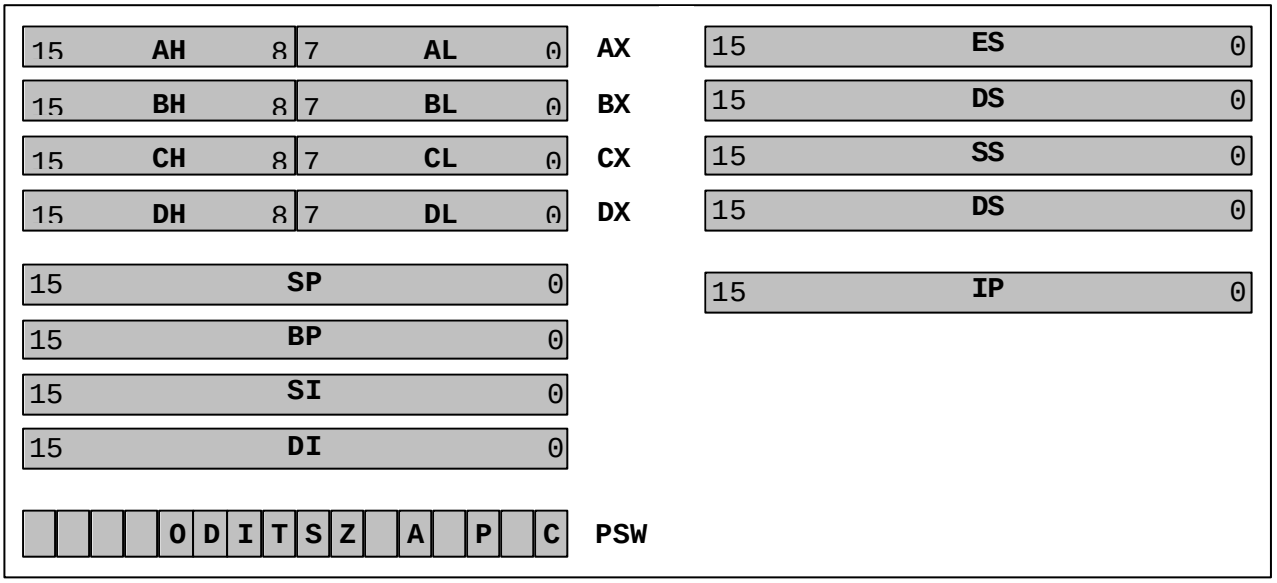
El microprocesador de 16 bits 8086

INTRODUCCIÓN

En 1978 Intel sacó al mercado el 8086, con un bus de datos de 16 bits y capaz de direccionar hasta 1 Mb de memoria. La importancia del 8086 se debe a que fue elegido por IBM para desarrollar el Personal Computer, que fue tomado como estándar por casi todos los fabricantes de ordenadores. Posteriormente Intel fabricó sucesivamente el 80186, el 80286, el 80386, el i486 y los Pentiums, manteniendo en todos ellos la compatibilidad software con los anteriores.

MODELO DE PROGRAMACIÓN



TERMINALES EN EL 8086

- Alimentación única de +5V y dos terminales de GND.
- Entrada única de reloj CLK que requiere una relación cíclica del 30%.
- Una línea de entrada de RESET.
- Terminal de entrada MN/MX para seleccionar los modos mínimo o máximo en la CPU. En el modo mínimo la CPU genera las señales de control del sistema, está pensado para sistemas sencillos. En el modo máximo, pensado para aplicaciones complejas, es necesario un circuito controlador de bus (8288).
- Bus de datos de 16 bits, cuyas líneas están multiplexadas con las 16 líneas de menor peso del bus de direcciones (AD0-AD15).
- Bus de direcciones de 20 bits. Las cuatro líneas de mayor peso (D16-D19) están multiplexadas con cuatro de las 8 líneas indicadoras del estado del procesador.
- Ocho líneas de salida indicadoras del estado del procesador (S0-S7), multiplexadas con distintas señales.
- Terminal de salida /BHE (multipexada con S7) que habilita la parte alta (D8-D15) del bus de datos.
- Terminal de entrada READY utilizada para sincronizar el procesador con periféricos o memorias más lentas.

