Cuaderno de Prácticas
Laboratorio de Fundamentos de Computadores

PARTE II:
Programación en ensamblador

Facultad de Informática
Universidad Complutense de Madrid

EL PROGRAMA ENSAMBLADOR

1. Descripción

El *programa* ensamblador es el programa que realiza la traducción de un programa escrito en ensamblador a lenguaje máquina. Esta traducción es directa e inmediata, ya que las instrucciones en ensamblador no son más que nemotécnicos de las instrucciones máquina que ejecuta directamente la CPU.

2. Estructura

Un programa en ensamblador está compuesto por líneas, conteniendo cada una de ellas un comentario, una única instrucción o una directiva.

En lo que sigue se utilizan los símbolos `<>` para encerrar un identificador o un número que el programador debe escribir; los símbolos `[]` encierran elementos opcionales; los símbolos `{}` encierran elementos que se puede escribir consecutivamente varias veces; el carácter `|` separa elementos opcionales.

2.1 Comentarios

Una línea con un asterisco (`*`) en la primera columna se interpretará como un comentario.

Ej:

```
* Esto son dos líneas de comentario porque el carácter
* de la primera columna es un asterisco
```

Un comentario también puede escribirse al final de la misma línea en que se escribe una instrucción.

Ej:

```
        ADD   D0,D1    Comentario sobre la instrucción ADD
```

2.2 Instrucciones

```
[<etiq>] <Cód_ope>.<tamaño>  <Fuente>,<Dest.>  [<Coment.>]
```

La etiqueta `etiq` es opcional; si existe debe empezar en la primera columna; si no existe debe haber por lo menos un espacio en blanco antes el código de operación. El comentario también es opcional.

No todas las instrucciones tienen dos operandos. Las hay con uno solo, e incluso sin ningún operando. Ejemplos:

```
Etiq1    MOVE.W D0,D1    Copiar el contenido de D0 en D1
         ADD.W    #3,D2
         NEG.W    D2
         BEQ.S    Etiq1
         NOP
```

2.3 Directivas

Controlan acciones auxiliares que realiza el programa ensamblador. No son traducibles a código máquina, sino que indican al ensamblador las preferencias del programador de cómo ha de efectuarse la traducción a lenguaje máquina. Ejemplo:

```
Variable   DS.B   1
```

`Variable` es la etiqueta, `DS.B` es la directiva y `1` es su argumento.

2.4 Control de ensamblado

`IDNT`: Identificación del programa (Identification). Debe ser la primera línea (no comentario) del programa. Tiene dos argumentos: los números de la versión y la revisión del programa. Ej:

```
Programa1   IDNT  1,0
```

`ORG`: Establece en qué posición de memoria absoluta se cargará el programa (ORiGin). Ej:

```
ORG     $2000
```

END: Marca el final del programa. A partir de esta directiva se ignora el resto del archivo. Ej:

```
END
```

2.5 Definición de símbolos

<Símbolo> EQU <Valor>

Sustituye **<Símbolo>** por **<Valor>** en el resto del programa (EQUal).

2.6 Definición y reserva de memoria

<Símbolo> DC.<tamaño> <Valor>[{,Valor}]

DC (Define Constant) reserva una o varias posiciones de memoria, cada una de tamaño **<tamaño>**, inicializándola con **<Valor>**. En el resto del programa podremos referirnos a esta posición de memoria por su **<Símbolo>**.

<Símbolo> DS.<tamaño> <Cantidad>

DS (Define Storage) reserva **<Cantidad>** posiciones de memoria de tamaño **<tamaño>**, sin asignarles ningún contenido inicial. En el resto del programa podremos referirnos a la posición inicial de este espacio de memoria por su **<Símbolo>**.

2.7 Bases de numeración

Un número puede expresarse en diferentes bases de numeración, precediéndolo del carácter que indica la base:

& (o nada) decimal

$	hexadecimal
%	binario
@	octal
"	cadenas ASCII

Se recomienda trabajar siempre en hexadecimal, ya que esta es la base de numeración que emplea el simulador al mostrar las direcciones de memoria y el contenido de la memoria y de los registros.

3. Modelo de programa

Todos los programas que se desarrollen en este laboratorio deben seguir el siguiente esquema:

1.- Directivas iniciales: identificación (**IDNT**) y especificación del origen absoluto del programa en la posición $2000 (**ORG $2000**).
2.- Bifurcación incondicional (**JMP Inicio**) al principio del programa, que se encuentra a continuación de la zona de datos.
3.- Declaración de símbolos **EQU**.
4.- Reserva de espacio de memoria para los datos del programa. (**DC** y **DS**)
5.- Instrucciones que componen el programa.
6.- Instrucción **STOP #$2700**, que causa que el simulador pare de simular. Esta instrucción debe ser alcanzable sea cual sea el flujo de ejecución del programa.
7.- Directiva **END**, que el ensamblador reconoce como el fin del código fuente.

Se añadirán comentarios, tanto en líneas separadas como en aquellas instrucciones que se consideren necesario. Además se respetará el formato de sangrías del siguiente ejemplo:

```
************************************************
*              PRACTICA 0: Ejemplo             *
************************************************
*
Cálculos    IDNT     1,1
            ORG      $2000
            JMP      Inicio
*
Valor1      EQU      7
*
X           DC.W     Valor1
*
Inicio      MOVE.W   X,D0
            STOP     #$2700
            END
```

PRÁCTICA 0: Codificación y depuración de programas en ensamblador.

El objetivo de esta primera práctica es la toma de contacto con las herramientas de ensamblado y simulación de programas ensamblador del Motorola 68000, que se utilizarán para la realización de las prácticas de que consta esta parte de la asignatura.

El desarrollo de un programa se compone de las siguientes etapas:

- Especificación del problema.
- Diseño de un diagrama de flujo orientado al lenguaje objetivo (ensamblador del 68000).
- Escritura del programa en lenguaje ensamblador (codificación).
- **Edición** del programa fuente.
- **Ensamblado** y **enlace**.
- **Simulación**.
- **Depuración** de errores.
- **Optimización**.

En esta primera práctica se aprenderán las nociones básicas necesarias para la realización de las cinco últimas fases, que son las más directamente relacionadas con el trabajo a efectuar en el laboratorio.

