5 Ensamblador para 8085

5.1. Introducción

Si examinamos el contenido de la memoria de un computador, un programa aparece como una serie de dígitos hexadecimales indistinguibles unos de otros. El procesador o CPU interpreta estos dígitos como códigos de instrucción, direcciones o datos.

Sería posible escribir un programa en esta forma, pero resultaría un proceso lento y costoso. Por ejemplo, el siguiente programa se almacena en la memoria como se muestra:

```
repite:   MOV A,M          0100  7Eh
          CPI 0h           0101  FEh 00h
          JNZ fin          0104  C2h 0Dh 01h
          MVI M, FFh       0107  36h FFh
          INX H            0109  23h
          JMP repite       010A  C3h 00h 01h
fin:      HLT              010D  76h
```

- El byte 7E es interpretado por el procesador como el código de la instrucción MOV de transferencia de memoria (indicada por HL) al registro A.
- Los bytes FE 00 se corresponden con el código de la instrucción de comparación CPI, del acumulador con el dato inmediato 00h.
- Los bytes C2 0D 01 indican una instrucción de salto condicional a la dirección 010Dh.
- Los bytes 36 FF realizan la transferencia de FF a la memoria.
- El byte 23 indica que el par de registros HL debe incrementarse como si fueran un registro de 16 bits.
- Los bytes C3 00 01 indican un salto incondicional a la dirección 1000h.
- El byte 76 es la instrucción HALT o parada del procesador.

El texto de izquierda es un programa escrito en ensamblador, mientras que el de la derecha es un volcado directo de la memoria. Las diferencias entre ambos son obvias en cuanto a términos de legibilidad.

El programa anterior realiza una tarea muy sencilla. Utilizando el par de registros HL como dirección de memoria carga el contenido en el acumulador. Si el contenido es diferente de 00 termina, si no, cambia el contenido por FFh y continua por la siguiente dirección. Incluso en un programa tan simple vemos lo costoso que puede ser trabajar directamente sobre la memoria.

Sin embargo, existe un problema añadido. Supongamos que deseamos introducir una instrucción más al programa propuesto. Por ejemplo, supongamos que quiere especificar una dirección inicial de memoria en HL. Esto se puede hacer con la secuencia de dígitos hexadecimales: 21 LL HH, donde LL representa la parte baja de la dirección de memoria y HH la alta.

	LXI H,A000h	0100	**21h 00h A0h**
repite:	MOV A,M	0103	7Eh
	CPI 0h	0104	FEh 00h
	JNZ fin	0107	**C2h 10h 01h**
	MVI M, FFh	010A	36h FFh
	INX H	010C	23h
	JMP repite	010D	**C3h 03h 01h**
fin:	HLT	0110	76h

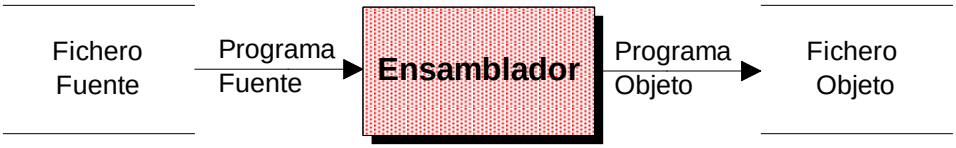
En negrita se han marcado los cambios sobre el programa original. Como vemos, en la memoria la introducción de una nueva instrucción da lugar a efectos laterales que obligan a modificar otras del programa. El riesgo de cometer un fallo es mayor.

La ilegibilidad del programa agrava el riego de error si se intentan añadirse más instrucciones y efectuar más cambios.

Para evitar esta forma engorrosa y tediosa de trabajar, sobre todo en programas de cierta complejidad, el primer paso está en utilizar el lenguaje ensamblador, que

proporciona una notación de las instrucciones completamente legible, y evita al programador tener que referirse a direcciones de memoria específicas.

El lenguaje ensamblador es, en síntesis, una secuencia de instrucciones que se convierten a un código hexadecimal ejecutable por la máquina a través de un programa llamado Ensamblador.



El Ensamblador convierte el programa fuente, escrito en lenguaje ensamblador, en su equivalente en hexadecimal, denominado programa objeto. El programa objeto es muy similar a la representación en memoria que tendrá el programa.

5.2. Sintaxis

En un programa ensamblador se distinguirá los siguientes tipos de elementos: Directivas, Instrucciones de ensamblaje e Instrucciones de la maquina.

- *DIRECTIVAS:*

Las Directivas ofrecen información al ensamblador sobre el tipo de elementos que se va a encontrar a continuación y la dirección de memoria donde debe disponerlos (si corresponde). Se caracterizan por ir precedidas por un punto.

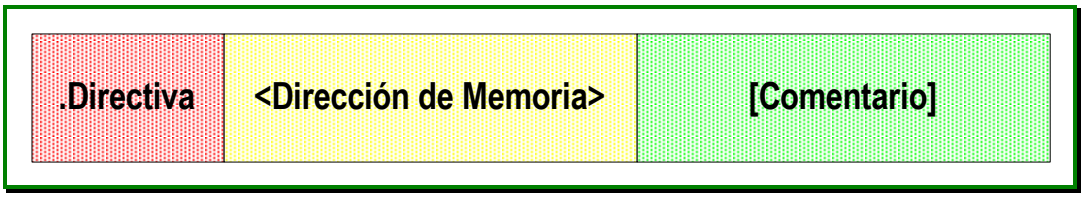


Figura 5.1. Estructura de una Directiva.

Disponemos de tres directivas distintas. Estas directivas nos permiten hacer definiciones (*define*), introducir datos (*data*) e introducir instrucciones (*org*). Cada directiva declara, por tanto, un bloque dentro del programa.

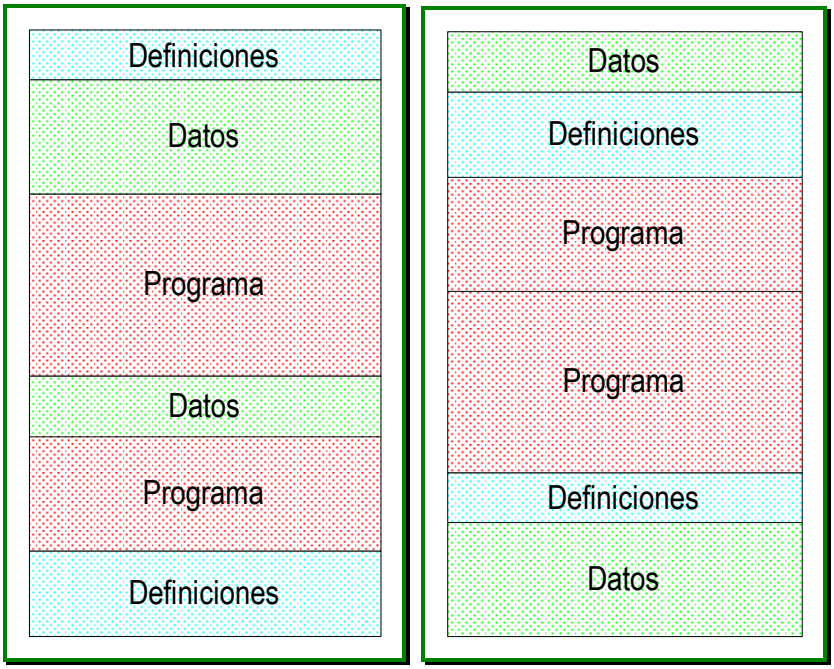


Figura 5.2. Organización Aleatoria de Bloques.

La disposición de bloques dentro del programa ensamblador no esta sujeta a ningún tipo de restricción inicial. Además, es posible declarar varios bloques de un mismo tipo.

Igualmente, ningún bloque es imprescindible. Se pueden construir programas sin declaraciones, datos e, incluso, instrucciones. Aunque esto ultimo parezca poco razonable nos puede llegar a ser útil si se quiere únicamente introducir datos en la memoria para usarlos con otro programa. En ultimo extremo podremos construir un programa vacío, que será ensamblado como un programa vacío.

Generalmente, la forma más usual de un programa será la siguiente:

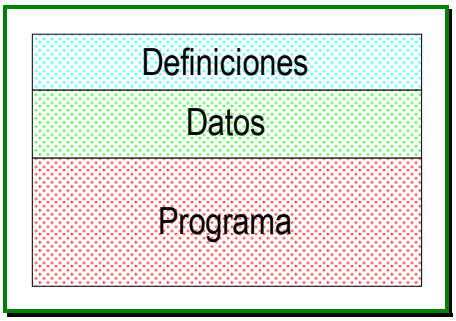


Figura 5.3. Organización Usual de Bloques.