- Terminal de salida /RD para indicar operación de lectura y /WR para indicar operación de escritura en memoria o puerto de entrada/salida.
- Terminal de salida M/IO que indica si el acceso es a memoria o a un puerto.
- Terminal de entrada /TEST que es examinado en la instrucción `WAIT` para determinar si se para o no el procesador hasta una interrupción.
- Dos líneas de entrada de interrupciones: INTR (enmascarable) y NMI (no enmascarable).
- Línea de salida /INTA de reconocimiento de petición de interrupción enmascarable.
- Línea de salida ALE que indica dirección válida en el bus de direcciones/datos.
- Terminal de salida /DEN que valida el traspaso de datos en el bus de datos.
- Línea de salida DT/R que indica el sentido de la transferencia en el bus de datos.
- Terminal de entrada HOLD para que otro dispositivo tome el control del bus y terminal de salida HLDA de cesión del citado bus.
- Terminales de salida que informan sobre el estado del procesador: S0-S2 indican la operación que se está ejecutando, S3-S3 están multiplexadas con A16-A17 e indican el registro de segmento que se está utilizando y S5 (multiplexada con A18) reproduce el flag I.
- Las señales RQ/GT0 RQ/GT1 permiten la comunicación con otros procesadores del mismo bus local.
- La señal de salida /LOCK se activa con las instrucciones que incluyen el prefijo LOCK, para indicar que no es posible la cesión del bus a ningún otro procesador.
- Las líneas QS0-QS1 informan sobre el estado de la cola interna de instrucciones.

Las señales M/IO, /WR, /INTA, ALE, DT/R, /DEN, HOLD y HLDA son exclusivas del modo mínimo. Las señales S0-S2, RQ/GT0, RQ/GT1, /LOCK y QS0-QS1 solo están disponibles en modo máximo

EL REGISTRO DE ESTADO (PSW)

- `C` (*Carry*) Flag de acarreo.
- `P` (*Parity*) Flag indicador de paridad.
- `A` (*Auxiliar*) Flag de acarreo auxiliar (del bit 3 al bit 4).
- `Z` (*Zero*) Flag indicador de cero.
- `S` (*Sign*) Flag indicador de signo del resultado.
- `T` (*Trap*) Flag indicador de ejecución paso a paso.
- `I` (*Interrupt*) Flag habilitación de interrupciones enmascarables.
- `D` (*Direction*) Flag de dirección ascendente/descendente en instrucciones de cadenas.
- `O` (*Overflow*) Flag indicador de desbordamiento.

INTERRUPCIONES

El 8086 puede tratar 256 tipos diferentes de interrupciones, que se numeran en hexadecimal entre `0x00` y `0xFF` y, según el modo en el que se producen, pueden ser *internas* o *externas*. Las **interrupciones externas** son aquellas producidas por señales eléctricas procedentes del exterior del microprocesador, normalmente producidas por los periféricos. Las **interrupciones internas** son las producidas por la propia CPU en la ejecución de un programa (por ejemplo ante el intento de división por cero) o por una instrucción específica (son las *interrupciones programadas* que se producen cuando se ejecuta la instrucción `INT`).

Para poder realizar la gestión de todas las interrupciones, los primeros 1024 bytes de memoria están reservadas para los *vectores de interrupción* que almacenan los valores que se

deben cargar en los registros `IP` y `CS`, para ejecutar la rutina de atención a la interrupción correspondiente (la interrupción N tendrá su vector de interrupción a partir del byte de dirección Nx4).

El 8086 dispone de dos líneas de petición de interrupción hardware:

- La línea **NMI**, que provoca una interrupción no enmascarable que es atendida siempre mediante la rutina de vector 2.
- La línea **INTR**, que provoca una interrupción enmascarable, que solo será atendida si el flag I está a 1, recibiendo en las 8 líneas de menor peso del bus de datos el número del vector de interrupción. Esta línea es controlada por el circuito PIC (*Programmable Interrupt Controller*) que se encarga también de situar el número de interrupción en el bus de datos.

La propia CPU del 8086 es capaz de generar automáticamente tres interrupciones internas:

- La **interrupción 0** que se genera cuando se produce un intento de división por cero.
- La **interrupción 1** que se genera después de ejecutarse cada instrucción si el flag T del registro de estado está a 1, lo que permite la ejecución paso a paso y la depuración de programas.
- La **interrupción 4** es la interrupción de overflow y se genera cuando se produce una operación aritmética con desbordamiento.