En lo que sigue, utilizaremos un problema ejemplo sobre el que basaremos esta práctica. Este ejemplo consiste en la siguiente especificación:

Dados dos números X e Y, calcular:

1.- su suma
2.- el mayor de ellos.

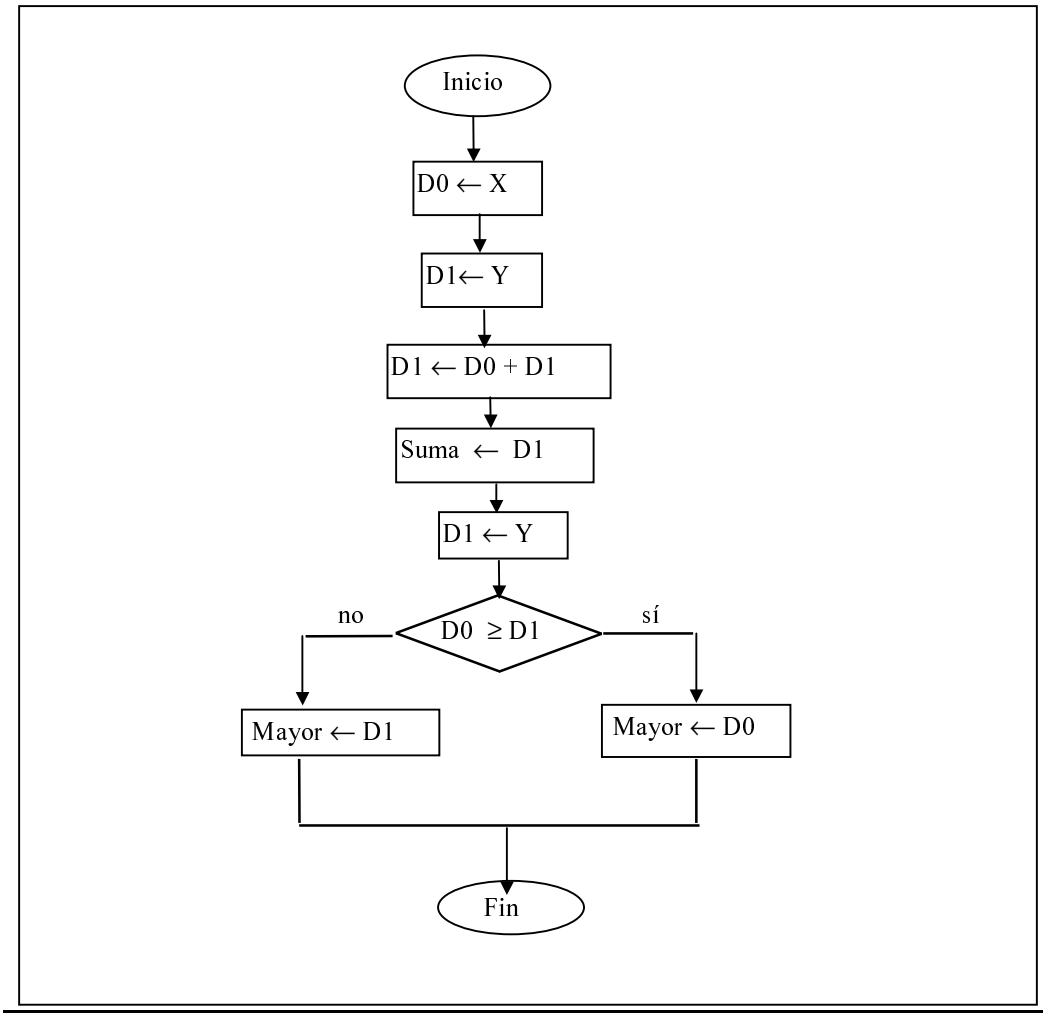
Una especificación más formal del problema es la siguiente:

Dados dos números naturales X e Y, se pide calcular los resultados Suma y Mayor:

$$Suma = X + Y$$

$$Mayor = \begin{cases} X, si\ X \geq Y \\ Y, si\ X < Y \end{cases}$$

Como primera aproximación, realizaremos un diagrama de flujo de las operaciones, en donde detallamos los registros que necesitamos y la colocación espacial de los bloques. Debemos tener en cuenta que un programa en ensamblador no se compone únicamente de instrucciones que expresan de forma abstracta el algoritmo que implementa, sino que, al contrario de lo que ocurre en alto nivel, el programador necesita sopesar las distintas opciones que la arquitectura final ofrece: dónde y cómo almacenar variables, cómo manejar datos y control, etc.



Tras el diagrama de flujo en donde, como vemos, hemos reflejado las etiquetas, las variables intermedias, etc, editaremos el código sustituyendo las expresiones por instrucciones en ensamblador, resultando el siguiente listado, que corresponde al programa fuente:

```
*************************************************
*               PRACTICA 0: Cálculos            *
*************************************************
*
Cálculos IDNT   1,1         Nombre del programa
         ORG    $2000       Dirección de ubicación del
*                           programa
         JMP    Inicio      Salta a la primera instrucción
*
* Constantes
*
Valor1   EQU      7      Valor inicial para X
Valor2   EQU      25     Valor inicial para Y
*
* Variables
*
X        DC.W   Valor1    Número X
Y        DC.W   Valor2    Número Y
Suma     DS.W   1         Primer resultado
Mayor    DS.W   1         Segundo resultado
*
* Programa principal
*
Inicio   MOVE.W X,D0      Copia la var. X al registro D0
         MOVE.W Y,D1      Copia la var. Y al registro D1
         ADD.W  D0,D1     Suma los registros D0 y D1
         MOVE.W D1,Suma   Almacena el resultado en Suma
         MOVE.W Y,D1      Vuelve a cargar Y en D1
         CMP.W  D1,D0     Compara los registros (D0-D1)
         BGE.S  Then      Salto si D0 es >= que D1
         MOVE.W D1,Mayor  Almacena D1 en la var. Mayor
         BRA.S  Fin       Salto al final del programa
Then     MOVE.W D0,Mayor  Almacena D0 en Mayor
*
Fin      STOP   #$2700    Finaliza el programa
         END
```

Observemos en ese listado lo que serán las zonas que poseerá todo programa en ensamblador, zonas heredadas de las existentes en los programas de alto nivel.