¿La distribución usual por bloques es obligatoria?
No, puede organizar las directivas a su gusto. La distribución usual es la recomendada. La distribución representada en la figura anterior se obtiene empleando las directivas por este orden: DEFINE, DATA y por último ORG.

- *INSTRUCCIONES DE ENSAMBLAJE:*

Son un tipo de instrucciones especiales que únicamente se tienen en cuenta en el proceso de ensamblaje del programa, pero que realmente no tienen ejecución dentro de la maquina, esto es, una vez terminado aquel proceso.

Las instrucciones de ensamblaje se emplean únicamente en el proceso de ensamblaje. De esta forma, no tienen una aparición explicita dentro del código objeto, sin embargo si aparecen de forma implícita.

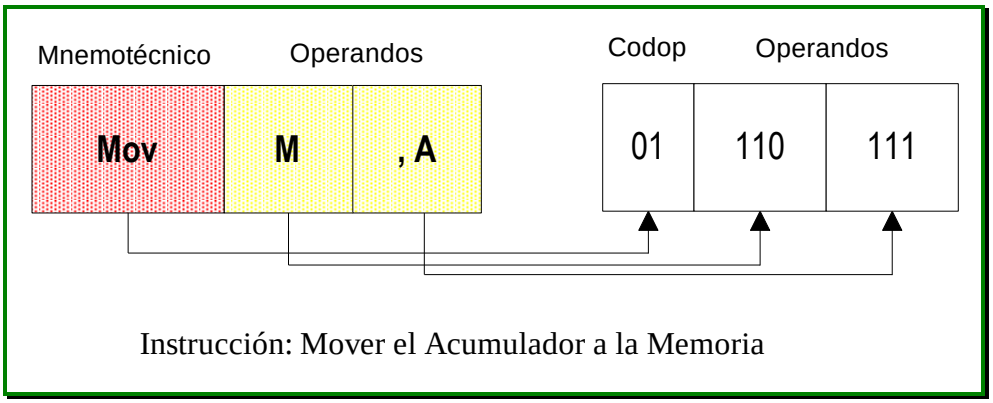
Las instrucciones de ensamblaje difieren según el bloque, o directiva previamente declarada, en la que estemos. Por ello, veremos cada una de ellas dentro de su ámbito correspondiente. Por ahora solo diremos que hay de dos clases, una para hacer definiciones y otra para declarar datos.

- *INSTRUCCIONES DE LA MAQUINA:*

A diferencia de las anteriores, las instrucciones de la maquina, o simplemente instrucciones, "corren" o se ejecutan en la maquina. Cada instrucción tiene su correspondiente operación en la maquina.

Las instrucciones se introducen tras la directiva "*.org*", o lo que es lo mismo, cuando se declara un bloque de programa. Más adelante veremos con más detalle la sintaxis dentro de este bloque, aquí solo haremos una pequeña introducción.

Básicamente, el ensamblador realiza una traducción entre un nemotécnico con unos operadores a un número. Cada nemotécnico representa una operación y se denominan así porque este permite acordarnos fácilmente de lo que realiza la operación correspondiente:



Instrucción: Mover el Acumulador a la Memoria

En este caso el ensamblador convierte de forma automática la expresión: "`mov M, A`" a `78h`.

Realmente, el ensamblador permite complicar todo esto mucho más facilitando el trabajo al programador. Como ya dijimos, esto lo veremos más adelante.

5.2.1. Sintaxis de las Definiciones

El objetivo de una definición es, simplemente, poner un nombre a un número. Durante el ensamblaje, cada vez que nos encontremos con un operando que contiene el nombre este será sustituido de forma automática por el valor correspondiente con el que se ha definido.

Más adelante, si es necesario alterar este número por alguna razón, únicamente se tendrá que cambiar la declaración, lo que nos evita la tarea de buscar todas las apariciones del número en el texto, comprobar si se trata realmente del número que queremos alterar y modificarlo.

Las definiciones solo se pueden realizar si previamente se ha declarado la directiva de definiciones "`.define`". Si no es así se producirá un error, ya que el ensamblador no entenderá la secuencia de léxicos.

La estructura de un bloque de definiciones es la siguiente:

```
. define
      [definición]
      [definición]
      [definición]
      ....
```

¿Existe alguna restricción sobre la tabulación y uso de espacios en la directiva y resto de definiciones?

No, las tabulaciones mostradas anteriormente solo son indicativas. No tienen porque realizarse de forma estricta. La palabra 'define' puede estar junto al punto sin separarse por espacios y las definiciones al principio de línea.

Cada definición tiene la siguiente estructura sintáctica:

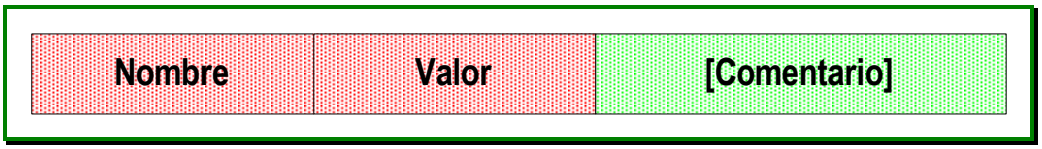


Figura 5.4. Estructura de una Definición.

donde "*nombre*" declara el nombre, mientras que en "*valor*" se asigna el valor correspondiente. *Nombre* y *Valor* deben estar separados por al menos un espacio en blanco. Por ejemplo:

```
. define
      maximo FFFFh
      minimo 0
      umbral 0CA1h
      euro 166
```

Los identificadores `maximo`, `minimo`, `umbral` y `euro` son nombres, mientras que `FFFFh`, `0`, `0CA1h` y `166` son valores que se asignan respectivamente a los nombres.

Los identificadores de nombre no pueden ser números, es decir, deben comenzar al menos por una letra. Como máximo, deben tener un tamaño no superior a 255 caracteres. Igualmente, un nombre ya usado en una definición no puede volver a emplearse nuevamente en ninguna otra definición. En este caso el ensamblador muestra un mensaje de advertencia y asigna el nombre al valor correspondiente de la primera definición.

Aunque es un caso poco probable, puesto que si seguimos la organización usual por bloques no es posible, un nombre de definición no puede ser un nombre ya usado en una etiqueta declarada con anterioridad. En el apartado "***Sintaxis de las Instrucciones***" cuando se explican las etiquetas se hace referencia a esta limitación. También puede encontrar un ejemplo de este error en el capitulo "**Mensajes producidos por el Ensamblador**".

Cada definición es una instrucción de ensamblaje. Como vemos no hay una instrucción maquina análoga a ella, de forma que no queda expresada de forma explicita. Sin embargo, la instrucción es tomada por el ensamblador que reemplaza cada nombre por el valor correspondiente.

¿Es una definición sensible a la capitalización?
A diferencia del resto de elementos del ensamblador, el identificador de una definición es sensible a minúsculas y mayúsculas. El objetivo de esto es obtener una mayor diversidad de nombres.

5.2.2. Sintaxis de los Datos

Dentro de un programa se establece una clara diferencia entre datos e instrucciones. Los datos constituyen básicamente los operandos de las instrucciones.

La distribución en memoria de datos e instrucciones suele estar separada y claramente diferenciada. Esto se debe a que tanto datos como instrucciones son tratados de la misma forma por la maquina según la situación en la que este, es decir, un dato no es un dato, ni una instrucción una instrucción, por sí sola. Depende de como sea tratada por la maquina.

Los datos pueden ser de tres tipos. Se han distinguido tres tipos distintos de datos simples con los que trabajar en el ensamblador. La diferencia fundamental entre estos tipos es el tamaño, como muestra la Tabla 5.1.

Tabla 5.1. Tamaño de los datos.

Tipo	Tamaño	Ejemplos
Byte	8 bits	1, 10, 255, 127, -127, ...
Word	16 bits	FE00h, 65535, FFFFh, ...
String	8 bits x nº caracteres	"Esto es un ejemplo", "Salida del programa", ...

