ArquiSim

MANUAL DE USUARIO

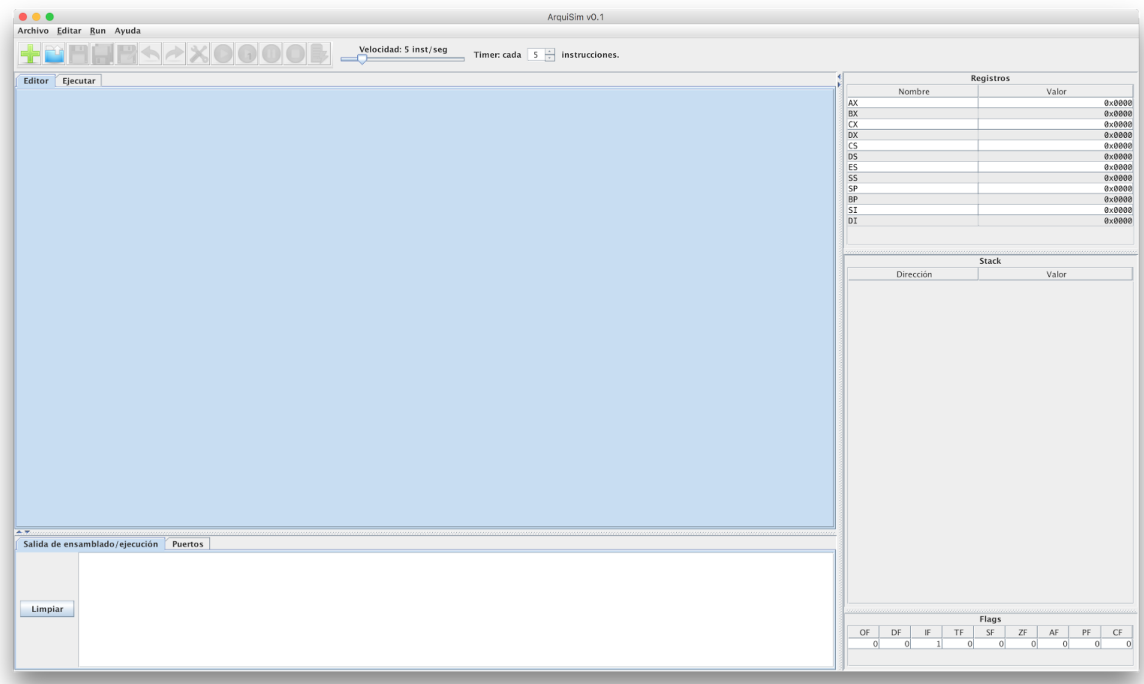
ÍNDICE

1. Introducción…………………………………………………………………………………..3
2. Estructura de archivo de entrada.…………………………………………………………….5
3. Compilación…………………………………………………………………………………..8
4. Ejecución………………………………………………………………………………….10
5. Entrada/Salida……..……..……..……..……..……..……..……..……..……..…..13
6. Interrupciones……………………………………………………………………………15
7. Anexo - Instrucciones soportadas y directivas…………………………………………17

1 Introducción

El presente documento pretende explicar cómo utilizar el software ArquiSim para escribir, compilar y ejecutar código assembler 8086.

Al iniciar el programa se puede ver la vista principal:



En la parte superior podemos ver la barra de herramientas, donde inicialmente se puede crear un archivo nuevo ( ) o abrir uno ya existente ( ). Además se encuentran los clásicos botones *Guardar* ( ), *Guardar Todo* ( ), para cuando se tienen varios archivos abiertos, *Guardar Como* ( ), y otros que veremos más adelante.

La sección de color celeste corresponde al Editor (inicialmente vacío), la parte inferior corresponde a los mensajes y donde se encuentran dos pestañas: *Salida de ensamblado/ejecución* y *Puertos*. En la primera se imprimen errores de compilación, mientras que la segunda se utiliza para mostrar las escrituras hacia puertos de E/S.

Por último, sobre el lado derecho podremos ver el estado de los *registros*, *stack* y *flags* durante la ejecución de nuestro programa escrito en assembler 8086.

2 Estructura de archivo de entrada

Para poder compilar y ejecutar código se deben seguir ciertas pautas respecto a la esctura del archivo para poder mantenerlo de forma clara. Para esto existen 4 secciones, cada una con un propósito específico:

- ➔ Sección de datos (opcional): **`.data`**
  En esta sección se utilizan directivas para modificar ciertos aspectos del entorno y declarar variables.
  Las directivas soportadas son:

  - ❖ `ORG`
    Sirve para definir un desplazamiento por defecto (adicional al especificado por `DS`) que se aplicará en todas las definiciones que le siguen.
    Ejemplo:
    `ORG 100h` provocará un desplazamiento total de `DS + 100h` para todas las definiciones siguientes a dicha directiva.