1.- Información del programa: número de práctica, nombre,...
2.- Una identificación del mismo: directiva **IDNT**.
3.- Especificación del lugar de carga del programa: directiva **ORG**.
4.- Salto incondicional a la primera instrucción que implementa el algoritmo.
5.- Zona de datos del programa, desglosada en declaración de constantes (directiva **EQU**) y en declaración de almacenamiento de variables (directivas **DC** y **DS**).
6.- Cuerpo del programa.
7.- Instrucción de fin de ejecución: **STOP #$2700**.
8.- Cuando existan subrutinas, zona de declaración de las mismas (en este programa tampoco aparecen).
9.- Fin de código fuente: directiva **END**.

En las próximas secciones veremos cada uno de los puntos que hemos resaltado en el proceso del desarrollo de un programa.

1. Edición del programa fuente

Para editar el programa fuente podríamos usar cualquier editor de texto convencional; en nuestro caso usaremos el que nos proporciona el sistema operativo MS-DOS:

C:\> EDIT <programa>.SA

Se indicará como argumento el nombre del fichero donde se va almacenar el texto del programa fuente que deberá terminar con la extensión **.SA**.

Es un editor sencillo orientado a menús que permite el uso de ratón y con él seleccionar algunas órdenes simples tales como: cortar, pegar, copiar, buscar, abrir, cerrar, guardar, etc.

En nuestro caso, escribiríamos por ejemplo **EDIT P0.SA** para escribir o editar el programa fuente. Una vez escrito, seleccionaremos del menú **Archivo** la opción **Guardar** y tras esta **Salir**, de este modo tendremos en disco almacenado nuestro programa en ensamblador.

2. Ensamblado y enlazado

Una vez que disponemos de un fichero con el texto del programa fuente, deberemos traducirlo al formato que puede procesar la máquina. Para ello invocaremos al programa ensamblador-enlazador mediante la orden:

C:\> ens <programa>

Indicando únicamente el nombre del programa sin ninguna extensión, como por ejemplo **ens P0**.

Durante este proceso el programa ensamblador genera varios archivos. Uno de ellos es el listado de ensamblado, un fichero de igual nombre que nuestro programa y extensión **.PRN**, que contiene información sobre la ubicación de los errores sintácticos que posee nuestro código fuente. El contenido de este fichero, que puede mostrarse en pantalla o editarse, es puramente informativo: la corrección de los errores indicados debe hacerse editando el programa original, no en el listado de ensamblado.

Otro de los archivos generados es un fichero de extensión **.HEX** que contiene las instrucciones máquina en un formato especial de la casa Motorola formado por caracteres hexadecimales, denominado S-Record.

En nuestro caso escribiríamos:

C:\> ens P0

Si hubiésemos cometido el error de escribir OGR en lugar de la directiva ORG, habríamos obtenido un error del que nos informa el ensamblador:

```
>>>  ERROR  line[13] Invalid opcode   [13] P0.SA
                           13.                 OGR     $2000
```

Este error lo podemos observar también en el archivo P0.PRN.

En este archivo aparecen a la derecha las direcciones en hexadecimal de cada una de las instrucciones y su codificación hexadecimal, así como las variables del programa y su contenido en hexadecimal.

Supongamos que se ha arreglado el error anterior y examinamos el siguiente fragmento del archivo P0.PRN:

```
Dirección Contenido    Número de línea
00002000  4EF9 0000200E     8.          JMP  Inicio  Salta a la primera instruccion
                           9.  *
                          10.  * Constantes
                          11.  *
     # 00000007           12.  Valor1   EQU    7       Valor inicial para X
     # 00000019           13.  Valor2   EQU    25      Valor inicial para Y
                          14.  *
                          15.  * Variables
                          16.  *
00002006  0007            17.  X        DC.W  Valor1   Numero X
00002008  0019            18.  Y        DC.W  Valor2   Numero Y
0000200A  <2>             19.  Suma     DS.W  1        Primer resultado
0000200C  <2>             20.  Mayor    DS.W  1        Segundo resultado
                          21.  *
                          22.  * Programa principal
                          23.  *
0000200E  3038 2006       24.  Inicio MOVE.W X,D0      Copiar la variable X al registro D0
```

Podemos ver que el programa empieza en la dirección 00002000 (2000 hexadecimal), en donde se produce el salto a Inicio. A esta instrucción de salto le corresponde la codificación 4EF9 y la dirección a la que se salta será 0000200E, que coincide con la dirección de la instrucción etiquetada como Inicio.

Por otra parte, podemos ver que la variable X (inicializada a Valor1) está en la dirección 2006, y que su contenido es 0007 en hexadecimal, que en decimal es 7. Este es el valor definido previamente para la constante Valor1, como podemos observar en el programa.

3. Simulación

Una vez ensamblado el programa, podemos proceder a su simulación. Arrancamos el programa simulador SIM68K mediante la orden:

C:\> simu

Inmediatamente aparecerá el indicativo del programa:

sim68k |

A partir de este momento, el simulador está listo para recibir órdenes e interpretarlas. Las órdenes se escriben directamente en respuesta al indicativo y constan de dos letras identificadoras y algunos argumentos.

En cualquier momento pueden mostrarse en pantalla la lista de órdenes disponible pulsando Intro. Sin embargo, por el momento, no los usaremos todos, tan sólo los más interesantes que nos permitirán realizar las siguientes acciones:

- Reiniciar el procesador.
- Cargar en memoria las instrucciones y datos de un programa.
- Ver y/o modificar el contenido de una posición de memoria dada.
- Desensamblar el contenido de la memoria.
- Ver y/o modificar el contenido de los registros del procesador.
- Ejecutar las instrucciones contenidas en memoria.
- Depurar los programas usando puntos de ruptura y ejecución paso a paso.

Cuando se entra en el simulador, éste se comporta exactamente como una CPU M68000, tal como la ve el programador, con el espacio de direccionamiento completo totalmente accesible, es decir, como si realmente tuviese 16 Mbytes de memoria RAM, ya inicializado y listo para comenzar a trabajar.