Generalmente la mayoría de los datos serán de tipo *Byte*, puesto que el 8085 solo opera con datos de 8 bits. Los datos de tipo *Word* se emplearan con mayor probabilidad para definir direcciones de memoria, ya que el 8085 es capaz de direccionar direcciones de hasta 16 bits. Las cadenas de caracteres solo proporcionan una mayor legibilidad al programa ensamblador puesto que se pueden definir alternativamente mediante una secuencia de *Bytes*:

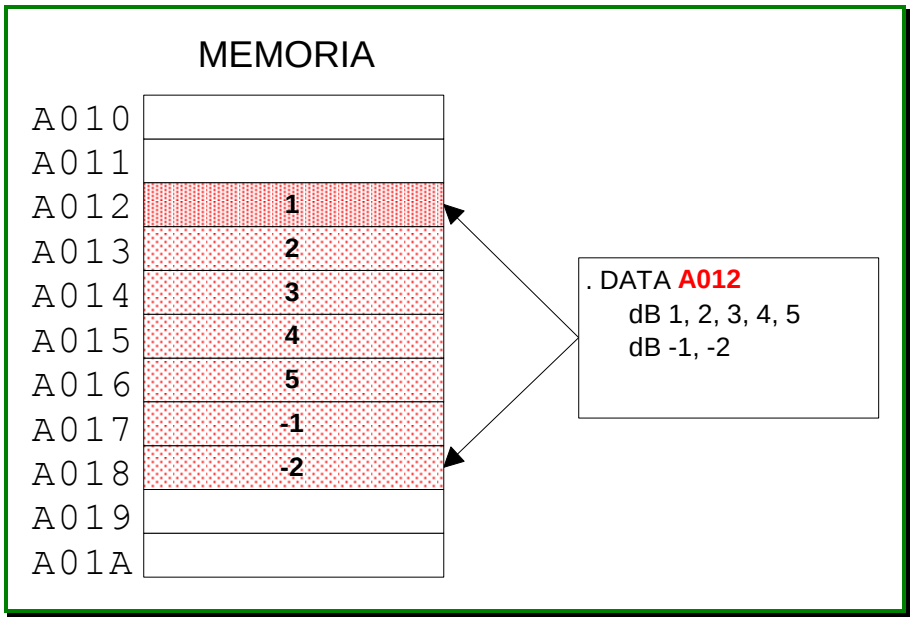
```
"Esto es un ejemplo"
45h 73h 74h 6Fh 20h 65h 73h 20h 75h 6Eh 20h 65h 6Ah 65h 6Dh 70h 6Ch 6Fh
```

En la primera línea se muestra la cadena original, mientras que la segunda la cadena de bytes correspondiente. Como vemos, es mucho más sencillo de construir y modificar la notación para cadenas de caracteres.

La estructura de un bloque de datos es la siguiente:

```
. data <dirección origen>
     [declaración datos]
     [declaración datos]
     [declaración datos]
     ....
```

En dirección origen se indica la dirección de memoria en la que se comenzara a introducir los datos.



Cada declaración de datos tiene la siguiente sintaxis:

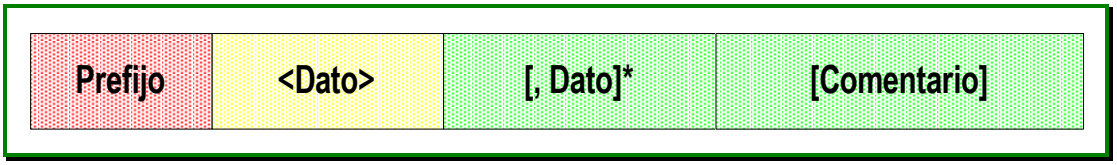


Figura 5.5. Estructura de una Declaración.

donde:

- El *prefijo* indica el tipo de dato que vamos a introducir. Los posibles prefijos coinciden con los tipos descritos anteriormente:
  - *dB*: declaración de Bytes o números de 8 bits.
  - *dW*: declaración de Words o números de 16 bits.
  - *dS*: declaración de cadenas de caracteres.

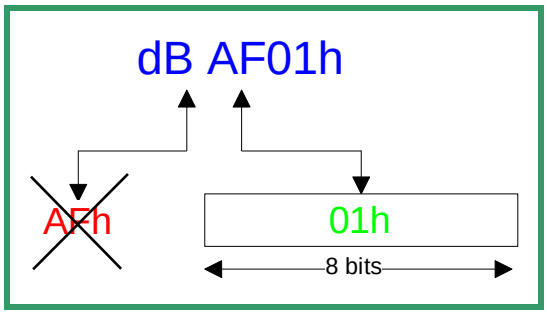
  Los prefijos no son sensibles a la capitalización de letras. Por ejemplo, ‘db’, ‘dB’, ‘DB’ y ‘Db’ se reconocen igualmente por un prefijo de declaración de Bytes.

- Los *datos* simplemente son una secuencia de números o expresiones lógicas y aritméticas separados por comas o una cadena de caracteres.

Cuando se realiza una declaración de un dato (ya sea explícitamente o como resultado de una expresión aritmética y/o lógica) como un byte, este se almacena en el próximo byte útil de memoria.

Cuando la declaración es de un word **los 8 bits menos significativos de la expresión se almacenan en el próximo byte útil de memoria, mientras que los ocho bits más significativos se almacenan en el siguiente**. Como se ve, la dirección está colocada en la memoria en orden inverso. Normalmente, las direcciones se encuentran en la memoria en esta forma. Esta operación se utiliza básicamente para crear una tabla de direcciones constantes.

Si la representación del dato excede el número de bits con que ha sido declarado entonces se informa al programador mediante una advertencia de ensamblaje. **Únicamente quedaran almacenados los bits menos significativos.**



A continuación mostramos un ejemplo de declaración de datos:

```
.DATA 100h
   dB      1, 2 , 3, 4, 5, 6, 7, 8, 9, 0    ;cifras
   dB      32, 25, FFh, 32, -5, -120
   dW      0, FFFFh, 01A0h              ;mis posiciones de memoria
   dS      "Pulse una tecla"            ;mensaje para continuar
   dS      "Fin del programa"           ;mensaje de salida
```

Además, es posible emplear etiquetas durante una definición, esto nos permite poder referenciar más adelante en el programa un determinado dato. El uso de etiquetas durante una definición pone las bases de lo que, en lenguajes de más alto nivel, serán las variables.

Finalmente la estructura de una declaración de datos es:

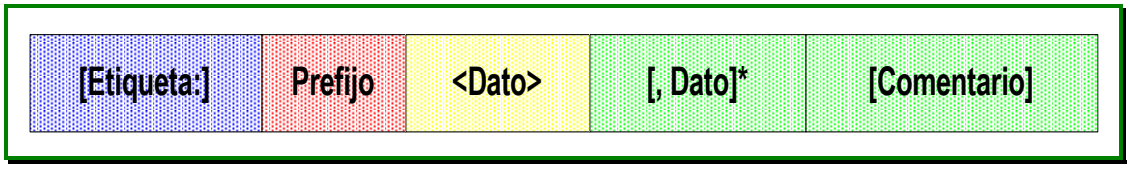


Figura 5.6. Estructura Extendida de una Declaración.

En el apartado 5.2.3 veremos con más detenimiento las etiquetas y como se emplean estas en las instrucciones. De esta forma veremos la utilidad real de la definición de etiquetas durante una declaración de datos.

5.2.3. Sintaxis de las Instrucciones

La estructura de un bloque de instrucciones es la siguiente:

```
. org <dirección origen>
      [instrucción]
      [instrucción]
      [instrucción]
      ....
```

Las instrucciones del lenguaje ensamblador vienen dadas por una serie de reglas que conforman la sintaxis de este. Dentro de cada instrucción hay cuatro partes o campos separados:

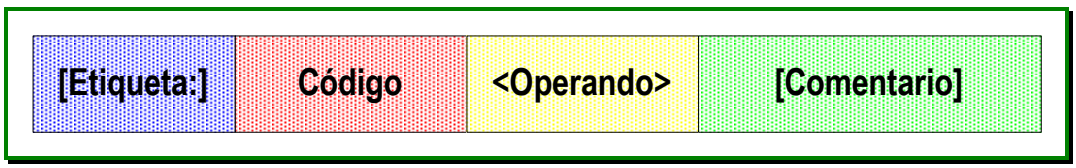


Figura 5.7. Estructura de una Instrucción.

Los campos Etiqueta y Comentario son prescindibles y no tienen porqué estar necesariamente en cada instrucción ensamblador. El Código de Operación, también conocido por *Codop*, es completamente necesario si queremos definir una instrucción. Los operadores dependerán del Código de Operación correspondiente, ya que hay instrucciones que no requieren ningún operador mientras que otras necesitan varios de ellos.