  - ❖ `DB | DW | DD`
    Se utiliza para definir datos en memoria y, opcionalmente, asignarle un nombre de variable para poder referenciar dicho lugar de memoria desde el código.
    Ejemplos:
    ```
    num  DB 5
    num DW 0xFFFF
    num DD ?
    ```
    Nota: el símbolo `?` se utiliza para reservar espacio en memoria sin asignarle un valor predefinido.

  - ❖ `#DEFINE`
    Con esta directiva se puede modificar el valor inicial de cualquier registro.
    Por ejemplo:
    ```
    #define DS 300h
    ```

- ➔ Sección de código (obligatoria): **`.code`**
  Como su nombre lo indica, es la sección donde se escriben las instrucciones correspondientes al programa principal, así como también la definición de los procedimientos utilizados en el mismo. Para esto existen las directivas `PROC` y `ENDP`.
  Ejemplo:
  ```
  .code
        call prueba
        prueba proc
              mov ax, 7777h
              Ret
        prueba endp
  ```

  Por más información sobre cómo utilizar estas directivas ver el Anexo I.

- Sección de interrupciones (opcional): **`.interrupts`**
  En esta sección se definen las rutinas de atención a las distintas interrupciones que se quiere simular, utilizando las directivas `!INT` y `!ENDINT`.
  Por más información sobre cómo definir interrupciones utilizando estas directivas, ver la sección de *Interrupciones*.

- Sección de puertos (opcional): **`.ports`**
  Por último, en esta sección se definen los puertos de E/S que se desean simular con sus datos precargados. Un puerto puede ser *PS* (puerto secuencial) o *PDDV* (puerto con datos de duración variable).
  Por más información sobre cómo definir puertos, ver la sección de *Entrada/Salida*.

3 Compilación

En la barra de herramientas podemos ver el ícono . Éste corresponde a la acción de *Compilar* y realiza los siguientes puntos:

- Verifica que la sintaxis del programa sea correcta.
- Verifica que la semántica del programa sea correcta.
- Limpia la memoria, el stack y los registros (incluyendo las flags).
- Traduce las directivas de registros y datos y los escribe en sus correspondientes registros o lugares de memoria.
- Crea las instrucciones del programa principal, procedimientos e interrupciones y los ubica en sus correspondientes segmentos. Recordar que estos segmentos son ficticios, por lo que ningún tipo de código se coloca en memoria.

Notar que el proceso de compilación **no** verifica algunos puntos que son indispensables para una ejecución satisfactoria, por lo que el usuario debe asegurarse de que se cumplan:

- En los procedimientos se ejecuta en algún momento la instrucción `RET` para retornar de la llamada.
- Las interrupciones ejecutan en algún momento la instrucción `IRET` de forma de avisarle al procesador que finalizó la atención a la interrupción en cuestión y que se puede retornar a lo que se estaba ejecutando previamente.

Cuando exista un error de compilación se generará el mensaje correspondiente en la parte inferior, bajo la pestaña *Salida de ensamblado/ejecución*, indicando línea y columna donde se produjo el mismo. Este mensaje será de utilidad para comprender el motivo por el cual no se pudo realizar la compilación. Por ejemplo:

➔ Código:

```
mov [bx], 1
```

Mensaje generado:

```
Error semantico:
Falta una declaración explícita del tamaño (PTR) en el operando a memoria.
```

Corrección:

Se debe especificar un tamaño a escribir en memoria, por ejemplo:

```
mov byte ptr [bx], 1
```

➔ Código:

```
mov al, bx
```

Mensaje generado:

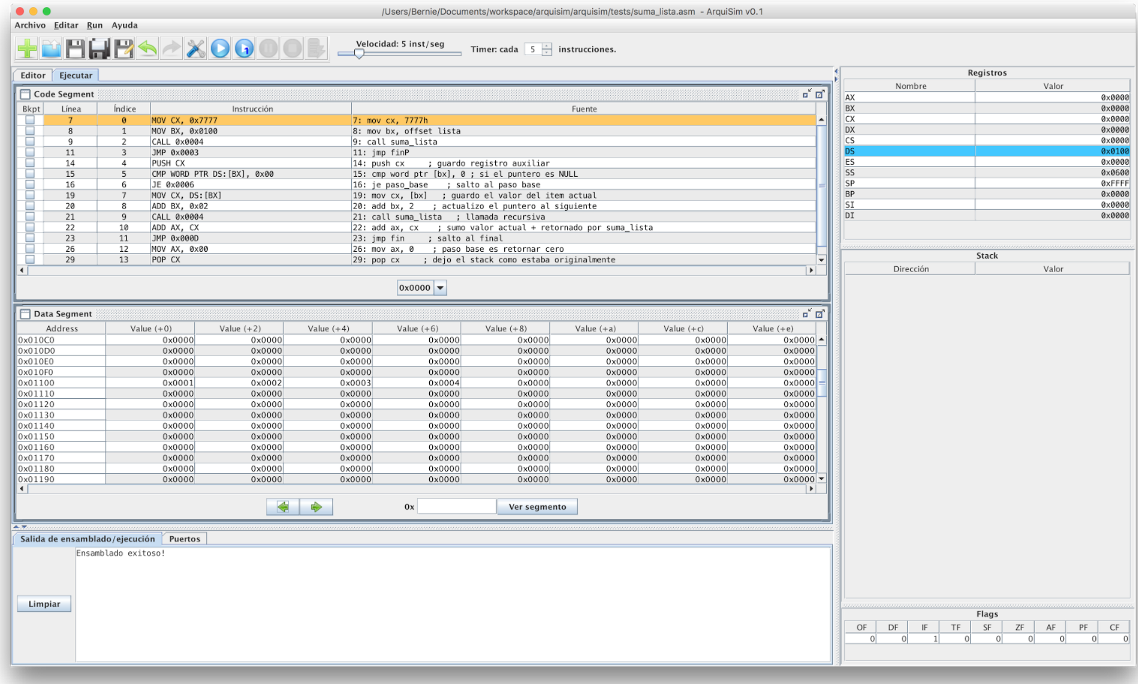
```
Error semantico:
Los operandos deben tener el mismo tamaño.
```

Corrección:

```
mov ax, bx
```

4 Ejecución

Una vez realizada la compilación del código se pasa automáticamente a la pestaña de ejecución:



Como vemos, existen 2 secciones en esta pestaña: *Code Segment* y *Data Segment*.
En la sección de código se presenta una tabla que incluye las columnas:

- _Bkpt_: activa/desactiva un breakpoint en esta línea.
- _Línea_: número de línea de la instrucción en el código fuente.
- _Índice_: hace referencia al desplazamiento respecto al CS de la instrucción.
- _Instrucción_: muestra la instrucción luego de haberla pre-procesado.
- _Fuente_: código fuente original.

Debajo de la tabla tenemos la opción de seleccionar el segmento de código a visualizar. Si no se definen interrupciones, solo se verá la opción para ver el código del programa principal, de lo contrario se podrá ver el código asociado al segmento de cada interrupción.

En la parte del centro vemos la sección de datos en memoria. Ésta muestra el segmento referenciado por el registro `DS` y se divide en 64 páginas de 1 KB. Para navegar entre las páginas se utilizan los botones y , mientras que si se quiere cambiar el segmento a visualizar se puede escribir la dirección en el campo de texto precedido por *0x* y apretar el botón *Ver segmento*.
Recordar que para acceder a memoria mediante una dirección de segmento y un offset, se realiza el cálculo `(dir_segmento*16) + offset`, por lo que el valor ingresado en el campo se multiplicará por 16 y esa será la primera dirección del segmento.
Por ejemplo, si se especifica `0x100` como dirección de segmento, la primera dirección de memoria que se visualizará será `0x1000`.

En la barra de herramientas se tiene varias utilidades para la ejecución:

	Iniciar ejecución contínua.
	Ejecutar la instrucción actual (marcada en color naranja).
	Pausar la ejecución en la instrucción actual.
	Parar la ejecución completamente.
Velocidad: 5 inst/seg	Velocidad de ejecución en instrucciones por segundo.
Timer: cada 5 instrucciones.	Cantidad de instrucciones a ejecutar luego de las cuales se generará una interrupción del reloj.

Al iniciar la ejecución del programa se podrán ver los cambios realizados a nivel de registros, stack y memoria. Con el fin de ayudar a visualizar estos cambios, se pintarán de determinados colores aquellas zonas que han tenido modificaciones:

- Al modificarse el valor de un registro, ya sea de forma explícita por parte de una instrucción o de forma implícita por el procesador, éste se pintará de color celeste.
- En cuanto al stack, al realizarse un push explícito o implícito por parte del procesador, el tope del stack se pintará de color verde claro.
- Al escribir un valor en memoria se pintará dicho lugar de color verde oscuro.