Mediante la instrucción `INT` *`num`* se genera la interrupción representada por `num`, que será atendida siempre, independientemente del estado del flag I.

ORGANIZACIÓN DE LA MEMORIA EN EL 8086

Aunque el 8086 dispone de 20 líneas en el bus de direcciones, la dirección de memoria se obtiene a partir de dos registros de 16 bits mediante un sistema de *segmentación de la memoria*, que equivale a subdividir la memoria en **segmentos** de 64 Kb, de modo que un *registro de segmento* determina la dirección de comienzo de ese segmento de memoria y la posición relativa (off-set o desplazamiento) la fija otro registro también de 16 bits. Así mediante la combinación de ambos registros es posible direccionar cada una de las 65536 posiciones del segmento de 64 Kb.

Mediante la segmentación, los 5 dígitos hexadecimales necesarios para direccionar hasta 1 Mb, se obtienen desplazando un dígito a la izquierda el número representado en el registro de segmento y sumándolo al número que representa el *desplazamiento* con respecto al origen, es decir:

```
DIRECCIÓN FÍSICA = SEGMENTO * 16 + DESPLAZAMIENTO
```

Cuando se utiliza el método segmentado, las direcciones se expresan mediante el número base y el *offset* o *desplazamiento*, ambos de 4 dígitos hexadecimales y separados por dos puntos. Por ejemplo la dirección física `56F7A` se puede representar escribiendo `56F7:000A` ó `5000:6F7A` ó `5600:0F7A`, ... Con este procedimiento, toda la memoria puede dividirse en segmentos, los cuales comienzan siempre en una dirección múltiplo de 16 y que, por supuesto, pueden solaparse.

A la vista del su modelo de programación, el 8086 puede operar con datos de tipo byte y de tipo palabra que se almacenan en memoria según el convenio de Intel, es decir, con la parte

menos significativa siempre en la dirección más baja, de modo que, por ejemplo, si suponemos que `AX=1234` y que `TABLA` representa a la dirección 100h (dentro del segmento de datos), la instrucción `MOV TABLA,AX` hará que se almacene 34h en la dirección 100h y 12 en la 101h.

LENGUAJE ENSAMBLADOR DEL 8086

El ensamblador del 8086 establece para las líneas de programa la siguiente sintaxis:

`Etiqueta: Código_de_Operación Operandos ;Comentarios`

El campo **`Etiqueta`** es opcional y debe finalizar con dos puntos «**:**». En el campo correspondiente al **`Código_de_Operación`** debe incluirse el mnemónico correspondiente a la instrucción. El campo **`Operandos`** incluirá uno o dos operandos, según la instrucción. Si son dos los operandos necesarios, se incluirán separados por una coma de acuerdo con la sintaxis `Destino,Fuente`. El campo **`Comentarios`** debe ir precedido del signo *punto y coma*.

A continuación se muestra la estructura general que debe darse a un programa en ensamblador, con las **directivas del ensamblador** de uso más frecuente.

```
        TITLE ESTRUCT.ASM

        COMMENT % Comentarios en varias líneas %
        ;Es comentario todo lo que sigue a un signo de punto y coma

        DOSSEG
        .MODEL SMALL     ;Tipo de programa

        .STACK 100h      ;Reserva de memoria para la pila

        .DATA            ;Segmento de datos del programa
nombre1  EQU 00h         ;El símbolo nombre1 será sustituido por 00h
nombre2  EQU 001h
var1    DB 00h           ;Un un byte de memoria inicializado a 00h
var2    DB  ?            ;Un byte de memoria sin inicializar
var3    DW  0FFFFh       ;Reserva una palabra y la inicializa a FFFFh
array1  DB  10 DUP (?)   ;10 bytes sin inicializar a partir de array1
tabla1  DB  00,01,02,03  ;Bytes de memoria inicializados
mensaje1 DB  "Mensaje:...",$"
                         ;El servicio 09h-INT 21h escribe cadenas de
                         ;caracteres en pantalla hasta el signo $.
buffer  DB 255,?,255 DUP(?)
                         ;Buffer de 255 caracteres preparado para el
                         ;0Ah-INT 21h.

        .CODE            ;Segmento de código del programa

NomMacro1  MACRO param1, param2, ...
                         ;Definicion de una macro
        LOCAL eti1       ;Etiquetas propia de la macro
        ...
eti1:       [código]
        ...
        ENDM             ;Final de la macro
```

```
NomProc1    PROC          ;Comienzo de un procedimiento
        [código]
        ...
        RET
NomProc1 ENDP             ;Final del procedimiento

inicio:  MOV AX, @DATA    ;Comienzo del módulo principal del programa
        MOV DS, AX        ;Inicialización del registro DS
etiq1:   [código]
        ...
        NomMacro1         ;Llamada a la macro
        ...
bucle:   [código]
        ...
        CALLNomProc1      ;Llamada a la subrutina (procedimiento)
        JMP bucle         ;Instrucción de salto incondicional
        ...
FIN:     MOV AH, 4CH      ;Servicio 4Ch-INT 21h para regresar al DOS.
        INT 21h

        END inicio        ;Fin del código e indicación del punto de
                          ;comienzo de la ejecución del programa.
```