Estos campos están separados por espacios en blanco, no existe ninguna restricción sobre el número de estos que debe haber entre cada dos campos, siempre que al menos halla uno.

ETIQUETA

Es un campo de utilización opcional que, cuando está presente, puede tener una longitud de 1 a 255 caracteres. El primer carácter de la etiqueta debe ser una letra del alfabeto. De no ser así, como veremos más adelante podría ser confundido con una dirección real de memoria. Estos caracteres deben ir seguidos de dos puntos (:). Los códigos de instrucción están especialmente definidos por el ensamblador y no pueden ser utilizados como etiquetas. Nuestro ensamblador es capaz de diferenciar entre ellos, por lo que esta restricción no está presente, sin embargo se recomienda no mezclar etiquetas de salto con códigos de instrucción.

El objetivo de una etiqueta es referenciar una dirección de memoria. Esto es, cuando una etiqueta se sitúa en la declaración de un dato o delante una instrucción de programa esta haciendo referencia a la dirección de memoria en la que se encuentra dicho dato o instrucción.

Gracias a las etiquetas el programador de ensamblador no tiene que calcular a mano la dirección de un determinado dato que necesita ni la de una instrucción a la que se debe saltar. De esta manera se puede simplificar enormemente la tarea del programador con respecto a los saltos y a la carga o almacenamiento de datos.

Si tenemos en cuenta el significado de una etiqueta es muy fácil reconocer las limitaciones que el uso de estas conlleva. Por un lado, una etiqueta define solamente una dirección. No puede repetirse. Por tanto, esto no es posible:

```
Salto:    mov a, b
          ...
Salto:    call sub
          ...
          jmp Salto
```

Es obvio que el ensamblador no puede determinar qué dirección es a la que debe ir la instrucción JMP.

No obstante, la situación contraria es posible (aunque poco útil), esto es, asignar la misma dirección a dos etiquetas. La siguiente secuencia de instrucciones es válida:

```
Salto:

Salto2:   mov a, b
          ....
          jmp Salto
```

Ahora podemos justificar porque una etiqueta debe al menos comenzar por una letra. De no ser así, la etiqueta podría ser una secuencia completa de dígitos. Si esto ocurre el ensamblador es incapaz de distinguir una etiqueta de un número igual. Esta ambigüedad daría lugar a fallos.

Tabla 5.2. Ejemplo de etiquetas.

Dirección	Instrucción
0100	Mov a, b
0101	0100: Mov c, d
0102	Jmp 0100

Como podemos ver existe una clara ambigüedad sobre lo que realmente se quiere codificar. Por un lado se podría interpretar como que el salto de la dirección `0102` es hacia donde indica la etiqueta `0100` (dirección `0101`). Pero por otro lado también es posible decir que realmente se quiere saltar a la dirección `0100`.

Tampoco es posible que una etiqueta tenga el mismo nombre que una definición previa. Como en el caso anterior, el ensamblador no es capaz de distinguir a que elemento, si etiqueta o definición, esta haciendo referencia el nombre encontrado.

Por ultimo, al igual que los nombres de definición, las etiquetas son sensibles a mayúsculas y minúsculas.

Resumiendo :

- Las etiquetas son sensibles a mayúsculas y minúsculas.
- Tienen que empezar al menos por una letra.
- Tienen que ser menores de 255 caracteres.
- No pueden coincidir con definiciones anteriores.
- No deben coincidir con etiquetas anteriores.

CODIGO DE OPERACIÓN

Es el campo más importante de la instrucción ya que en el se define la operación a realizar por la máquina (suma, resta, salto, etc.). Cada una de las instrucciones tiene un código determinado, que es privativo de la misma, y que debe aparecer en el campo del código. Por ejemplo, las letras "`JMP`" definen exclusivamente la operación de "salto incondicional" y ninguna otra instrucción. No existen dos instrucciones con el mismo código ni dos códigos para una misma instrucción.

Después de las letras del campo del código, debe haber como mínimo un espacio en blanco:

`Jmp Salto`	Salto incondicional a Salto (etiqueta)
`Adi 7Bh`	Sumar 7Bh con acumulador
`Mov a, b`	Transferir registro B al acumulador

A cada secuencia de letras que definen exclusivamente una operación se denomina nemotécnico. Los nemotécnicos a diferencia que las etiquetas y los identificadores de definiciones no son sensibles a las mayúsculas ni minúsculas. Esto es "`Jmp`", "`JMP`", "`jmp`" o "`JmP`" se refieren al mismo nemotécnico.

OPERANDO

La información contenida en este campo se usa conjuntamente con el campo del código a fin de definir con precisión la operación a ejecutar por la instrucción. Según el contenido del campo del código, el campo del operando puede no existir, o consistir en una cifra o palabra, o bien en dos, separados ambas por una coma.

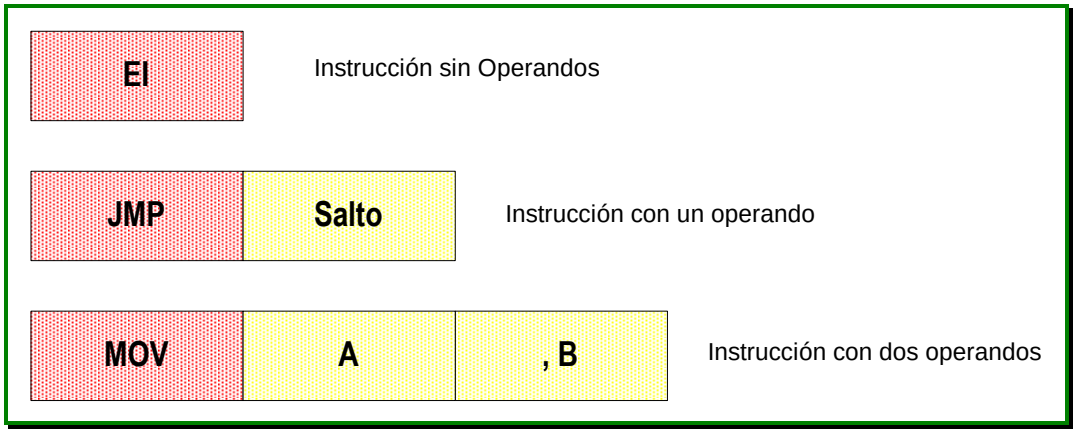


Figura 5.8. Tipos de Instrucciones.

Hay cuatro tipos de información válidos como campo del operando:

- **Registro**. Un registro (o código indicativo de una referencia a memoria) definido como fuente o destino de datos en una operación. Un número puede usarse para especificar el registro o referencia a memoria mencionados, pudiéndose obtener dicho número a partir de una expresión, pero el número finalmente evaluado debe estar entre 0 y 7. La correspondencia se muestra en la tabla 5.3.

Tabla 5.3. Números de registros.

Valor	Registro
0	B
1	C
2	D
3	E

Valor	Registro
4	H
5	L
6	Referencia a memoria
7	Acumulador (A)

Por ejemplo, la instrucción MVI permite la carga inmediata del segundo operador en el registro indicado por el primer operador. Esto es, usos típicos de esta instrucción serían:

`MVI A, 1`	Transferir 1 al acumulador
`MVI H, dir_alta`	Transferir dir_alta (definición) al registro H
`MVI L, dir_baja`	Transferir dir_baja (definición) al registro L

Cada uno de los registros se pueden sustituir de diversas formas:

`MVI 7, 1`	7 es el acumulador
`MVI 8/2, dir_alta`	8/2 es 4 que se corresponde con H
`MVI regL, dir_baja`	RegL es una definición con valor 5 (L)

- **Par de registros**. Un par de registros utilizado como fuente o destino de una operación con datos. Los pares de registros se especifican como se ve en la tabla 5.4.

Tabla 5.4. Pares de registros.

Especificación	Par de Registros
B	Registros B y C
D	Registros D y E
H	Registros H y L
PSW	Un byte que indica el estado de los bits de condición y el registro A.