Resaltar las zonas que fueron modificadas resulta muy útil para identificar de forma rápida los cambios que se están produciendo a medida que se ejecuta el código.

Por último, en la pestaña *Puertos* de la parte inferior se mostrará la historia de escrituras en los puertos de E/S mediante la instrucción `OUT`.

5 Entrada/Salida

Para simular los puertos de E/S se tienen dos tipos: *PS* (Puerto Secuencial) y *PDDV* (Puerto con Datos de Duración Variable).

Los *PS* consisten en una lista de valores a leer en forma secuencial, es decir, al ejecutar `IN` sobre un *PS*, se lee el primer valor de la lista y se elimina de la misma. Al leer nuevamente del mismo puerto, se devolverá el siguiente valor de la lista y se eliminará, y así sucesivamente.

Por otro lado, los *PDDV* almacenan una lista de pares (valor, duración) donde para cada valor a ser leído se especifica una duración en instrucciones. Esta cantidad se puede ver como un contador y hace referencia al número de instrucciones por el cual el valor estará disponible para ser leído antes de expirar. Cuando dicho valor es el primero en la lista, su duración se reduce en 1 por cada instrucción ejecutada y el mismo expira al llegar a 0.

Todo puerto definido en el bloque **`.ports`** debe ser PS o PDDV. La forma de definir cada uno es la siguiente:

- PS:
  `número_puerto: dato1, dato2, … , datoN`

- PDDV:
  `número_puerto: (dato1, duración1), … , (datoN, duraciónN)`

donde la duración de los datos en los *PDDV* se especifica en cantidad de instrucciones y cada dato, en ambos casos, puede ser:
- Valor de 16 bits representado en decimal, octal (sufijo **q**), hexadecimal (sufijo **h** o prefijo **0x**) o binario (sufijo **b**).
- Signo de interrogación (`?`) que representa un valor aleatorio.

Por ejemplo:
```
.ports
200: 1h, 10b, 3
201:(100000000b,9), (200h,3), (?,4)
```

- El puerto número 200 es un *PS* con datos (en decimal): 1, 2 y 3.
- El puerto número 201 es un *PDDV* con datos (en decimal):
  - `256` con una duración de 9 instrucciones.
  - `512` con una duración de 3 instrucciones.
  - Dato aleatorio con una duración de 4 instrucciones.

Decimos que los datos de un puerto fueron completamente consumidos cuando no quedan más datos por leer en dicho puerto. Esto incluye el caso en que un *PDDV* no tiene más datos debido al proceso de expiración, independientemente de si fueron utilizados o no.
Se debe tener en cuenta lo siguiente:

- La escritura a un puerto predefinido que todavía tiene datos vigentes no realizará modificación alguna en dicho puerto hasta que todos los datos se hayan consumido.
- Una vez que se consumieron todos los datos de un puerto predefinido, ya sea *PS* o *PDDV*, éste se convertirá automáticamente en un *PS* conteniendo un dato fijo aleatorio. Este dato puede ser modificado escribiendo en dicho puerto mediante la instrucción `OUT`.
- Los puertos no definidos en el bloque `.ports` se consideran *PS* con dato fijo, en principio, aleatorio. El mismo puede modificarse escribiendo en dicho puerto mediante la instrucción `OUT`.

6 Interrupciones

La arquitectura 8086 soporta hasta 256 interrupciones. Para que el usuario pueda definir las rutinas de interrupción correspondientes se dispone del bloque `.interrupts` y las directivas `!INT` y `!ENDINT`.

Cada interrupción se identifica por su número y va acompañado de un valor de prioridad (a menor valor, mayor prioridad). De esta forma, en caso de generarse más de una interrupción el simulador asignará para atender a la más prioritaria.

Dentro del bloque `.interrupts` se definirá cada interrupción utilizando las directivas `!INT` y `!ENDINT` de la siguiente forma:

```
!INT número prioridad
     código
!ENDINT
```

Dado que los segmentos de código no se muestran en memoria sino que son virtuales, el compilador le asignará automáticamente un número de segmento a cada interrupción definida.

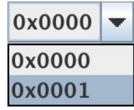
Como ejemplo, definiremos la interrupción asociada al reloj (definida en 8086 como la número 8) con prioridad 5 la cual escribirá el valor 777h en el registro AX cada vez que se atienda:

```
.interrupts
!INT 8 5
     mov ax, 7777h
     iret
!ENDINT
```

De no incluir la instrucción `IRET` nunca se retornaría de la rutina de interrupción. Si se omite, se generará un error en tiempo de compilación:

```
Error semantico:
Falta la instrucción IRET en la interrupción número 8.
```

En la sección de código de la pestaña de ejecución podremos cambiar la vista hacia el segmento de código de la interrupción mediante el *combo box* ubicado debajo de la tabla:



Como se puede ver, la rutina de interrupción del timer se ubicó en el segmento virtual 1 y seleccionando dicho segmento podremos ver el código relacionado a dicha *ISR*.

