

# CAPITULO 3: EL SIMULADOR DEL TMS320C30 EVM

## 3.1. INTRODUCCIÓN AL SIMULADOR DEL TMS320C30 EVM

A continuación, se explica con detalle cómo se ha realizado el simulador del TMS320C30 EVM, y los pasos dados durante este proceso.

Primeramente, se realizó el entorno gráfico del simulador. Gracias a la herramienta de programación *Borland C++ Builder 6*, se consiguió reproducir el entorno de depuración del TMS320C30.

El depurador del C30 es un interfaz avanzado para la programación, que ayuda al programador a desarrollar, probar y refinar programas en C y en ensamblador.

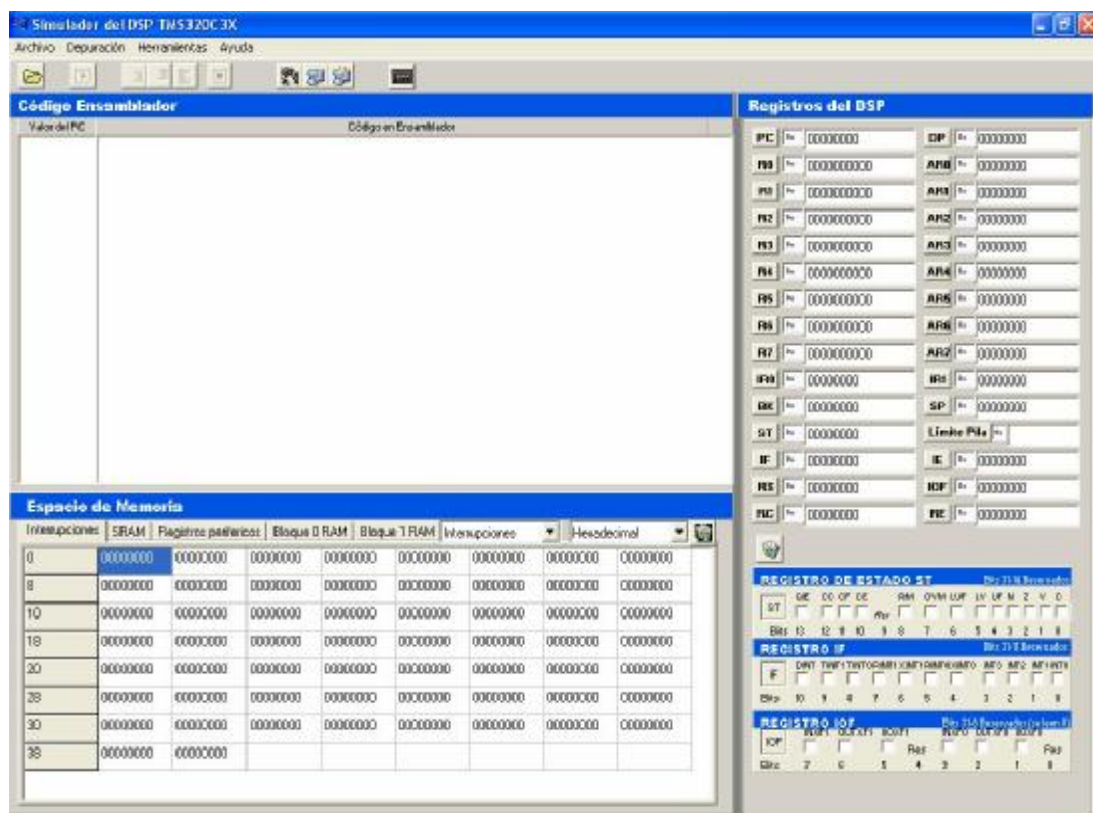


Figura 3.1. Simulador del TMS320C30 EVM

El simulador es una herramienta que permite interpretar código en formato ensamblador del DSP, además de poseer alguna particularidad del linker, como se explica posteriormente en este capítulo. Gracias a su portabilidad, permite el desarrollo de programas en lenguaje ensamblador sin necesidad de tener un módulo DSP.

## 3.2. COMPONENTES DEL DSP INCLUIDOS EN EL ENTORNO GRÁFICO

Como ya se ha comentado en el apartado anterior, la primera tarea que se ha desarrollado es el entorno gráfico. Para una correcta visualización de los componentes del TMS320C30 EVM, se ha procedido a dividir la pantalla del simulador en partes, incluyendo las siguientes:

- Ventana de registros del DSP: En ella se reflejan en cada momento los 29 registros del DSP, el límite de la pila (posteriormente se desarrollará este tema) y las banderas del registro de estado ST y los registros IF (*Interrupt Flag register*) e IOF (*I/O Flag register*). Además, existe un botón de limpieza de los registros.
- Ventana de memoria: Incluye las siguientes secciones de memoria:
  - Vectores de Interrupción: de la posición 0 hasta la 0x39 de memoria.
  - SRAM: de la posición 0x40 hasta la 0x3FFF.
  - Registros Periféricos: desde la posición 0x808000 hasta la 0x80806F (No es la zona de registros periféricos completa pero incluye la parte necesaria para el proyecto)
  - RAM 0: de la posición 0x809800 hasta la 0x809BFF.
  - RAM 1: de la posición 0x809C00 hasta la 0x809FFF.
- Ventana de ejecución: Es la ventana de visualización del código ensamblador. Los números de línea del programa ensamblador cargado son fijados automáticamente, pero esta particularidad se a lo largo de este capítulo.