SP	**El registro de 16 bits del puntero de stack.**

Por ejemplo:

`INX B`	Incrementar BC (como un registro de 16 bits)
`PUSH PSW`	Guardar en pila los bits de estado y el registro A
`LXI D,0h`	Cargar HL con 0h

- **Dato inmediato**. Un dato expresado de forma inmediata en la instrucción. El dato puede ser resultado de una expresión. Por ejemplo:

`ADI 5+5`	Sumar 10 al contenido del acumulador
`ORI 0Fh`	OR del acumulador con 0Fh
`LXI D,10h+(MOV A,B)`	Cargar HL con 10h+78h

- **Dirección de 16 bits.** Una dirección de memoria de 16 bits o la etiqueta de una instrucción en memoria. Por ejemplo:

`JMP 34F0h`	Saltar a la dirección 34F0
`LXI H,A000h`	Cargar HL con A000h

COMENTARIOS

La única regla que rige la utilización de los comentarios es que éstos deben ir precedidos de un punto y coma (;).

```
AQUI:     MVI  C, DADH   ;esto es un comentario.
```

El campo del comentario puede aparecer sólo en una línea, a pesar de que no exista instrucción en la misma:

```
;comentario sin instrucción
```

5.2.4. Sintaxis del Operando

Existen nueve formas de especificar el operando. Estas nuevas formas se detallan a continuación:

1. **Dato Hexadecimal**. Todo número hexadecimal esta expresado en base 16 para cuya representación emplea los dígitos del 0 al 9 y las 6 primeras letras del alfabeto (A, B, C, D, E y F). Son los más comunes cuando se trabaja en ensamblador. Los datos hexadecimales van seguidos de una letra `H`. Por ejemplo:

```
A2h (162), 10AFh (4271), FFFFh (65535), 11Ah (282)
```

2. **Dato Decimal**. Expresados en base 10, son los números que todos conocemos con dígitos entre 0 y 9. Pueden ir seguidos de la letra `D`, o bien ir solos. Por ejemplo:

```
0, 1, 255d, 35, 1050d
```

3. **Dato Octal**. Expresados en base 8, utilizan solo los dígitos del 0 al 7. Deben ir seguidos de la letra `O` ó `Q` para que sean reconocidos como tales. Por ejemplo:

```
777q (511), 123o (83), 242q (162), 22o (18)
```

4. **Dato Binario**. Números binarios expresados en base 2 (0,1). Van seguidos de la letra `B`. Se emplean generalmente para definir mascaras de bits. Por ejemplo:

```
0101011101B (349), 11110000B (240)
```

5. **El contenido actual del contador de programa**. Este se define con el carácter $ y equivale a la dirección de la instrucción en ejecución.

6. **Una constante ASCII**. Son uno o más caracteres ASCII encerrados entre comillas simples.

```
'a' (65), 'b' (66), '*'(42),'?' (63)
```

7. **Una instrucción encerrada entre paréntesis.** Una instrucción entre paréntesis puede emplearse como operando. El valor concreto del operando será el código hexadecimal con el que la instrucción es codificada.

```
(MOV A,B) (65), (INX H) (35), (PUSH PSW) (245)
```

8. **Identificadores de Etiquetas**. Ya sea que se les ha asignado un valor numérico por el propio ensamblador, como vimos en el caso de los registros representados por números. O bien etiquetas definidas por el programador que aparecen en el campo de la etiqueta de otra instrucción o declaración de dato.

9. **Expresiones Lógicas y Aritméticas**. Todos los operandos descritos anteriormente son expresiones. Las expresiones lógicas y aritméticas son expresiones unidas mediante los operadores: + (suma), - (resta o cambio de signo), * (multiplicación), / (división), MOD (módulo), los operadores lógicos NOT, AND, OR, XOR, SHR (rotación hacia la derecha), SHL (rotación hacia la izquierda), y paréntesis a derecha e izquierda.

   Todos los operadores tratan sus argumentos como cantidades de 16 bits y genera como resultados cantidades de 16 bits. Cada operador realiza la siguiente operación:

   - El operador **+** genera la suma aritmética de sus operandos.

     A01Bh + 00FFh = A11H

   - El operador **-** genera la resta aritmética de sus operandos, cuando se usa como sustracción (operador binario), o como aritmética negativo cuando se utiliza como cambio de signo (operador unario).

     A01Bh - 00FFh = 9F1Ch

   - El operador ***** indica el producto aritmético de los dos operandos.

A01Bh * **00FFh** = **9F7AE5h**

- El operador **/** calcula el cociente entero entre los dos operandos, el resto de la división se descarta.

A01Bh / **00FFh** = **A0h**

- El operador **MOD** calcula el resto de la división entre los dos operandos descartando el cociente.

A01Bh MOD **00FFh** = **BBh**

- El operador **NOT** realiza el complemento de cada bit del operando.

NOT **0110110b** = **1001001b**

- El operador **AND** (o **&**) realiza la operación lógica AND bit a bit entre los operandos.

011101b AND **111000b** = **011000b**

- El operador **OR** (o **|**) realiza la operación lógica OR bit a bit entre los operandos.

011101b OR **111000b** = **111101b**

- El operador **XOR** (o **^**) realiza la operación lógica O-EXCLUSIVO bit a bit entre los operandos.

011101b XOR **111000b** = **100101b**

- Los operadores **SHR** y **SHL** realizan un desplazamiento del primer operando a derecha e izquierda respectivamente el número de posiciones definidas por el segundo operando, introduciendo ceros en las nuevas posiciones.

011101b **SHR** **3** = **000011b**

011101b **SHL** **3** = **101000b**

El programador debe asegurarse de que el resultado generado por una de estas operaciones cumple los requisitos necesarios. En caso de que así no fuera los bits más significativos que no pudieran almacenarse se perderían. Por ejemplo:

`MVI A, NOT 0`	NOT 0 es FFFFh, MVI espera un dato inmediato de 8 bits.
`MVI A, -1`	-1 es FFFFh en complemento a dos.
`MVI A, -1&(FFh)`	Forma correcta de especificación.

Múltiples operadores pueden estar presentes en una expresión. Esta claro que el orden en que estos se apliquen va a dar lugar a diferentes resultados. Por esta razón las expresiones producidas por los operadores deben ser evaluadas en el orden de prioridad mostrado en la tabla 5.5.

Tabla 5.5. Operadores de una expresión.

Prioridad	
1	**Expresiones entre paréntesis**
2	**Multiplicación (*), División (/), MOD, SHL, SHR**
3	**Suma (+), Resta (-)**
4	**NOT**
5	**AND**
6	**OR, XOR**

En el caso de las expresiones entre paréntesis, el contenido entre los mismos debe evaluarse primero.

5.3. Especificación Formal de la Sintaxis del Ensamblador

La estructura del lenguaje ensamblador viene dada por una serie de reglas que conforman la sintaxis de este. El lenguaje ensamblador es el primer escalón dentro de los niveles de abstracción de la maquina, es decir, está muy cerca de la propia maquina, por lo que su complejidad como lenguaje es significativamente menor que cualquier otro lenguaje de más alto nivel.

La sintaxis de un lenguaje de programación se especifica mediante la gramática independiente del contexto que lo genera. Una gramática independiente del contexto es una cuádrupla **(N, T P, S)** en la que:

a) **N** es el conjunto de símbolos no terminales del lenguaje,
b) **T** es el conjunto de símbolos terminales del lenguaje,
c) **P** es el conjunto de reglas de producción, y
d) **S** es el axioma o símbolo inicial de la gramática,

y además cumple que todas las reglas de producción adoptan la forma:

$$A \to \alpha \text{, donde } A \in N, \alpha \in (N \cup T)^*$$

En otras palabras, es una gramática de tipo 2.

Notación empleada

Para especificar formalmente la sintaxis de los lenguajes de programación se suelen usar BNF. BNF es un metalenguaje que se usa para especificar la sintaxis de los lenguajes de programación. Esto se consigue especificando en BNF la gramática que genera el correspondiente lenguaje.

BNF, Backus Naur Form, establecido por John Backus y Peter Naur (Junio 1.959, 2 de enero 1.960). La equivalencia entre las gramáticas independientes del contexto y BNF, en el sentido de tener la misma potencia expresiva, fue demostrada por S. Ginsburg, y H.G. Pice en 1.962

En BNF la gramática de un lenguaje de programación se expresa por medio de sus reglas de producción, por lo que la sintaxis con la que se escriban estas reglas deberá poner claramente de manifiesto cuales son:

- el símbolo inicial,
- los símbolos no terminales y
- los símbolos terminales de la gramática del lenguaje

Para ello el símbolo inicial de la gramática deberá aparecer en la parte izquierda de la primera regla de producción Y el resto de símbolos no terminales deberán aparecer en la parte izquierda de al menos una regla de producción. Se dice que esa regla define al símbolo no terminal.