REPERTORIO DE INSTRUCCIONES

MNEMÓNICO	OPERACIÓN	MNEMÓNICO	OPERACIÓN
AAA	ASCII Adjust after Addition	JS	Jump If Sign
AAD	ASCII Adjust before Division	JZ	Jump If Zero
AAM	ASCII Adjust after Multiply	LAHF	Load Register AH from
AAS	ASCII Adjust after Subtraction	LDS	Load Pointer Using DS
ADC	Add with Carry	LEA	Load Effective Address
ADD	Addition	LES	Load Pointer Using ES
AND	Logical AND	LOCK	Lock the Bus
CALL	Call Procedure	LODS	Load String (Byte or Word)
CBW	Convert Byte to Word	LODSB	Load String Byte
CLC	Clear Carry Flag	LODSW	Load String Word
CLD	Clear Direction Flag	LOOP	Loop on Count
CLI	Clear Interrupt-Enable Flag	LOOPE	Loop While Equal
CMC	Complement Carry Flag	LOOPNE	Loop While Not Equal
CMP	Compare	LOOPNZ	Loop While Not Zero
CMPS	Compare String (Byte or Word)	LOOPZ	Loop While Zero
CMPSB	Compare String Byte	MOV	Move (Byte or Word)
CMPSW	Compare String Word	MOVS	Move String (Byte or Word)
CWD	Convert Word to Doubleword	MOVSB	Move String Byte
DAA	Decimal Adjust after Addition	MOVSW	Move String Word
DAS	Decimal Adjust after Subtraction	MUL	Multiply, Unsigned
DEC	Decrement	NEG	Negate
DIV	Divide, Unsigned	NOP	No Operation
ESC	Escape	NOT	Logical NOT
HLT	Halt	OR	Logical OR
IDIV	Integer Divide, Signed	OUT	Output to Port

MNEMÓNICO	OPERACIÓN	MNEMÓNICO	OPERACIÓN
IMUL	Integer Multiply, Signed	POP	Pop a Word from the Stack
IN	Input Byte or Word	POPF	Pop Flags from the Stack
INC	Increment	PUSH	Push Word onto Stack
INT	Interrupt	PUSHF	Push Flags onto Stack
INTO	Interrupt on Overflow	RCL	Rotate through Carry Left
IRET	Interrupt Return	RCR	Rotate through Carry Right
JA	Jump If Above	REP	Repeat
JAE	Jump If Above or Equal	REPE	Repeat While Equal
JB	Jump If Below	REPNE	Repeat While Not Equal
JBE	Jump If Below or Equal	REPNZ	Repeat While Not Zero
JC	Jump If Carry	REPZ	Repeat While Zero
JCXZ	Jump if CX Register Zero	RET	Return from Procedure
JE	Jump If Equal	ROL	Rotate Left
JG	Jump If Greater	ROR	Rotate Right
JGE	Jump If Greater or Equal	SAHF	Store Register AH into
JL	Jump If Less	SAL	Shift Arithmetic Left
JLE	Jump If Less or Equal	SAR	Shift Arithmetic Right
JMP	Jump Unconditionally	SBB	Subtract with Borrow
JNA	Jump If Not Above	SCAS	Scan String (Byte or Word)
JNAE	Jump If Not Above or Equal	SCASB	Scan String Byte
JNB	Jump If Not Below	SCASW	Scan String Word
JNBE	Jump If Not Below or Equal	SHL	Shift Logical Left
JNC	Jump If No Carry	SHR	Shift Logical Right
JNE	Jump If Not Equal	STC	Set Carry Flag
JNG	Jump If Not Greater	STD	Set Direction Flag
JNGE	Jump If Not Greater or Equal	STI	Set Interrupt Enable Flag
JNL	Jump If Not Less	STOS	Store String (Byte or Word)
JNLE	Jump If Not Less or Equal	STOSB	Store String Byte
JNO	Jump If No Overflow	STOSW	Store String Word
JNP	Jump If No Parity	SUB	Subtract
JNS	Jump If No Sign	TEST	Test
JNZ	Jump If Not Zero	WAIT	Wait
JO	Jump If Overflow	XCHG	Exchange Registers
JP	Jump If Parity	XLAT	Translate
JPE	Jump If Parity Even	XOR	Exclusive OR
JPO	Jump If Parity Odd		