3.1. Inicialización del procesador

Como cualquier circuito secuencial, para que el procesador funcione con corrección debemos inicialmente llevarlo al primer estado. Eso se consigue con la orden:

sim68k | re

Que inicializa la CPU antes de ejecutar cualquier programa. También usaremos el mismo siempre que queramos que una nueva ejecución no dependa del estado previo del procesador.

3.2. Carga del programa

Tras inicializar el procesador, lo primero que debemos hacer es cargar en la memoria el programa que deseamos ejecutar, para ello usaremos la orden:

sim68k | lo <nombre>

No es necesario especificar la extensión. Una vez finalizada la orden, las instrucciones y los datos del usuario especificados en el programa fuente seleccionado se han cargado en las posiciones de memoria correspondientes, esta

carga se hará a partir de la posición indicada por la directiva **ORG** contenida en dicho programa.

Por tanto, tras esta orden, el programa se halla listo para su ejecución.

En nuestro ejemplo se reduce a escribir **lo p0**.

A partir de este punto se darán los fundamentos del manejo del simulador para que se puedan aplicar en el aprendizaje con el programa de ejemplo. Se trata, pues, de que cada uno practique con el simulador examinando y alterando posiciones de memoria y registros, y ejecutando el programa completo y realizando la depuración paso a paso. La práctica a partir de este punto se centra en aplicar la teoría que se presenta a continuación en el programa de ejemplo.

3.3. Visualización y modificación de la memoria

Podemos comprobar que el programa ha sido cargado correctamente en memoria examinando el contenido de la misma. La orden que permite visualizar la memoria entre dos posiciones dadas es:

sim68k | dm <dirección_inicial> [<dirección_final>]

La orden visualiza siempre como mínimo grupos de 16 bytes, a partir de la dirección inicial, en el formato siguiente:

```
          8    A    C    E    0    2    4    6
00002048  303C 0001 0C40 000F 6E0E 3E00 CEFC 0002  0<...@..n.>.....
```

La primera columna muestra la dirección donde comienza el volcado de memoria, las siguientes 8 columnas indican los contenidos, en hexadecimal, de las siguientes 16 posiciones, y la última muestra el valor ASCII de esos contenidos. En caso de no especificar la dirección final, mostrará de manera predeterminada 256 bytes.

También existe una orden que permite visualizar el contenido de la memoria en forma simbólica, es decir, interpretándolo como instrucciones. Este proceso se llama desensamblado y podemos invocarlo con la orden:

sim68k | di <dirección_inicial> [<dirección_final>]

Debemos hacer notar que el simulador no conoce si lo que está mostrando son datos o instrucciones, así que el usuario deberá interpretar lo que vea, para decidir si tiene sentido o no. Por ejemplo, la orden mostrará las anteriores posiciones de memoria, de la forma:

```
00002048  303C 0001    MOVE.W   #$0001,D0
0000204C  0C40 000F    CMPI.W   #$000F,D0
00002050  6E0E         BGT      $00002060
00002052  3E00         MOVE.W   D0,D7
00002054  CEAC 0002    ADD.W    #$00002,D7
```

La primera columna muestra las direcciones donde comienza el desensamblado, la segunda el contenido hexadecimal y la tercera la posible instrucción simbólica.

Pero el simulador no sólo permite visualizar la memoria, sino también modificarla, usando la orden:

sim68k | em <dirección>

Cuando se emplea esta orden, el simulador entra en un modo de edición, que permite modificar la memoria byte a byte a partir de la posición especificada. Como respuesta a esta orden, nos mostrará el contenido de dicha posición:

00002048 3A

quedando a la espera de lo que le indique el usuario, éste entonces podrá elegir entre:

- Volver al modo normal, saliendo del modo de edición de la memoria, pulsando Intro (no se modifica nada).

- Modificar el contenido en esa dirección, escribiendo el nuevo valor a almacenar (dos dígitos hexadecimales). A continuación se pasará a editar el siguiente byte de la memoria.

- Pasar a editar el contenido del siguiente byte sin modificar el actual, pulsando "**.**".

- Pasar a editar el contenido del byte precedente sin modificar el actual, pulsando "**^**" (y a continuación la barra espaciadora, debido a que el teclado que utilizamos está configurado en español, y la tecla "^" funciona de manera especial).

Esta orden será útil para introducir directamente en memoria nuevos valores de los datos de entrada sin necesidad de modificar el programa fuente. Así podremos comprobar rápidamente el comportamiento del programa para conjuntos diferentes de datos.

3.4. Visualización y modificación de registros

La orden:

sim68k | dr

visualiza los valores, en hexadecimal, de todos los registros del procesador, incluyendo aquellos de propósito particular: el PC (contador de programa) que fija la siguiente instrucción a ejecutar y el SR (registro de estado) en donde se indica individualmente el estado de cada uno de los códigos de condición..

Si lo que deseamos es modificar el valor de uno de cualquiera de los registros, bastará ejecutar la orden:

sim68k | sr <nombre_registro> <nuevo_valor>

donde el nombre del registro es su nombre convencional (D1, A3, PC ...).

3.5. Ejecución del programa

La primera instrucción de nuestros programas se encontrará siempre en la posición $2000, debido a la directiva **ORG $2000**, que incluiremos al principio de todos nuestros programas. Por lo tanto , el primer pasa para ejecutar (en este caso, simular) nuestro programa, es poner el valor $2000 en el contador de programa PC (registro que apunta a la dirección de memoria donde se encuentra la próxima instrucción a ejecutar):

sim68k | sr pc 2000

A continuación, diremos al simulador que empiece la ejecución mediante la orden:

sim68k | ex [<numero_instrucciones>]

El programa se ejecutará comenzando por la instrucción señalada por el PC, hasta que o bien haya completado el número de instrucciones especificadas, o encuentre un punto de ruptura, o encuentre una instrucción **STOP**. En cualquier caso la ejecución se detendrá y se presentará la próxima instrucción a ejecutar y el contenido de los registros del procesador. Para continuar la ejecución bastará ejecutar nuevamente la misma orden.