A partir del concepto de gramática independiente del contexto, es obvio que ningún símbolo terminal podrá aparecer en la parte izquierda de ninguna regla de producción.

El orden natural que se suele seguir al escribir las reglas de producción es, dada una regla, escribir a continuación de ella las reglas correspondientes a los símbolos no terminales que aparezcan en su parte derecha. Si alguno o algunos, de estos símbolos son comunes a varias reglas la regla que define este símbolo se pondrá al final de todas ellas. Habitualmente las reglas de producción en cuya parte derecha solo aparecen símbolos terminales figuran al final de la especificación.

Para especificar las reglas de producción, la notación BNF consta de los siguientes metasímbolos:

- **< >** que se usan como delimitadores de los símbolos no terminales de la gramática al escribir las regias de producción.
- **::=** que se utiliza para separar las partes izquierda y, derecha de las reglas de producción, Y se lee "*se define como*".
- **|** que se use como separador de las diversas alternativas que puedan aparecer en la parte derecha de una regla de producciones y se lee "*o*".
- **[ ]** para representar que lo encerrado entre los corchetes es opcional.

- **{ }** para representar que lo encerrado entre los corchetes se repite 0 o varias veces.

5.3.1. Notación BNF del Ensamblador

<Programa> ::= *{<Bloque>}*

<Bloque> ::= *<Bloque definición>* | *<Bloque declaración>* | *<bloque Programa>*

<Bloque definición> ::= *<directiva define> {<definiciones>}*

<Bloque declaración> ::= *<directiva datos> {<declaraciones>}*

<Bloque Programa> ::= *<directiva programa> {<instrucciones>}*

<directiva define> ::= *<punto>* **DEFINE** *[***;** *<comentarios>]*

<directiva datos> ::= *<punto>* **DATA** *<expresión> [***;** *<comentarios>]*

<directiva programa> ::= *<punto>* **ORG** *<expresión> [***;** *<comentarios>]*

<definición> ::= *<identificador> <expresión> [***;** *<comentarios>]*

<declaración> ::= *[<identificador> :] <prefijo> <expresión>* **,** *{<expresión>} [***;** *<comentarios>]*

<instrucciones> ::= *<código operación> [<expresión reg>] [* **,** *<expresión reg>] [***;** *<comentarios>]*

<expresión reg> ::= *<expresión>* | *<registro>*

<expresión> ::= *<termino> { <operador débil> <termino> }*

<término> ::= <*elemento*> *{* <*operador fuerte*> <*elemento*> *}*

<elemento> ::= < *identificador*> | <*constante*> | **(** <*expresión*> **)**

<identificador> ::= <*letra*> | <*identificador*> <*letra*> | <*identificador*> <*digito*>

<constante> ::= <*constante hex*> | <*constante decimal*> | <*constante octal*> | <*constante bin*> | <*constante carácter*>

<constante hex> ::= <*digito hex*> *{*<*digito hex*>*}* **H**

<constante dec> ::= <*digito*> *{*<*digito*>*}* *[***D***]*

<constante octal> ::= <*digito oct*> *{*<*digito oct*>*}* **O** | <*digito oct*> *{*<*digito oct*>*}* **Q**

<constante bin> ::= <*digito bin*> *{*<*digito bin*>*}* **B**

<constante carácter> ::= **‘** <*letra*> | <*digito dec*> **’**

<código de operación> ::= **ANA** | **AND** | **ACI** | **ADI** | .... | **XCHG** | **XTHL**

<prefijo> ::= **dB** | **dW** | **dS**

<registro> ::= **A** | **B** | **C** | **D** | **E** | **F** | **H** | **S** | **PSW**

<operador debil> ::= **+** | **-** | **AND** | **NOT** | **OR**

<operador fuerte> ::= ***** | ***/***

<digito> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<digito hex> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F**

<digito oct> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

<digito bin> ::= **0** | **1**

<letra> ::= **a** | **b** | **c** | ... | **z** | **A** | **B** | **C** | ... | **Z**

<punto> ::= **.**

5.4. Mensajes de Error producidos por el Ensamblador

5.4.1. Introducción

Los mensajes de error se generan cuando existe algún problema en el código fuente a ensamblar. Estos errores provocan que si el ensamblador siguiera convirtiendo el fichero en ensamblador, el programa resultante fuera totalmente incorrecto por lo que el proceso de ensamblaje se termina cada vez que se produce un error.

También es posible que el ensamblador genere mensajes de advertencia. Estos mensajes de advertencia indican incoherencias o ambigüedades en el código fuente. Estas ambigüedades no generan un programa totalmente incorrecto por lo que el proceso de compilación continua. Aunque funcione este conservara ciertos errores que deberían eliminarse. Más adelante veremos estos casos.

Los mensajes de error se muestran en rojo, mientras que los mensajes de advertencia se muestran en amarillo.

A continuación describiremos de forma detallada los mensajes de error y de advertencia de los que informa el ensamblador. Además mostraremos ejemplos en los que estos errores se producen.

5.4.2. Mensajes de Error

Errores en Directivas

Las Directivas ofrecen información al ensamblador sobre el tipo de elementos que se va a encontrar a continuación y la dirección de memoria donde debe disponerlos (si corresponde). Se caracterizan por ir precedidas por un punto.

	Error:	cada bloque debe comenzar por una directiva. Línea *<Nº línea>*

Cada bloque que se defina debe comenzar por la directiva correspondiente. Si no es así se lanza este error. Usualmente este error solo aparece cuando empezamos a introducir instrucciones en el programa o declaraciones de datos desde cero, sin definir ninguna directiva previa.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

```
; .org 100H                    ;.data 10H
        mvi b, 5
salto : mov a,b                dB   0, 127, FFh
        add b                  dS   "Error en directiva"
         jmp salto
```

	Error:	Identificador precediendo directiva no valido. Línea <*Nº línea*>

Antes del punto (.) que marca el comienzo de una directiva no puede haber nada salvo espacios en blanco o tabulaciones. En caso de que esto no se cumpla se genera el mensaje de error anterior.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
Directiva .org 100H
        mvi b, 5
salto : mov a,b
        add b
         jmp salto
```

	Error:	Directiva no reconocida. Línea <*Nº línea*>

Solo existen las directivas mencionadas anteriormente, esto es, DEFINE para las definiciones, DATA para las declaraciones de datos y ORG para comienzo del

programa. El ensamblador no es sensible a la capitalización de estas letras, siempre y cuando se correspondan con alguna directiva valida.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.mi_directiva 100H
        mvi b, 5
salto : mov a,b
        add b
         jmp salto
```

	Error:	Dirección incorrecta en la directiva. Línea <*Nº línea*>

Las directivas DATA y ORG necesitan una dirección de memoria inicial a partir de la cual colocar los datos e instrucciones respectivamente. Tanto si no se introduce esta dirección así como si es inválida se genera este error.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

```
.org 1A
        mvi b, 5
salto : mov a,b
        add b
         jmp salto
```

```
.data

dB   0, 127, FFh
dS   "No hay dirección
origen"
```

En el primer caso la expresión es errónea, ya que para definir un valor hexadecimal es necesario introducir el sufijo 'h'. El número no es reconocido correctamente dando lugar al error. Este caso se puede extender a cualquier otro en el que la expresión que decide la dirección de memoria es incorrecta.

En el segundo caso no se ha puesto ninguna dirección. Esto da también lugar a un error de ensamblado que genera el mensaje actual.

Errores en Definiciones

El objetivo de una definición es, simplemente, poner un nombre a un número. Durante el ensamblaje, cada vez que nos encontremos con un operando que contiene el nombre este será sustituido de forma automática por el valor correspondiente con el que se ha definido.

	Error:	Valor '<*Valor*>' asociado a definición '<*Definición*>' incorrecto. Línea <*Nº línea*>

El valor que se ha asociado a un identificador de una definición no es correcto. En una definición sólo se puede asignar un valor (o cualquier expresión que se resuelva en un valor) a un identificador. Este mensaje de error indica que el valor que se ha introducido no es valido.

- *Valor*. Secuencia de símbolos que no se pueden reconocer como un valor.
- *Definición*. Nombre del idenficador al que se intenta asignar un valor incorrecto o no reconocible.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

```
.define
  pvp 180A
  iva 16
.org 100H

...
```

```
.define
  sin_valor
  varios 16 17
.org 100H

...
```

En el primer ejemplo el valor asociado a “pvp” es incorrecto. Es un valor en hexadecimal al que le falta el sufijo ‘h’. Esto es extensible a toda expresión que contenga errores.

