultado de la operación deja el file handle en el registro EAX
Si la apertura resulta fallida, en el registro EAX quedará menor o igual que 0.

Ej:

```
global      _main
extern      _printf
extern      _fopen

section     .data
      fileName    db    "archivoTexto.txt",0
      mode        db    "r+",0
      fileHandle  dd    0
      msgErrOpen  db    "Error en apertura de archivo",0

section     .bss
section     .text
_main:
      .....
      ;Abro archivo para lectura y escritura
      push  mode     ;Parametro 2: dir string modo de apertura
      push  fileName ;Parametro 1: dir nombre del archivo
      call  _fopen   ;ABRO archivo y deja el handle en EAX
      add   esp,8

      cmp   eax,0                  ;Error en apertura?
      jle   errorOpen

      mov   [fileHandle],eax
      .....

errorOpen:
      push  msgErrOpen
      call  _printf
      add   esp,4

endProg:
      ret
```

Escritura (Archivo Texto)

Para escribir en un archivo de texto, se usa la rutina externa ***_fputs***.
Y deben indicarse los siguientes datos:

- En el parámetro 1, la dirección del área de memoria a escribir en el archivo.
- En el parámetro 2, el handle del archivo

Ej:

```
      ….
      ….
      ;Escribo una linea al archivo
write:
      push  dword[fileHandle];Parametro 2: handle del archivo
      push  linea           ;Parametro 1: dir area de memoria a copiar
      call  _fputs          ;ESCRIBO archivo.
      add   esp,8
      ….
```

Escritura (Archivo Binario)

Cabe mencionar para una correcta escritura de un archivo binario, es necesario abrir el archivo con el modo "*append binary (ab)*" o "write binary (wb)".
Ej:

```
...
section     .data
      mode        db    "ab+",0 ;modo append binary and read
                                ;(actualizacion y lectura)
...
```

Para escribir en un archivo binario, se utiliza la rutina externa ***_fwrite***.
Y deben indicarse los siguientes datos:

- En el parámetro 1, la dirección del área de memoria o registro a escribir en el archivo.
- En el parámetro 2, la longitud de un registro a escribir.
- En el parámetro 3, cantidad de registros a escribir.
- En el parámetro 4, el handle del archivo.

Ej:

```
      ….
      ….
      ;     Agrego un registro
      push  dword[fileHandle] ;Parametro 4: handle del archivo
      push  1                 ;Parametro 3: cantidad de registros
      push  12                ;Parametro 2: longitud del registro
      push  registro1         ;Parametro 1: dir area de memoria
                              ;donde se copia
      call  _fwrite       ;Escribo registro.
                          ;Devuelve en eax la cantidad de bytes leidos
      add   esp,16
      ….
```

Lectura (Archivo Texto)

Para la lectura de un archivo de texto, se utiliza la rutina externa ***_fgets***.
Y deben indicarse los siguientes datos:

- En el parámetro 1, la dirección del área de memoria a copiar del archivo.
- En el parámetro 2, la cantidad de bytes máximas a leer (o hasta fin de línea).
- En el parámetro 3, el handle del archivo.

Ej:

```
      ….
      ….
      ;     Leo registro
read:
      push  dword[fileHandle] ;Parametro 3: handle del archivo
      push  80                ;Parametro 2: cantidad de bytes maximas a
                              ;leer (o hasta fin de linea)
      push  registro          ;Parametro 1: dir area de memoria donde
                              ;se copia
      call  _fgets            ;LEO registro. Devuelve en eax
                              ;la cantidad de bytes leidos
      add   esp,12

      cmp   eax,0             ;ver si llegó a fin del archivo
      jle   eof
      ….
```

Lectura (Archivo Binario)

Cabe mencionar para una correcta lectura de un archivo binario, es necesario abrir el archivo con el modo "read binary (rb)".

```
...
section     .data
      mode        db    "rb",0 ;modo read binary
...
```

Para la lectura de un archivo binario, se utiliza la rutina externa ***_fread***.
Y deben indicarse los siguientes datos:

- En el parámetro 1, la dirección del área de memoria a copiar del archivo.
- En el parámetro 2, la longitud de un registro a leer.
- En el parámetro 3, la cantidad de registros a leer.
- En el parámetro 4, el handle del archivo.

Ej:

```
      ….
      ;     Leer un registro
read:
      push  dword[fileHandle] ;Parametro 4: handle del archivo
      push  1                 ;Parametro 3: cantidad de registros
      push  12                ;Parametro 2: longitud del registro
      push  registro          ;Parametro 1: dir area de memoria
                              ;donde se copia del archivo
      call  _fread        ;Leer registro.
                          ;Devuelve en eax la cantidad de bytes leidos
      add   esp,16

      cmp   eax,0         ;ver si es fin de archivo?
      jle   eof
      ….
```

Cierre

Para cerrar un archivo se usa la rutina externa ***_fclose***, y debe indicar únicamente el handle del archivo a cerrar.
Si el cierre resulta fallido, en el registro EAX quedará menor o igual que 0.

Ej:

```
      ….
      ;     cierro el archivo
      push  dword[filehandle] ;parametro 1: handle del archivo
      call  _fclose           ;cierro archivo
      add   esp,4

      cmp   eax,0             ;Error en el cierre?
      jle   errorClose
      ….

errorCierre:
      push  msgErrClose
      call  _printf
      add   esp,4

endProg:
      ret
```