MODOS DE DIRECCIONAMIENTO

En el 8086 las direcciones se construyen mediante la combinación de un registro de segmento y un registro de offset o desplazamiento (SEG:DESP), de acuerdo con las posibilidades que se muestran en la siguiente tabla:

Tipo de referencia a memoria	Segmento por defecto	Segmento alternativo	Offset o desplazamiento
Búsqueda de instrucción	CS	--	IP
Operación sobre la pila (stack)	SS	--	SP

Tipo de referencia a memoria	Segmento por defecto	Segmento alternativo	Offset o desplazamiento
Variables	DS	CS - ES - SS	Dirección efectiva
Cadenas fuente	DS	CS - ES - SS	SI
Cadenas destino	ES	--	DI
BP como registro base	SS	CS - ES - SS	Dirección efectiva

- IMPLÍCITO: La propia instrucción indica la situación de los operandos. No hay dirección efectiva (DE).
  `TAX` Pasa el dato de A a X.

- INMEDIATO: No existe dirección efectiva pues el dato viene incluido en la propia instrucción.
  `MOV AX, 1234h` Carga en el registro AX el dato 1234h.

- POR REGISTRO: No existe dirección efectiva pues el dato está en el registro indicado.
  `MOV BX,AX` Copia el contenido de AX en BX

- DIRECTO: La dirección efectiva está incluida en la propia instrucción, bien mediante una expresión numérica o bien mediante una etiqueta. El segmento por defecto es DS.
  `MOV AX, [1234h]` Guarda en el registro AL el contenido de la dirección 1234h y en AH el dato contenido en la dirección 1235h.
  `MOV BL, DATO` Guarda en BL el byte apuntado por la etiqueta DATO.

- INDIRECTO MEDIANTE REGISTRO: La dirección efectiva es el contenido de uno de los registros BX, SI, DI ó BP. Si se utilizan BX, DI o SI el registro de segmento por defecto es DS y si se utiliza BP el registro de segmento es SS.
  `MOV DX,[BX]` Lleva el dato de contenido en la posición de memoria contenida en BX a DL y el contenido en la siguiente a DH.

- RELATIVO A BASE: La dirección efectiva se obtiene al sumar un desplazamiento que se incluye en la instrucción al contenido del registro BX o BP. El registro de segmento por defecto para BX es DS y para BP es SS.
  `MOV DL,[BX+7Ah]` Guarda en DL el contenido de la posición de memoria cuya dirección resulta de sumar 7Ah al contenido de BX. Puede escribirse `MOV DL,7Ah[BX]` ó `MOV DL,[SI+7Ah]`

- DIRECTO INDEXADO: La dirección efectiva se obtiene al sumar un desplazamiento que se incluye en la instrucción al contenido del registro DI o SI. El registro de segmento por defecto es DS.
  `MOV AL,TABLA[DI]` Guarda en AL el contenido de la posición de memoria cuya dirección se obtiene al sumar a la dirección representada por tabla el contenido del registro DI.