3.6. Ejecución paso a paso (traza de un programa)

La traza permite ejecutar una única instrucción, visualizar el contenido de los registros después de la ejecución, y elegir entre continuar con la siguiente o devolver el control al simulador. En este último caso podemos examinar, si lo deseamos, el contenido de la memoria y después volver a invocar a cualquier orden, incluido una nueva traza. Este tipo de ejecución permite examinar cuidadosamente el funcionamiento de las zonas más críticas de un programa y se invoca con la orden:

sim68k | ss <dirección_comienzo>

3.7. Puntos de ruptura

Un punto de ruptura (o breakpoint), es una marca que se coloca sobre una o varias instrucciones de un programa, de forma que la simulación se detiene inmediatamente antes de la misma. De esta forma, colocando un punto de ruptura en una zona conflictiva del programa, será posible detener su ejecución y examinar el contenido de la memoria y/o de los registros, para intentar detectar un posible error. Después podrá continuar normalmente la ejecución.

El simulador permite localizar cuatro puntos de ruptura, es decir, podemos definir cuatro puntos distintos, numerados del 1 al 4, asociándoles direcciones de memoria que serán recordadas por el simulador hasta que termine la sesión. Para asociar un punto de ruptura con una dirección usaremos la orden:

sim68k | bk <numero_breakpoint> <dirección_memoria>

Si especificamos como dirección la 0, se borrará el punto de ruptura asociado al número correspondiente; si se omiten ambos argumentos, se mostrarán por pantalla las direcciones asociadas a todos los puntos de ruptura activos.

Es importante destacar que al asociar un punto de ruptura a una dirección de memoria, no modifica para nada el contenido de esa dirección, así mismo un punto de ruptura no es ninguna instrucción especial, simplemente es un vigía puesto por el simulador a petición del usuario. Después de la parada podremos examinar el contenido de la memoria o los registros para ver si se han producido los cambios esperados y continuar la ejecución en el punto en que se había detenido.

4. Técnicas de depuración

Una vez cargado en memoria el programa, en primer lugar hay que probarlo. Para ello se ejecutará el programa, y se comprobará si el funcionamiento es el correcto. En algunos casos esta comprobación resultará evidente, mientras que en otros será preciso examinar el contenido de la memoria o de determinados registros antes y después de la ejecución.

En la mayoría de los casos el funcionamiento del programa no será el esperado. En tal caso se deberá buscar en la estructura del programa los puntos claves (aquellos más propensos al error) y con la ayuda de un listado del ensamblado, ir colocando con buen criterio puntos de ruptura, los cuales nos permitirán examinar los resultados parciales producidos por el programa y a partir de ellos ejecutar algunas instrucciones paso a paso si es preciso. De esta forma se va avanzando, hasta el punto en donde los resultados se desvían de lo esperado.

En ocasiones puede suceder que el programa no se detenga en el punto de ruptura indicado, aunque debiera pasar por él. La causa de este problema es que el error afecta al flujo del programa (probablemente a alguna bifurcación condicional). En este caso deberán examinarse cuidadosamente las bifurcaciones problemáticas, y revisar las instrucciones que activan los códigos de condición sobre los que se produce la bifurcación.

5. Ejercicios a realizar

Sobre el programa de ejemplo se deben hacer los siguientes ejercicios:

1) Realizar los pasos de edición del programa fuente de ejemplo, su ensamblado y su corrección si hay errores.

2) Con el simulador hacer el reset del procesador seguido de la carga del programa en memoria.

3) Ver el programa en memoria con la opción de la visualización en forma simbólica. Identificar la correspondencia entre lo que se visualiza de esta forma y el programa fuente que se ha escrito, es decir, identificar la dirección de comienzo, las direcciones y contenidos de las variables y las instrucciones.

4) Ejecutar el programa e inspeccionar el nuevo contenido de las variables Suma y Mayor. Comprobar que almacenan el resultado correcto.

5) Ejecutar el programa de nuevo paso a paso. Comprobar cada alteración que se produzca en los registros o en memoria con las órdenes adecuadas del simulador.

6) Modificar en la memoria (no en el programa fuente sino con las instrucciones del simulador) las posiciones de memoria correspondientes a las variables X e Y de manera que contengan los números 20 y 10 respectivamente. Ejecutar de nuevo el programa y comprobar que el resultado es correcto.

7) Situar un punto de ruptura en la instrucción MOVE.W Y, D1. Ejecutar el programa y alterar la posición de memoria correspondiente a la variable Y para que contenga el número 30. Reanudar la ejecución y observar los resultados de las variables Suma y Mayor.

PRACTICA 1: Programa Simple. Estructuras de control.

OBJETIVO: Desarrollar un programa simple en ensamblador y verificar su corrección mediante la simulación.

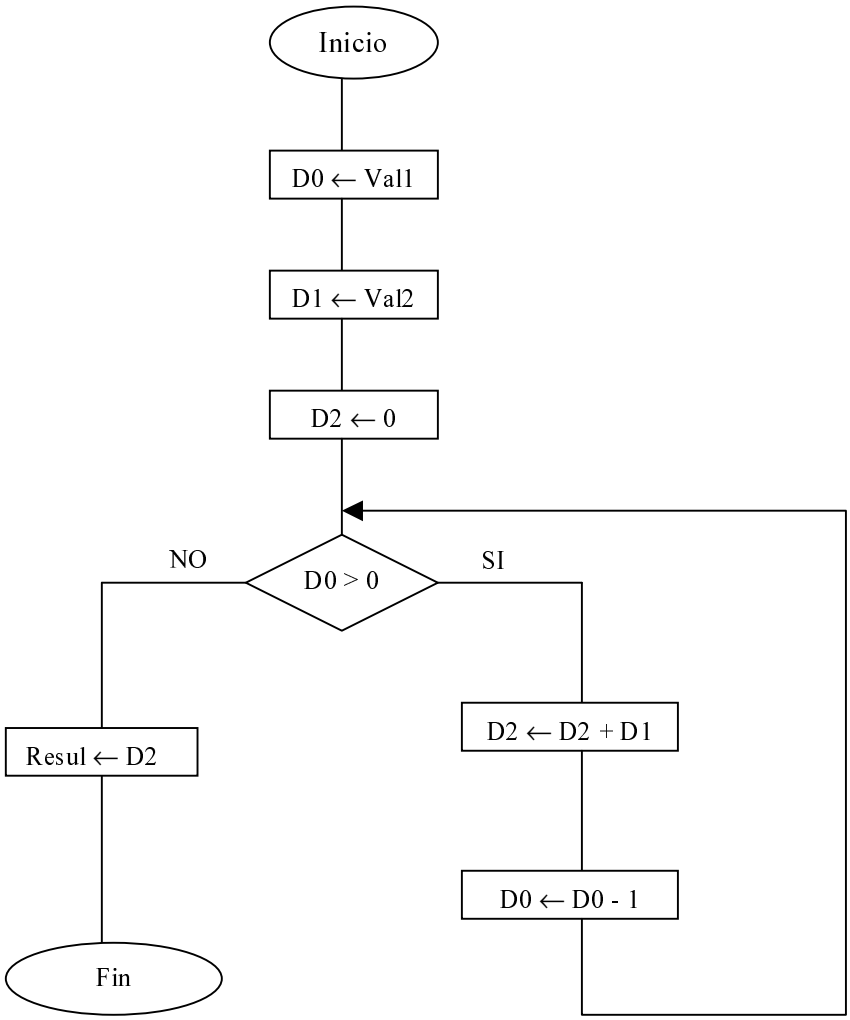
ESPECIFICACIONES:

Realizar un programa que multiplique dos números naturales de longitud WORD usando la instrucción de suma (ADD) en lugar de la instrucción de multiplicación que nos ofrece el repertorio del M68000.

El algoritmo es sencillo y puede expresarse brevemente en pseudo-PASCAL de la siguiente manera:

```
PROGRAM Mult;
CONST
     Val1 = 35;
     Val2 = 58;
VAR
     Mult1, Mult2 : WORD;
     Resul        : DOUBLE WORD;
BEGIN
     Mult1 := Val1;
     Mult2 := Val2;
     Resul := 0;
     WHILE Mult1 > 0 DO BEGIN
         Resul := Resul+Mult2;
         Mult1 := Mult1-1;
     END;
END.
```

El diagrama de flujo asociado a este programa sería el siguiente:



PRACTICA 2: Datos compuestos: vectores. Estructuras de control I.

OBJETIVO: Estudiar nuevos modos de direccionamiento y estructuras de control.

ESPECIFICACIÓN:

Parte A:

Realizar un programa que sume dos vectores y deposite el resultado en un tercer vector, utilizando la estructura de control "bucle FOR" y el método de direccionamiento indirecto con registro de direcciones con postincremento.

Parte B:

Realizar un programa que genere un vector resultado de invertir el orden de las componentes de otro dado.

Implementación:

Las componentes de los vectores sumando deben ser Byte. Las componentes del vector resultado deben ser Word. Hay que realizar extensiones de signo para asegurar operaciones correctas asumiendo que se trabaja con números enteros codificados en complemento a 2.

PRACTICA 3: Datos compuestos: vectores. Estructuras de control II.

OBJETIVO: Afianzar los conocimientos de evaluación de condiciones, estructuras de control y direccionamiento de vectores.

ESPECIFICACIÓN:

Diseñar un programa que calcule el máximo común divisor de dos números por el algoritmo de Euclides.

ALGORITMO: (en pseudo-PASCAL)

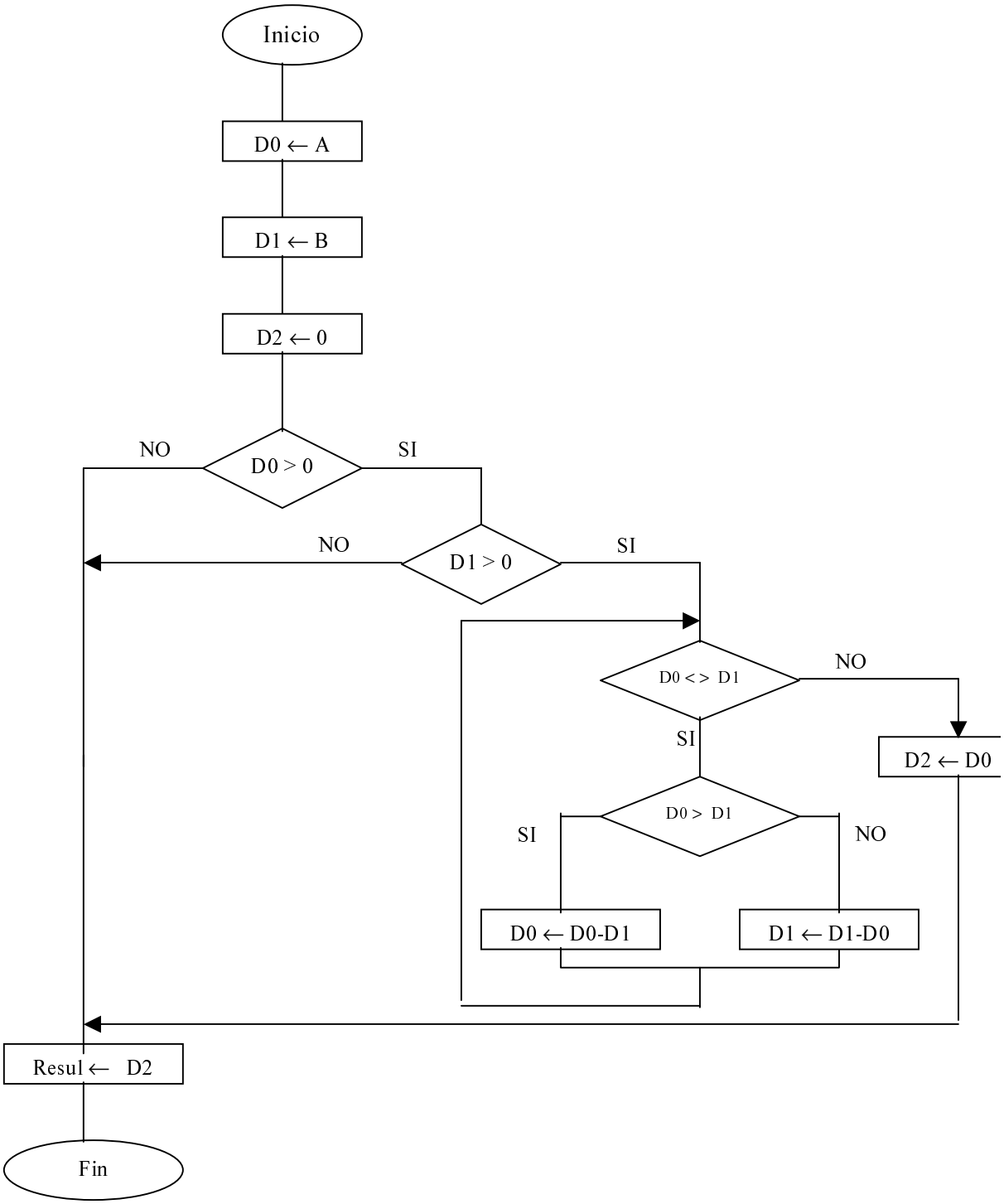
```
PROGRAM Mcd;

CONST    A := 51; B := 595;

VAR      Num1, Num2, Resul : NATURAL.W;

BEGIN
  Num1 := A; Num2 := B;
  IF  Num1 > 0  AND  Num2 > 0 THEN
  BEGIN
    WHILE  Num1 <> Num2  DO
      IF  Num1 > Num2  THEN  Num1 := Num1-Num2
                       ELSE  Num2 := Num2-Num1;
    Resul := Num1
  END
  ELSE  Resul := 0;
END.
```