En el segundo ejemplo se muestran dos errores. En el primero no se ha especificado un valor al que asignar el identificado “sin_valor”, el segundo intenta asignar varios valores a la vez en la misma línea.

Error:	Colisión con la declaración previa de la etiqueta: '<*Nombre*>'. Línea <*Nº línea*>

Se pretende emplear un nombre identificador en una definición que ya ha sido previamente empleado para una etiqueta.

- *Nombre*. Identificador de la declaración que ya se ha empleado anteriormente.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

salto: mvi a,1
       add a
       jmp salto


.define

      salto 1000h
```

Error:	El identificador '<*Nombre*>' no es valido para una definición. Línea <*Nº línea*>

El nombre que se le quiere asignar a la definición no es valido. Esto se debe a que, o bien contiene caracteres inválidos, o que comienza por un número.

Recordemos que el primer carácter de un nombre de definición debe ser una letra del alfabeto. De no ser así, como ya vimos antes podría ser confundido con un número empleado en el programa.

- *Nombre*. Nombre de la definición que no es válido.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:	
`.define`	`.define`
`1salto 1000h`	`12345 1000h`

Errores en Declaraciones de Datos

Dentro de un programa se establece una clara diferencia entre datos e instrucciones. Los datos constituyen básicamente los operandos de las instrucciones.

Error:	Declaración no valida. Expresión incorrecta: '<*Valor*>'. Línea <*Nº línea*>

Existe un error en los valores introducidos en una declaración de datos. Ya sea de 8 (bytes) o 16 (words) bits, si el valor correspondiente no se puede calcular, debido a algún error tipográfico, se genera este error.

- *Valor*. Secuencia de símbolos que no se pueden reconocer como un valor.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:	
`.define`	`.define`
`1salto 1000h`	`12345 1000h`

Error:	Declaración incorrecta. No se han abierto comillas. Línea <*Nº línea*>

En la declaración de una cadena no se han abierto comillas antes de introducir la secuencia de letras correspondiente. Las comillas son necesarias para delimitar la

cadena que se quiere almacenar en memoria. Por esta razón es necesario que las cadenas de caracteres se expresen delimitadas por comillas.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.data 0H
   dS No se han abierto comillas”

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

	Error:	Declaración incorrecta. No se han cerrado comillas. Línea <*Nº línea*>

Como antes, pero el error contrario. Tras abrir comillas es necesario indicar el final de la secuencia de caracteres, esto se realiza mediante cierre de comillas. Aunque parezca una formalidad, el cierre de comillas puede evitar errores del programa que se deben a simples errores tipográficos.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.data 0H
   dS “No se han abierto comillas    ; Esto se metería

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

Como vemos, el problema de introducir cadenas de caracteres es que si estas no se delimitan correctamente se pueden introducir caracteres espurios.

Error:	Declaración incorrecta. Solo se puede introducir una cadena. Línea <*Nº línea*>

El prefijo de declaración de cadenas (dS) solo permite introducir una cadena de caracteres, a diferencia que dB y dW no es posible introducir secuencias consecutivas en la misma línea separadas por comas.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.data 0H
   dS "Cadena valida","Cadena no valida"

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

Error:	Prefijo '<*Prefijo*>' desconocido. Línea <*Nº línea*>

El ensamblador solo reconoce tres prefijos distintos. Estos son, el prefijo de enteros de 8 bits o Bytes (*dB*), el prefijo de palabras de 16 bits o Words (*dW*) y el prefijo de cadenas de caracteres o Strings (*dS*).

El ensamblador no es sensible a la capitalización de los prefijos, pero cualquier otro prefijo es rechazado generando este mensaje de error.

- *Prefijo*. Secuencia de caracteres que el ensamblador no reconoce como prefijo.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

```
.data 0H
   dB FFh
   dW FFFFh
   dS "Cadena valida"
   dI 0a123                  ;Prefijo no valido
   iva 16                    ;Prefijo no valido
   mvi a,1                   ;Prefijo no valido

.org 100H

salto: mvi a,1
       add a
       jmp salto
```

Errores en Instrucciones

Las instrucciones del lenguaje ensamblador vienen dadas por una serie de reglas que conforman la sintaxis de este.

	Error:	Mnemotécnico '<*Mnemotécnico*>' desconocido. Línea <*Nº línea*>

Los mnemotécnicos representan el código de cada instrucción. Cada una de las instrucciones tiene un código determinado, que es privativo de la misma, y que debe aparecer en el campo del código. Los códigos ensamblador están claramente definidos por este.

- *Mnemotécnico.* El mnemotécnico no se reconoce como una instrucción propia del procesador 8085.
- *Nº de línea.* Línea en la que se encuentra el error.

Ejemplo del error:

```
.data 0H
   dB FFh
   dW FFFFh
   dS "Cadena valida"

.org 100H

salto: mvi a,1
       paddsb a,b
       add a
       jmp salto
```

Error:	Sintaxis incorrecta en la instrucción. Primer operando no valido. Línea *<Nº línea>*

El primer operando de la instrucción no es un operando válido. Esto se puede deber a diferentes causas.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

```
.org 100H

salto: mvi a, 5
       add a
       jmp salto+(3*)
```

```
.org 100H

salto: mvi a,1
       push a
       jmp salto
```

En el primer ejemplo se produce este error debido a que la expresión introducida es errónea.

A la vez que se genera este error se muestra también el error de expresión correspondiente.

En el segundo caso se intenta emplear un registro no valido para la instrucción (push a), por esta razón también se muestra un error.

Error:	Sintaxis incorrecta en la instrucción. Segundo operando no valido. Línea *<Nº línea>*

El segundo operando de la instrucción no es un operando válido. Esto se puede deber a las diferentes causas explicadas anteriormente.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

```
.org 100H

salto: mvi a, 5+3*(5+)
       add a
       jmp salto
```

```
.org 100H

salto: mov a, 1
       add a
       jmp salto
```

En el primer ejemplo se produce este error debido a que la expresión introducida es errónea.

A la vez que se genera este error se muestra también el error de expresión correspondiente.

En el segundo caso se intenta emplear la función *mov* con una carga inmediata, lo cual no es posible (para eso está *mvi*). Aquí también se genera el mensaje de error.

Error:	Sintaxis incorrecta en la instrucción. Falta operando. Línea *<Nº línea>*

La información contenida en el campo operando se usa conjuntamente con el campo del código (mnemotécnico) a fin de definir con precisión la operación a ejecutar por la instrucción.

Según el contenido del campo del código, el campo del operando puede no existir, o consistir en una cifra o palabra, o bien en dos, separados ambas por una coma.

Si la instrucción necesita más operandos de los que hay especificados se producirá este error.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

salto: mov a
       add a
       jmp
```

MOV es una instrucción que necesita necesariamente dos operandos. Solo hemos introducido uno, por lo que se genera un error.

De igual forma *JMP* necesita un operando.

	Error:	Sintaxis incorrecta en la instrucción. Sobra operando. Línea <*Nº línea*>

Situación contraria a la anterior en la que no falta un operando, sino que al contrario, hay operandos de más.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

salto: mov a,b,c
       add a
       jmp salto, salto2
```

Cualquier instrucción en el 8085 no requiere más de dos operadores. MOV tiene tres, por lo que se genera un error.

De igual forma *JMP* solo necesita un operando.

	Error:	Sintaxis incorrecta en la instrucción. MOV no permite dos direcciones de memoria como operandos. Línea <*Nº línea*>

La instrucción de transferencia permite tres tipos de transferencias: transferencia entre registros (direccionamiento registro), transferencia desde la memoria (direccionamiento registro indirecto) y transferencia hacia la memoria (direccionamiento registro indirecto). En ningún caso permite la transferencia a la vez desde y hacia la memoria, esto es, especificar ambos operandos como direcciones de memoria.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

salto: mov M,M
       add a
       jmp salto
```

Errores en Expresiones

Una expresión es una secuencia de números y operadores que pueden resolverse. En ensamblador, a diferencia que otros lenguajes de más alto nivel, las expresiones han de resolverse en tiempo de compilación y no de ejecución, ya que el ensamblador solo traduce instrucciones, no compila. Por esta razón, las expresiones en ensamblador no contienen variables (aunque sí definiciones) ni registros.