- INDEXADO RELATIVO A BASE: La dirección efectiva se obtiene un desplazamiento que se incluye en la instrucción al contenido del registro SI o DI y al contenido del registro BX o BP. Cuando se utiliza BX el registro de segmento por defecto es DS y si se utiliza BP entonces es SS.

`MOV AH,[BP+SI+7]` Copia en el registro AH el contenido de la posición de memoria cuya dirección resulta de sumar los contenidos de BP y SI y el número 7. También puede escribirse `MOV AH,7[BP+SI]` ó `MOV AH,7[BP][SI]`

- DIRECCIONAMIENTO DE CADENAS: Se utiliza en instrucciones sobre cadenas. La dirección efectiva de los elementos de la cadena fuente viene dada por DS:SI y la dirección efectiva de la cadena destino viene dada por ES:DI.

  `MOVSB` Copia la cadena desde cuya dirección se encuentra en DS:[SI] a la posición de memoria cuya dirección se encuentra en ES:[DI].

EL PROGRAMA MICROSOFT CODEVIEW

El programa **CodeView** es una utilidad para la depuración de programas para entornos tipo PC, que se incluye en el Microsoft Macro Assembler versión 5.1.

Para el mejor aprovechamiento de las posibilidades del CodeView es preciso generar el programa ejecutable con las siguientes opciones:

- Ensamblar: `MASM /Zi` *`PROGRAMA`*`;`
- Enlazar: `LINK /CO PROGRAMA;`

EJEMPLOS DE PROGRAMAS PARA EL 80x86

- Programa que suma sin signo los bytes contenidos en dos posiciones de memoria y guarda el resultado en dos bytes, de acuerdo con el criterio establecido por INTEL para el almacenamiento en memoria de datos de mas de un byte.

```
        DOSSEG
        .MODEL SMALL
        .STACK 100h
        .DATA
Dato1   db      0FFh
Dato2   db      0FFh
Resul   dw      ?
       .CODE
Inicio: mov  ax,@data
        mov  ds,ax
        xor  ax,ax
        mov  al,Dato1
        add  al,Dato2
        adc  ah,0       ;El acarreo al byte alto
        mov  Resul,ax   ;Almacena el resultado
Fin:    mov  ah,4Ch
        int  21h
        END  Inicio
```

- Programa que inicializa a FFh un total de 100h posiciones de memoria.

```
        DOSSEG
        .MODEL SMALL
        .STACK 100h
```

```
        .DATA
Mem     db      100h DUP (?)    ;Zona de memoria a inicializar
        .CODE
Inicio: mov  ax,@data
        mov  ds,ax
        mov  al,0FFh    ;Dato a almacenar
        mov  cx,100h    ;Contador
        xor  si,si      ;Índice
Bucle:  mov  Mem[si],al
        inc  si
        loop Bucle      ;Repite si no terminó
Fin:    mov  ah,4Ch
        int  21h
        END  Inicio
```

- Programa que inicializa un bloque de memoria de 100h bytes con los valores 0, 1, 2, ..., FFh.

```
DOSSEG
        .MODEL SMALL
        .STACK 100h
        .DATA
Mem     db      100h DUP (?)    ;Zona de memoria a inicializar
        .CODE
Inicio: mov  ax,@data
        mov  ds,ax
        xor  al,al      ;Dato a almacenar
        mov  cx,100h    ;Contador
        xor  si,si      ;Índice
Bucle:  mov  Mem[si],al
        inc  al         ;Cambia el dato
        inc  si
        loop Bucle      ;Repite si no terminó
Fin:    mov  ah,4Ch
        int  21h
        END  Inicio
```

- Programa anterior modificado para que la inicialización se realice en orden inverso, es decir, con los valores FFh, FEh, FDh, ..,1,0.

```
       DOSSEG
        .MODEL SMALL
        .STACK 100h
        .DATA
Mem     db      100h DUP (?)    ;Zona de memoria a inicializar
        .CODE
Inicio: mov  ax,@data
        mov  ds,ax
        mov  al,0FFh    ;Dato a almacenar
        mov  cx,100h    ;Contador
        xor  si,si      ;Índice
Bucle:  mov  Mem[si],al
        dec  al         ;Cambia el dato
        inc  si
        loop Bucle      ;Repite si no terminó
Fin:    mov  ah,4Ch
```

```
        int  21h
        END  Inicio
```