El diagrama de flujo asociado a este programa sería el siguiente:


Inicio
D0 ← A
D1 ← B
D2 ← 0
D0 > 0
NO
SI
D1 > 0
NO
SI
D0 < > D1
NO
SI
D2 ← D0
D0 > D1
SI
NO
D0 ← D0-D1
D1 ← D1-D0
Resul ← D2
Fin

PRACTICA 4: Datos compuestos: matrices.

OBJETIVO: Tratamiento de vectores multidimensionales con el modo de direccionamiento indirecto con desplazamiento e índice.

ESPECIFICACIÓN:

Diseñar, implementar y verificar un programa en ensamblador que realice la multiplicación de dos matrices cuadradas de tamaño N x N.

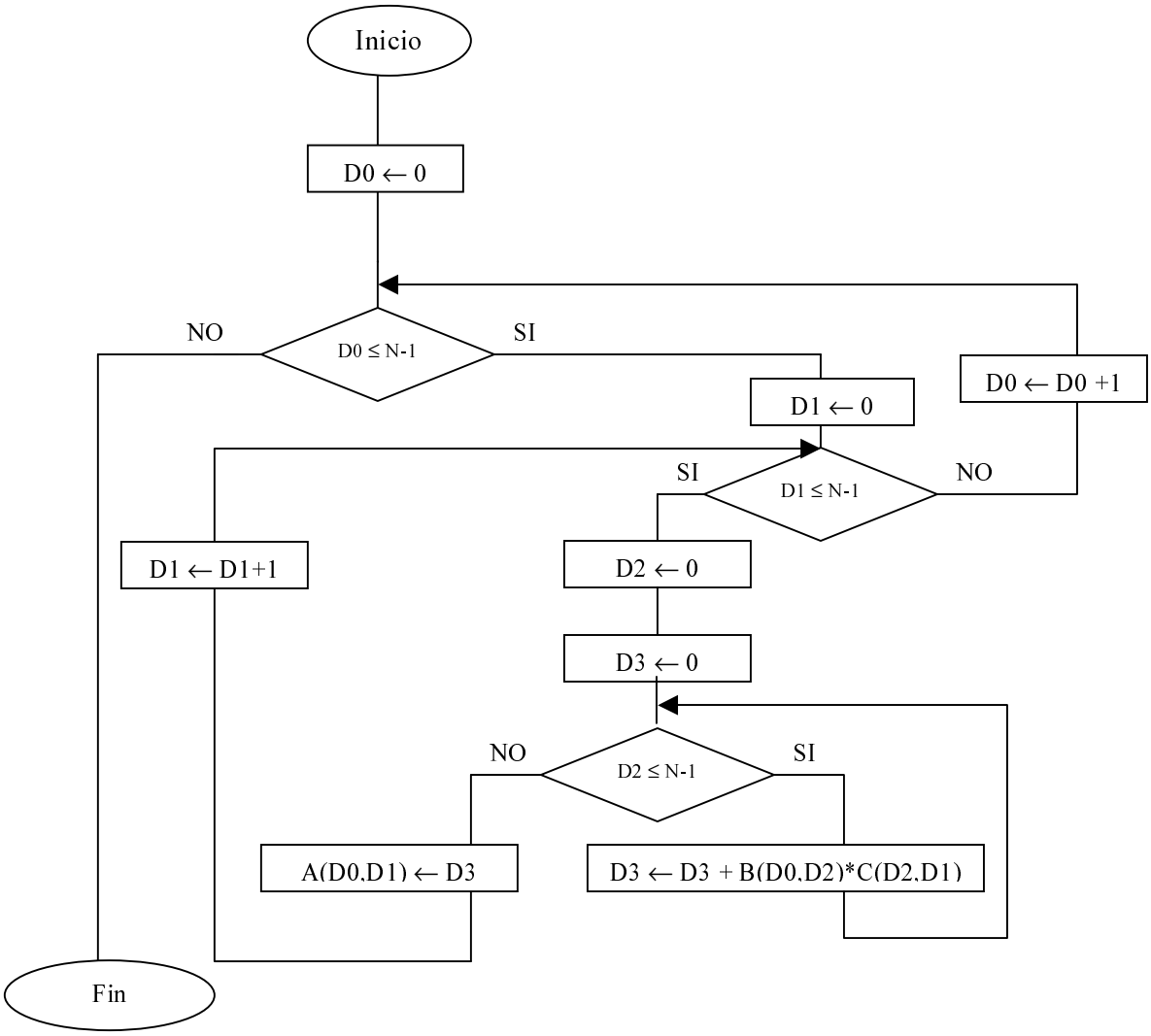
ALGORITMO: (en pseudo-PASCAL)

```
PROGRAM Mult_matrices;

CONST  N = 3;

VAR  B , C : ARRAY [0..N-1,0..N-1] OF WORD;
     A : ARRAY [0..N-1,0..N-1] OF LONG WORD;

BEGIN
  FOR  I := 0 TO N-1  DO
    FOR  J := 0 TO N-1  DO
    BEGIN
      S := 0;
      FOR  K := 0 TO N-1  DO
        S := S + B[I,K]*C[K,J];
      A[I,J] := S
    END;
END.
```


Inicio
D0 ← 0
NO
D0 ≤ N-1
SI
D0 ← D0 +1
D1 ← 0
SI
D1 ≤ N-1
NO
D1 ← D1+1
D2 ← 0
D3 ← 0
NO
D2 ≤ N-1
SI
A(D0,D1) ← D3
D3 ← D3 + B(D0,D2)*C(D2,D1)
Fin

APÉNDICE

1 Modelo de programación

1.1 Modelo de memoria

- Espacio de direccionamiento lineal.
- Espacio de direcciones de 16 Mb (direcciones de 24 bits).
- Direccionamiento a:

* bytes (8 bits)

Dirección	Contenido
N	Byte

* palabras (16 bits)

Dirección	Contenido
N	Byte más significativo (MSB)
N + 1	Byte menos significativo (LSB)

Notas:
N dirección par.
MSB: Most Significant Byte.
LSB: Least Significant Byte.