7 Errores

Código	Descripción
ERR_NOMATCH	El compilador no reconoce la secuencia encontrada (ubicada en la posición que informa el error) como una secuencia válida, esto puede significar que se ha encontrado un carácter no válido o posiblemente la secuencia no se encuentre registrada como un token válido.
ERR_NOPTR	Esto indica que para la instrucción marcada, es necesario saber de antemano cual va a ser la cantidad de bytes que se van a traer/enviar de/a memoria, se soluciona agregando una declaración explícita de su tamaño usando la directiva PTR.
ERR_NOIRET	Este error surge debido a que falta una ocurrencia de la instrucción IRET en la interrupción indicada.
ERR_UNOP	Indica que el compilador encontró un operando que no se encuentra en la lista de operandos permitidos por la instrucción (este error solo ocurre en instrucciónes de 1 solo operando), para solucionarlo se muestra una lista de los tipos de operando que la instrucción acepta como válidos.
ERR_UNSOP	Indica que el compilador encontró una combinación de operandos que no se encuentra entre los permitidos por la instrucción (este error solo ocurre en instrucciones de 2 operandos), para solucionarlo se muestran todas las posibles combinaciones de tipos de operandos aceptadas por la instrucción.
ERR_DIFSIZE	Este error ocurre cuando una instrucción tiene la restricción de que sus operandos deben tener el mismo tamaño y sus 2 operandos tienen tamaño definido pero no coinciden.
ERR_NORET	Este error surge debido a que falta una ocurrencia de la instrucción RET en el procedimiento indicado.
ERR_BADSEG	Indica que se está tratando de usar un registro de segmento incorrecto en un operando de acceso a memoria, se muestran los registros permitidos para ser utilizados.
ERR_BADBASEIND	El registro base o índice de un acceso a memoria es incorrecto, para solucionarlo se muestran las combinaciones permitidas de estos 2 registros.
ERR_NOBLOQUE	El bloque indicado por el mensaje de error no se encuentra definido en el archivo fuente, para solucionar el error se debe definir el bloque (ver anexo).
ERR_YABLOQUE	El bloque indicado se encuentra definido más de una vez en el código, para solucionar el error se debe dejar solo una definición del bloque.
ERR_NOENDP	Este error ocurre cuando el compilador llega al final del archivo pero quedan procedimientos sin cerrar, cada apertura de procedimiento

	debe estar acompañada de su cierre “ENDP” correspondiente. Para solucionar este error se debe cerrar el o los procedimientos que hayan quedado abiertos.
ERR_BADENDP	Este error ocurre cuando el compilador encuentra el cierre del procedimiento “X” cuando el último en abrirse fue el procedimiento “Y”, para solucionar este error hay que tener en cuenta que siempre se cierra el último procedimiento abierto.
ERR_ININT	Esto indica que el compilador encontró una directiva de apertura de una interrupción dentro de otra, no se pueden definir interrupciones dentro de otras por lo tanto para solucionar este error se debe sacar hacia afuera la definición de la interrupción interior.
ERR_NOID	La ubicación indicada por el error hace referencia a un identificador que no ha sido declarado en el código fuente, declarar el identificador para solucionar este error.
ERR_NOOPENPROC	Esto indica que el compilador encontró un token de cierre “ENDP” pero no hay ningun procedimiento abierto, para solucionar este error se debe eliminar este token de cierre o agregar una apertura de procedimiento si fue omitida.
ERR_BADTOKEN	Este error ocurre cuando el compilador espera encontrar un token, sin embargo recibe otro que no esperaba, en los detalles del error se incluyen los tokens esperados. Para solucionar este error se debe revisar cuidadosamente el código ya que puede deberse a un error en la declaracion de la instrucción.
ERR_BADTOKENNI	Este error ocurre cuando el compilador no reconoce el token de inicio encontrado, los posibles tokens de inicio son por ejemplo el código de una instrucción o la definición de un bloque. Para solucionar este error se debe revisar cuidadosamente el código.
ERR_TOOBIG8BITS	El compilador espera un número de 8 bits, sin embargo, el número encontrado es demasiado grande para ser representado con esa cantidad, para solucionar este error asegurarse de que el número sea representable en 8 bits.
ERR_TOOBIG16BITS	El compilador espera un número de 16 bits, sin embargo, el número encontrado es demasiado grande para ser representado con esa cantidad, para solucionar este error asegurarse de que el número sea representable en 16 bits.
ERR_TOOBIG32BITS	El compilador espera un número de 32 bits, sin embargo, el número encontrado es demasiado grande para ser representado con esa cantidad, para solucionar este error asegurarse de que el número sea representable en 32 bits.
ERR_YAID	Este error es debido a la existencia de un identificador duplicado, en los detalles del error se menciona la ubicación de ambos, para solucionarlo se debe renombrar uno de los 2 identificadores.