- Programa que copia un bloque de 100h bytes de memoria.

```
        DOSSEG
        .MODEL SMALL
        .STACK 100h
        .DATA
Origen  db      100h DUP (0Fh)  ;Zona de memoria a copiar
Destin  db      100h DUP (?)    ;Destino
        .CODE
Inicio: mov  ax,@data
        mov  ds,ax
        mov  cx,100h    ;Contador
        xor  si,si      ;Índice
Bucle:  mov  al,Origen[si]      ;Lee
        mov  Destin[si],al      ;Escribe
        inc  si
        loop Bucle      ;Repite si no terminó
Fin:    mov  ah,4Ch
        int  21h
        END  Inicio
```

- Programa que localiza el mayor y el menor de los 16 datos de un byte almacenados en una tabla de datos.

```
        DOSSEG
         .MODEL SMALL
         .STACK 100h
         .DATA
Tabla   db      41h, 5h, 69h, 0FEh, 0Fh, 28h, 0EFh, 011h
        db      22h, 3h, 25h, 0E4h, 77h, 0Ah,  78h,  015h
Mayor   db      ?
Menor   db      ?
         .CODE              ;Números SIN signo
Inicio: mov  ax,@data
        mov  ds,ax
        xor  si,si      ;Índice
        mov  cx,0Fh     ;Número de datos a comparar
        mov  ah,Tabla[si]       ;En ah el mayor
        mov  al,Tabla[si]       ;En al el menor
Bucle:  inc  si
        mov  bl,Tabla[si]
        cmp  ah,bl
        ja   NoCam1             ;Salta si ah es mayor: no cambia
        mov  ah,Tabla[si]
        jmp  Sigue
NoCam1: cmp  al,bl
        jb   NoCam2             ;Salta si a al es menor: no cambia
        mov  al,Tabla[si]
NoCam2:
Sigue:  loop Bucle              ;Repite
        mov  Mayor,ah
        mov  Menor,al
Fin:    mov  ah,4Ch
```

```
        int  21h
        END  Inicio

DOSSEG
        .MODEL SMALL
        .STACK 100h
        .DATA
Tabla   db      41h, 5h, 69h, 0FEh, 0Fh, 28h, 0EFh, 011h
        db      22h, 3h, 25h, 0E4h, 77h, 0Ah,  78h,  015h
Mayor   db      ?
Menor   db      ?
        .CODE             ;Números CON signo
Inicio: mov  ax,@data
        mov  ds,ax
        xor  si,si      ;Índice
        mov  cx,0Fh     ;Número de datos a comparar
        mov  ah,Tabla[si]       ;En ah el mayor
        mov  al,Tabla[si]       ;En al el menor
Bucle:  inc  si
        mov  bl,Tabla[si]
        cmp  ah,bl
        jg   NoCam1             ;Salta si ah es mayor: no cambia
        mov  ah,Tabla[si]
        jmp  Sigue
NoCam1: cmp  al,bl
        jl   NoCam2             ;Salta si al es menor: no cambia
        mov  al,Tabla[si]
NoCam2:
Sigue:  loop Bucle              ;Repite
        mov  Mayor,ah
        mov  Menor,al
Fin:    mov  ah,4Ch
        int  21h
        END  Inicio
```

- Programa que suma dos datos de 32 bytes cada uno y almacena el resultado en 33 bytes.

```
       DOSSEG
        .MODEL SMALL
        .STACK 100h
        .DATA
Dato1   DB      32      DUP (0FFh)
Dato2   DB      32      DUP (0FFh)
Numby   EQU     $-Dato2
Resul   DB      33
        .CODE
Inicio: mov  ax,@data
        mov  ds,ax
        xor  si,si      ;Índice
        mov  cx,NumBy   ;Número de bytes a sumar
        clc             ;Acarreo inicial a cero
Bucle:  mov  al,Dato1[si]
        adc  al,Dato2[si]
        mov  Resul[si],al
        inc  si
```

```
        loop bucle      ;Repite hasta terminar
        mov  al, 0      ;No se utiliza XOR para no perder el acarreo
        adc  al,0       ;El acarreo al byte alto
        mov  Resul[si],al       ;Último byte del resultado
Fin:    mov  ah,4Ch
        int  21h
        END  Inicio
```