* dobles palabras (32 bits)

Dirección	Contenido
N	Palabra más significativa (MSW)
N + 2	Palabra menos significativa (LSW)

Notas:
N dirección par.
MSW: Most Significant Word.
LSW: Least Significant Word.

1.2 Modelo de registros

- 16 registros programables:

  * Ocho registros de datos de 32 bits (4 bytes).

    D0, D1, D2, D3, D4, D5, D6 y D7.

    Acceso al registro Di (0 =< i =< 7):

    Di.L Se accede a los cuatro bytes del registro (doble palabra, esto es, bits 31 al 0).
    Di.W Se accede a los dos bytes menos significativos (palabra menos significativa, esto es, bits del 15 al 0).
    Di.B Se accede al byte menos significativo (bits 7 al 0).

  - 8 registros de direcciones de 32 bits (4 bytes).

    A0, A1, A2, A3, A4, A5, A6 y A7.

    Acceso al registro Ai (0 =< i =< 7):

    Ai.L Se accede a los cuatro bytes del registro (pero sólo a los bits del 0 al 23).
    Ai.W Se accede a los dos bytes menos significativos (palabra menos significativa, esto es, bits del 15 al 0).

    No se permite el acceso al byte menos significativo (bits 7 al 0).

    Nota:
    A7 es el puntero de pila, también denominado SP.

- Registro no programable:

  * Registro de estado de 16 bits llamado SR.

Byte del sistema | Registro de estado (CCR)
15 ... 8 | 7 ... 4 3 2 1 0
N Z C

C: Acarreo Z: Cero N: Negativo

2 Modos de direccionamiento básicos

2.1 Inmediato

El valor del operando se incluye en la instrucción. Ej:

```
MOVE.B #5,D0   Almacena el número 5 en el registro D0
```

2.2 Registro directo

El operando es un registro. El valor es el contenido en el registro. Ej:

```
MOVE.B D1,D0  Copia el contenido de D1 en D0
```

2.3 Absoluto

El operando está en memoria. El valor es el contenido de la dirección de memoria que se incluye en la instrucción. Ej:

```
X    DC.L   543
     ...
     MOVE.L X,D1
```

Almacena el contenido de la dirección X (que representa a la variable inicializada con el valor 543) en el registro D1.

3 Instrucciones básicas

3.1 Copia de datos

```
MOVE.z  <fuente>, <destino>
```

z ::= B|W|L (byte, palabra o doble palabra)
fuente -> destino

3.2 Aritmética de enteros

Adición (ADDition)
`ADD.z  <operando1>,<operando2>`
operando2 <- operando1 + operando2

Substracción (SUBtraction)
`SUB.z  <operando1>,<operando2>`
operando2 <- operando2 - operando1

Puesta a cero (Clear)
`CLR.z  <operando>`
operando <- 0

Complemento a dos (NEGation)
`NEG.z  <operando>`
operando <- 0 - operando

Extensión de signo (EXTension)
`EXT.z  <operando>`
operando <- operando con signo extendido

3.3 Comparación

(CoMParison)

CMP.z <operando1>,<operando2>
operando2 - operando1, alterando sólo los códigos de condición

3.4 Bifurcaciones

Saltos incondicionales

1: BRanch Always

BRA.S <desplazamiento>
PC <- PC + desplazamiento

2: JuMP

JMP <dirección>
PC <- dirección

Salto si igual (Branch if Equal)

BEQ.S <desplazamiento>
Si Z = 1 entonces PC <- PC + desplazamiento

Salto si distinto (Branch if Not Equal)

BNE.S <desplazamiento>
Si Z = 0 entonces PC <- PC + desplazamiento

Salto si mayor (Branch if Greater Than)

BGT.S <desplazamiento>
Si N = 0 y Z = 0 entonces PC <- PC + desplazamiento

Salto si menor o igual (Branch if Less or Equal than)

BLE.S <desplazamiento>
Si N = 1 o Z = 1 entonces PC <- PC + desplazamiento

Salto si mayor o igual (Branch if Greater or Equal than)

BGE.S <desplazamiento>
Si N = 0 entonces PC <- PC + desplazamiento

Salto si menor (Branch if Less Than)

BLT.S <desplazamiento>
Si N = 1 entonces PC <- PC + desplazamiento

Notas:

- En la conjugación de la instrucción de comparación y de salto condicional se debe observar que cuando se comparan dos números, el orden de comparación es de derecha a izquierda. Considerando que D0 contiene el número 7 en el siguiente ejemplo:

```
CMP.B  #5,D0
BGT.S  Etiqueta
```

la condición que se comprueba en la instrucción **BGT** (salto si mayor) se cumple puesto que el segundo operando (7, el contenido de Do) es mayor que el primer operando (5).

- En las instrucciones de salto normalmente no se especificará un desplazamiento numérico, sino una etiqueta simbólica que aparecerá identificando una instrucción del programa. El ensamblador sustituirá automáticamente esta etiqueta por el desplazamiento numérico adecuado.

3.5 Otras

NOP

Sin operación (No Operation)

STOP #$2700

Detener la ejecución. No hay que olvidar escribirlo exactamente como se indica, con la almohadilla y el símbolo del dólar. Por el contrario, cuando se escribe ORG, no hay que escribir el símbolo del dólar para especificar la base de numeración de la dirección, que de manera predeterminada se entiende como hexadecimal.