Anexo | Instrucciones soportadas y directivas

ADC	**ADC** *op1, op2* *Suma con carry.*	**Flags** O S Z A P C
ADC REG, { IM, REG, MEM } **ADC** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1 + op2* + C

ADD	**ADD** *op1, op2* *Suma.*	**Flags** O S Z A P C
ADD REG, { IM, REG, MEM } **ADD** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1 + op2*

AND	**AND** *op1, op2* *Operación lógica* ***and*** *bit a bit.*	**Flags** O S Z ~~A~~ P C
AND REG, { IM, REG, MEM } **AND** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1* **and** *op2* C = 0 O = 0

CALL	**CALL** *op1* *LLamada a un procedimiento.*	**Flags**
CALL { IM, IM32, REG16, MEM16, MEM32 }		Si es **near**: **PUSH** IP IP = *op1* Si es **far**: **PUSH** CS **PUSH** IP CS:IP = *op1*

CBW	**CBW** *Convierte AL a word.*	**Flags**
Sin operandos		AX = AL

CLC	**CLC** *Apaga la flag de carry.*	**Flags** C
Sin operandos		C = 0

CLI	**CLI** *Deshabilita las interrupciones.*	**Flags** I
Sin operandos		I = 0

CMP	**CMP** *op1, op2* *Compara op1 y op2 (cambia flags).*	**Flags** O S Z A P C
CMP REG, { IM, REG, MEM } **CMP** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1==op2*

DEC	**DEC** *op1* *Resta 1 de op1.*	**Flags** O S Z A P C
DEC { REG, MEM }		*op1=op1 -1*

DIV	**DIV** *op1* *Division sin signo.*	**Flags** ~~O S Z A P C~~
DIV { REG, MEM }		Si *op1* es de 8 bits: AL = AX **div** *op1* AH = AX **mod** *op1* Si *op1* es de 16 bits: AX = DX::AX **div** *op1* DX = DX::AX **mod** *op1*

IN	**IN** *op1, op2* *Lee del puerto op2 y escribe en op1.*	**Flags**
IN ACUM, { IM8, DX }		***in****(op2,op1)*

INC	**INC** *op1* *Suma 1 a op1.*	**Flags** O S Z A P C
INC { REG, MEM }		*op1=op1* + 1

INT	**INT** *op1* *Llama a la interrupción número op1.*	**Flags** I T

<table>
<tr><td colspan="2">INT IM8</td><td>PUSHF
PUSH CS
PUSH IP
I = 0
T = 0
CS:IP = MEM[op1*4]</td></tr>
</table>

<table>
<tr><td>IRET</td><td>IRET
Retorna de una interrupción.</td><td>Flags
O D I T S Z A P C</td></tr>
<tr><td colspan="2">Sin operandos</td><td>POP IP
POP CS
POPF</td></tr>
</table>

<table>
<tr><td>JA/JNBE</td><td>JA op1
Salto condicional.</td><td>Flags</td></tr>
<tr><td colspan="2">JA IM</td><td>Si C == 0 y Z == 0:
IP = IP + op1</td></tr>
</table>

<table>
<tr><td>JB/JNAE</td><td>JB op1
Salto condicional.</td><td>Flags</td></tr>
<tr><td colspan="2">JB IM</td><td>Si C == 1:
IP = IP + op1</td></tr>
</table>

<table>
<tr><td>JC</td><td>JC op1
Salto condicional.</td><td>Flags</td></tr>
<tr><td colspan="2">JC IM</td><td>Si C == 1:
IP = IP + op1</td></tr>
</table>

<table>
<tr><td>JBE/JNA</td><td>JBE op1
Salto condicional.</td><td>Flags</td></tr>
<tr><td colspan="2">JBE IM</td><td>Si C == 1 o Z == 1:
IP = IP + op1</td></tr>
</table>

<table>
<tr><td>JE/JZ</td><td>JE op1
Salto condicional.</td><td>Flags</td></tr>
<tr><td colspan="2">JE IM</td><td>Si Z == 1:</td></tr>
</table>