	Error:	Expresión incorrecta. Falta operando en expresión entre paréntesis. Línea *<Nº línea>*

En las expresiones es posible utilizar paréntesis. Si realizando una operación se nos olvida poner un operando y cerramos el paréntesis se generará este error.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

`(5+ )`
`16*75+(- )`
`('A'-15* )`

Error:	Expresión incorrecta. Operando tras operador no encontrado. Línea <*Nº línea*>

Tras un operador no se encuentra el operando sobre el que se aplica.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

`5+`
`16*75+`
`'A'-15*`

Error:	Expresión incorrecta. Operador no reconocido. Línea <*Nº línea*>

Se ha empleado un operador distinto de los disponibles.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:

`Sqrt(5)`
`Solve(x+5=0)`
`5 Modulo 2`

Error:	Expresión incorrecta. Falta operador en expresión entre paréntesis. Línea <*Nº línea*>

Al definir una operación de mayor prioridad mediante los paréntesis no se ha especificado el operador correspondiente a dicha operación.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:
`(5 16)`
`(16*5 75)`
`(('A' 15) (5+8))`

Error:	Expresión incorrecta. No se han cerrado todos los paréntesis. Línea <*Nº línea*>

Se han abierto más paréntesis de los que se han cerrado.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:
`(16*75`
`('A'-15)*(5+5`
`(NOT((5+4)*6)`

Error:	Expresión incorrecta. División por cero. Línea <*Nº línea*>

En la expresión se intenta llevar a cabo una división por cero.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplos del error:
`7/0`
`(55*13)/NOT(FFh)`
`15/((16*32+2)-(64*8)-2)`

Errores en Etiquetas

El objetivo de una etiqueta es referenciar una dirección de memoria. Esto es, cuando una etiqueta se sitúa en la declaración de un dato o delante una instrucción de programa esta haciendo referencia a la dirección de memoria en la que se encuentra dicho dato o instrucción.

Error:	No se ha dado nombre a la etiqueta. Línea *<Nº línea>*

Este error aparece cuando se han puesto los dos puntos (:) pero no se ha dado nombre a la supuesta etiqueta.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

    mvi b, 5
  : mov a,b
    add b
    jmp salto
```

Como vemos, aunque existe el símbolo indicativo de etiqueta (:) esta realmente no existe.

Error:	Identificador de etiqueta no valido. Línea *<Nº línea>*

Cuando al nombre de una etiqueta le anteceden caracteres diferentes de espacio o tabulador se genera este error. El nombre empleado para una etiqueta no puede contener espacios en blanco.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

                    mvi b, 5
Etiqueta de salto: mov a,b
                   add b
                    jmp 123
```

	Error:	El identificador '<*Nombre*>' no es valido para la etiqueta. Línea <*Nº línea*>

El nombre que se le quiere asignar a la etiqueta no es valido. Esto se debe a que, o bien contiene caracteres inválidos, o que comienza por un número. Recordemos que el primer carácter de la etiqueta debe ser una letra del alfabeto. De no ser así, como ya vimos antes podría ser confundido con una dirección real de memoria.

- *Nombre*. Idenficador asignado a la etiqueta que no es válido.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

        mvi b, 5
123 :   mov a,b
        add b
         jmp 123
```

	Error:	Colisión con la definición previa de: <*Nombre*>. Línea <*Nº línea*>

Se intenta establecer una etiqueta con un nombre identificativo que ya ha sido previamente empleado en una definición.

- *Nombre*. Identificador de la etiqueta que ya se ha empleado anteriormente.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.define
  salto 100h
.data (25)

.org 100H

         mvi b, 5
salto  : mov a,b
         add b
         jmp salto
```

Como vemos, existe una ambigüedad sobre hacia donde se debe saltar. El idenficador *salto* puede referirse o bien a la etiqueta o a la definición.

5.4.3. Mensajes de Advertencia

	Advertencia:	Declaración previa de la etiqueta: '<*Nombre*>'. Línea <*Nº línea*>

El nombre con el que se quiere definir la etiqueta fue previamente empleado para definir otra etiqueta, por lo que no debería reutilizarse. El ensamblador tomara la ultima declaración de la etiqueta como la válida.

- *Nombre*. Identificador de la etiqueta que ya se ha empleado anteriormente.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo de la advertencia:

```
.org 100H

salto : mvi b, 5
salto : mov a,b
        add b
        jmp salto
```

Como vemos, existe una ambigüedad sobre hacia donde se debe saltar. Por defecto el ensamblador optara por la última vez en que se declara la etiqueta de salto.

	Advertencia:	Declaración previa de la definición: '<*Nombre*>'. Línea <*Nº línea*>

El nombre al que se quiere aplicar la definición fue previamente empleado en otra definición, por lo que no debería reutilizarse. El ensamblador tomara la ultima definición como la válida.

- *Nombre*. Identificador de la definición que ya se ha empleado anteriormente.
- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo de la advertencia:

```
.define
  maximo 100
  minimo 0
  media 50
  maximo 120

.org 100H

salto : mvi b, 5
        add b
        jmp salto
```

El nombre *maximo* se utiliza dos veces para realizar una definición. De esta forma el valor de *maximo* puede ser 100 o 120. Por defecto será 120.

	Advertencia:	La directiva define no necesita dirección origen. Línea <*Nº línea*>

Solo las directivas .DATA y .ORG requieren una dirección de origen a partir de la cual situar datos o instrucciones respectivamente.

Las definiciones solo se emplean en tiempo de compilación, realizándose las oportunas sustituciones. Si se especifica una dirección, esta es reconocida pero no se empleará en ningún momento.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo de la advertencia:

```
.define A0h
  maximo 100
  minimo 0
  media 50

.org 100H

salto : mvi b, 5
        add b
         jmp salto
```

El nombre *maximo* se utiliza dos veces para realizar una definición. De esta forma el valor de *maximo* puede ser 100 o 120. Por defecto será 120.

	Advertencia:	La dirección '<*dirección*>' no se puede almacenar en 16 bits. Línea <*Nº línea*>

El procesador 8085 solo puede trabajar con direcciones de 16 bits. Cualquier otra dirección mayor de 65535 (o FFFFh) no es direccionable.

Si una directiva .DATA o .ORG reciben una dirección origen mayor a FFFFh se generará esta advertencia. Solo los 16 bits menos significativos de la dirección se almacenarán.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 1FFFFH

        mvi b, 5
salto : mov a,b
        add b
        jmp salto
```

	Advertencia:	El valor '<*Valor*>' no se puede almacenar en un byte. Línea <*Nº línea*>

Al realizar una declaración de datos en forma de bytes estos deben de ser representables en 8 bits. Si no ocurre esto se lanza esta advertencia.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.data 0H
   dB 256, 100h, 100000000b

.org 100H

        mvi b, 5
salto : mov a,b
        add b
        jmp salto
```

	Advertencia:	El valor '<*Valor*>' no se puede almacenar en un word. Línea <*Nº línea*>

Como en el caso anterior, con las palabras de 16 bits. Si el valor declarado no se puede representar en 16 bits se lanza esta advertencia.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.data 0H
   dW 65536, 10000h, 10000000000000000b

.org 100H

        mvi b, 5
salto : mov a,b
        add b
        jmp salto
```

Advertencias:	El primer operando (<*Op*>) no se puede representar en 8 bits. Línea <*Nº línea*>
	El segundo operando (<*Op*>) no se puede representar en 8 bits. Línea <*Nº línea*>

Ciertas instrucciones permite especificar un operando inmediato de 8 bits. Estas advertencias aparecen cuando el operando inmediato no puede representarse en 8 bits.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

        mvi b, 100h
salto : mov a,b
        add b
        jmp salto
```

Advertencia:	El primer operando (<*Op*>) no se puede representar en 16 bits. Línea <*Nº línea*>

Ciertas instrucciones permite especificar un operando inmediato de 16 bits. Esta advertencia aparece cuando el operando inmediato no puede representarse en 16 bits.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

        mvi b, FFh
salto : mov a,b
        add b
        jmp 10000h
```

	Advertencia:	El segundo operando (<*Op*>) no se puede representar en 16 bits. Línea <*Nº línea*>

Ciertas instrucciones permite especificar un operando inmediato de 16 bits. Esta advertencia aparece cuando el operando inmediato no puede representarse en 16 bits.

- *Nº de línea*. Línea en la que se encuentra el error.

Ejemplo del error:

```
.org 100H

        mvi b, FFh
salto : mov a,b
        add b
        LXI H, 10000h
```