	IP = IP + *op1*

JG/JNLE	**JG** *op1* *Salto condicional.*	**Flags**
JG IM		Si Z == 0 y S == O: IP = IP + *op1*

JMP	**JMP** *op1* *Salto incondicional.*	**Flags**
JMP { IM, IM32, MEM16, MEM32, REG16 }		Si es **near**: IP = *op1* Si es **far**: CS:IP = *op1*

JNB/JAE	**JNB** *op1* *Salto condicional.*	**Flags**
JNB IM		Si C == O: IP = IP + *op1*

JNC	**JNC** *op1* *Salto condicional.*	**Flags**
JNC IM		Si C == O: IP = IP + *op1*

JNE/JNZ	**JNE** *op1* *Salto condicional.*	**Flags**
JNE IM		Si Z == 0: IP = IP + *op1*

JNG/JLE	**JNG** *op1* *Salto condicional.*	**Flags**
JNG IM		Si Z == 1 o S != O: IP = IP + *op1*

JNO	**JNO** *op1* *Salto condicional.*	**Flags**
JNO IM		Si O == 0: IP = IP + *op1*

JNS	**JNS** *op1* *Salto condicional.*	**Flags**
JNS IM		Si S == O: IP = IP + *op1*

JG/JNLE	**JG** *op1* *Salto condicional.*	**Flags**
JG IM		Si Z == 0 y S == O: IP = IP + *op1*

JO	**JO** *op1* *Salto condicional.*	**Flags**
JO IM		Si O == 1: IP = IP + *op1*

JS	**JS** *op1* *Salto condicional.*	**Flags**
JS IM		Si S == 1: IP = IP + *op1*

MOV	**MOV** *op1, op2* *Copia el contenido de op2 a op1.*	**Flags**
MOV REG, { IM, REG, MEM } **MOV** MEM, { IM, REG, REG_SEG} **MOV** REG_SEG_NOCS, { REG16, MEM16} **MOV** REG16, { REG16, REG_SEG} *(Los operandos deben tener el mismo tamaño)*		*op1 = op2*

MUL	**MUL** *op1* *Division sin signo.*	**Flags** O ~~S Z A P~~ C

MUL { REG, MEM }	Si *op1* es de 8 bits: AX = AL * *op1* Si *op1* es de 16 bits: DX::AX = AX * *op1*

NEG	**NEG** *op1* *Complemento a 2.*	**Flags** O S Z A P C
NEG { REG, MEM }		*op1* = **not** *op1* + 1

NOT	**NOT** *op1* *Complemento a 1.*	**Flags**
NOT { REG, MEM }		*op1* = **not** *op1*

OR	**OR** *op1, op2* *Operación lógica **or** bit a bit.*	**Flags** O S Z ~~A~~ P C
OR REG, { IM, REG, MEM } **OR** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1* **or** *op2* C = 0 O = 0

OUT	**OUT** *op1, op2* *Escribe op2 en el puerto op1.*	**Flags**
OUT IM8, ACUM **OUT** DX, ACUM		***out***(*op1,op2*)

POP	**POP** *op1* *Quita una palabra del stack.*	**Flags**
POP { REG, MEM, REG_SEG_NOCS }		*op1*=MEM[SP] SP = SP + 2

POPF	**POPF** *Quita las flags del stack.*	**Flags** O D I T S Z A P C
Sin operandos		*flags*=MEM[SP] SP = SP + 2

PUSH	**PUSH** *op1*	**Flags**

	Agrega una palabra al stack.	
PUSH { REG, MEM, REG_SEG_NOCS }		SP = SP - 2 MEM[SP]=*op1*

# PUSHF	**PUSHF** *Agrega las flags al stack.*	**Flags**
Sin operandos		SP = SP - 2 MEM[SP]=*flags*

# RET	**RET** *Retorna de un procedimiento.*	**Flags**
Sin operandos		Si es **near**: **POP** IP Si es **far**: **POP** CS **POP** IP

# ROL	**ROL** *op1, op2* *Rotación a la izquierda.*	**Flags** O C
ROL REG, { UNO, CL } **ROL** MEM, { UNO, CL } *(Los operandos deben tener el mismo tamaño)*		*op1 rota op2 lugares a la izquierda.*

# ROR	**ROR** *op1, op2* *Rotación a la derecha.*	**Flags** O C
ROR REG, { UNO, CL } **ROR** MEM, { UNO, CL } *(Los operandos deben tener el mismo tamaño)*		*op1 rota op2 lugares a la derecha.*

# SAL/SHL	**SAL** *op1, op2* *Desplazamiento a la izquierda.*	**Flags** O S Z ~~A~~ P C
SAL REG, { UNO, CL } **SAL** MEM, { UNO, CL } *(Los operandos deben tener el mismo tamaño)*		*op1 se desplaza op2 lugares a la izquierda.*

# SAR	**SAR** *op1, op2* *Desplazamiento a la derecha.*	**Flags** O S Z ~~A~~ P C

SAR REG, { UNO, CL } **SAR** MEM, { UNO, CL } *(Los operandos deben tener el mismo tamaño)*	*op1 se desplaza op2 lugares a la derecha, conserva el signo.*

SBB	**SBB** *op1, op2* *Resta con carry.*	**Flags** O S Z A P C
SBB REG, { IM, REG, MEM } **SBB** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1 - op2 -* C

SHR	**SHR** *op1, op2* *Desplazamiento a la derecha.*	**Flags** O S Z ~~A~~ P C
SHR REG, { UNO, CL } **SHR** MEM, { UNO, CL } *(Los operandos deben tener el mismo tamaño)*		*op1 se desplaza op2 lugares a la derecha.*

STC	**STC** *Enciende la flag de carry.*	**Flags** C
Sin operandos		C = 1

STI	**STI** *Habilita interrupciones.*	**Flags** I
Sin operandos		I = 1

SUB	**SUB** *op1, op2* *Resta.*	**Flags** O S Z A P C
SUB REG, { IM, REG, MEM } **SUB** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1* **-** *op2*

XOR	**XOR** *op1, op2* *Operación lógica* ***xor*** *bit a bit.*	**Flags** O S Z ~~A~~ P C
XOR REG, { IM, REG, MEM } **XOR** MEM, { IM, REG } *(Los operandos deben tener el mismo tamaño)*		*op1=op1* **xor** *op2* C = 0 O = 0

Codigo de Operandos	
IM	Inmediato de 8 o 16 bits
IM8	Inmediato de 8 bits
IM16	Inmediato de 16 bits
IM32	Inmediato de 32 bits
REG	Registro de 8 o 16 bits
REG8	Registro de 8 bits
REG16	Registro de 16 bits
REG_SEG	Registro de segmento
REG_SEG_NOCS	Registro de segmento (excluyendo el CS)
MEM	Acceso a memoria de 8 o 16 bits
MEM8	Acceso a memoria de 8 bits
MEM16	Acceso a memoria de 16 bits
MEM32	Acceso a memoria de 32 bits
ACUM	Registro acumulador AX o AL
UNO	El inmediato 1

En cuanto a las directivas, también contamos con un subconjunto de las originales y a su vez unas directivas extra. Sus funcionalidades se detallan a continuación:

DIRECTIVAS	

OFFSET	Devuelve el desplazamiento de etiquetas y variables.
	Ejemplo de uso:
	`.data`
	`lista dw 1, 2, 3`
	`...`
	
	`.code`
	`mov ax, offset lista`

SEGMENT	Devuelve el segmento de etiquetas y variables. Ejemplo de uso: `.code`

	`...`
	`loop:`
	`...`
	`mov ax, segment loop`

DUP	Usada junto con DB, DW y DD, se usa para evitar repetir datos, su sintaxis es [DB/DW/DD] DUP(cant) valor1,valor2,...,, la cadena de valores va a ser escrita en memoria “cant” veces.
DB, DW, DD	Directivas usadas para predefinir datos en memoria antes de comenzar la ejecución del programa. Su sintaxis es: `(DB|DW|DD) valor1, valor2, ..., valorN` Esto agrega a memoria los valores especificados de manera contigua.
ORG	Define el desplazamiento con respecto al segmento de datos, si le siguen directivas db,dw,dd, los datos en memoria empezaran a ser colocados a partir de donde indique el org. Ejemplo de uso: `ORG 100h`
#DEFINE	Directiva fuera del set de 8086, definida para poder setear valores en los registros de segmento antes de comenzar el programa. Ejemplo de uso: `#define ds 200h`
PROC y ENDP	Se utilizan para definir procedimientos. <*nombre*> PROC <*código*> <*nombre*> ENDP donde <nombre> es un identificador que se utilizará en el código principal para llamar a dicho procedimiento. El nombre del procedimiento es un valor alfanumérico que no puede empezar con un número y además puede contener los símbolos guión (-) y guión bajo (_).
!INT y !ENDINT	Estas directivas se utilizan para definir interrupciones. !INT número prioridad <código> !ENDINT donde: ● `número`: Número de interrupción. Valor entre 0 y 255. ● `prioridad`: Prioridad de la ISR. Valor mayor o igual a 1, siendo 1 el valor de mayor prioridad.