### 3.2.1. REGISTROS

La ventana de visualización de registros permite ver los 29 registros del DSP. Los distintos formatos de visualización de los datos de cada registro son los siguientes:

- Hexadecimal: Ejemplo: 0xFFFFFFFF.

- Binario: Ejemplo: 1010111100001010.
- Entero con signo: Ejemplo: -234.
- Punto flotante: Ejemplo: -3.54E-07.
- Notación científica: Ejemplo: -2.34E+2.

**Registros del DSP**

<b>PC</b> 0x 00000000	<b>DP</b> 0x 00000000
<b>R0</b> 0x 000000000	<b>AR0</b> 0x 00000000
<b>R1</b> 0x 000000000	<b>AR1</b> 0x 00000000
<b>R2</b> 0x 000000000	<b>AR2</b> 0x 00000000
<b>R3</b> 0x 000000000	<b>AR3</b> 0x 00000000
<b>R4</b> 0x 000000000	<b>AR4</b> 0x 00000000
<b>R5</b> 0x 000000000	<b>AR5</b> 0x 00000000
<b>R6</b> 0x 000000000	<b>AR6</b> 0x 00000000
<b>R7</b> 0x 000000000	<b>AR7</b> 0x 00000000
<b>IR0</b> 0x 00000000	<b>IR1</b> 0x 00000000
<b>BK</b> 0x 00000000	<b>SP</b> 0x 00000000
<b>ST</b> 0x 00000000	<b>Límite Pila</b> 0x
<b>IF</b> 0x 00000000	<b>IE</b> 0x 00000000
<b>RS</b> 0x 00000000	<b>IOF</b> 0x 00000000
<b>RC</b> 0x 00000000	<b>RE</b> 0x 00000000

**REGISTRO DE ESTADO ST** Bits 31-14 Reservados

ST	GIE	CC	CF	CE	RM	OVM	LUF	LV	UF	N	Z	V	C
Bits	13	12	11	10	9	8	7	6	5	4	3	2	1

**REGISTRO IF** Bits 31-11 Reservados

IF	DINT	TINT1	TINT0	RINT1	XINT1	RINT0	XINT0	INT3	INT2	INT1	INT0
Bits	10	9	8	7	6	5	4	3	2	1	0

**REGISTRO IOF** Bits 31-8 Reservados (se leen 0)

IOF	INXF1	OUTXF1	I/OXF1	Res	INXF0	OUTXF0	I/OXF0	Res
Bits	7	6	5	4	3	2	1	0

**Figura 3.2. Registros del DSP**

El simulador permite la visualización del valor de cada registro en cualquiera de estos formatos. Como cada registro puede tener un formato distinto, esta propiedad será

independiente para cada uno de ellos. Tanto el programa como el usuario pueden cambiar dinámicamente el valor y formato de cada registro. La conversión de formatos para los distintos tipos de registro se explica en la sección 3.3.

Si hay algún cambio en el valor de registros ST (Registro de estado), IF (Registro de banderas de interrupción) o IOF (Registro de banderas de entrada / salida), se verá reflejado en la ventana de banderas (parte inferior), además de en la ventana del registro correspondiente. Al contrario, si modificamos alguna bandera de estos registros, el valor del registro queda modificado simultáneamente.

### 3.2.2. MEMORIA

Gracias a la disposición de la cuadrícula del programa, se puede comprobar fácilmente el valor de cada posición de memoria. Se ha dividido la ventana de memoria en cinco partes: vectores de interrupción, SRAM, registros periféricos, RAM0 y RAM1.

Espacio de Memoria								
Interrupciones	SRAM	Registros perifericos	Bloque 0 RAM	Bloque 1 RAM	Interrupciones	Hexadecimal		
0	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
8	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
18	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
28	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
38	00000000	00000000						

**Figura 3.3. Memoria del DSP**

La posición de los vectores de interrupción será fija en el comienzo del módulo de SRAM (0 hasta la 0x39). Debido a que no se ha implementado la funcionalidad de las interrupciones (sólo la del puerto serie), se ha preferido dejar esta zona fija en memoria.

La sección de memoria llamada SRAM (no confundir con el módulo SRAM), abarca desde la posición 0x40 hasta la 0x3FFF. Esta sección de memoria, junto con la de vectores de interrupción completa las 16K palabras de 32bits del módulo de SRAM.

La sección de memoria de registros periféricos abarca desde la posición 0x808000 del mapa de memoria hasta la posición 0x8097FF. Como la zona de esta sección que se ha utilizado es la del mapeado de los puertos serie, se puede visualizar en el simulador desde la posición 0x808000 hasta la 0x808060.

En cuanto a las secciones RAM0 y RAM1, la primera está mapeada desde la posición 0x809800 hasta 0x809BFF, y la segunda desde 0x809C00 hasta la 0x809FFF. Estas secciones pueden contemplarse como dos de 1K palabras, o una de 2K palabras, aunque la visualización sea partida en dos de 1K palabras.

Al igual que sucede con los registros, puede visualizarse las secciones de memoria en distintos formatos: hexadecimal, binario, entero con signo y punto flotante. La diferencia reside en que cuando se cambia la memoria de formato, el cambio se realiza para toda la memoria, sin posibilidad de alterar una parte concreta.

### 3.2.3. VENTANA DE EJECUCIÓN

Como ya se ha comentado en la sección 2.4.1, las utilidades para trabajar con el código ensamblador crean y usan ficheros objeto en un formato que TI ha llamado COFF (*Common Object File Format*). Los ficheros objeto contienen bloques separados (llamados secciones) de código y datos que se pueden cargar en diferentes zonas de memoria del C30.

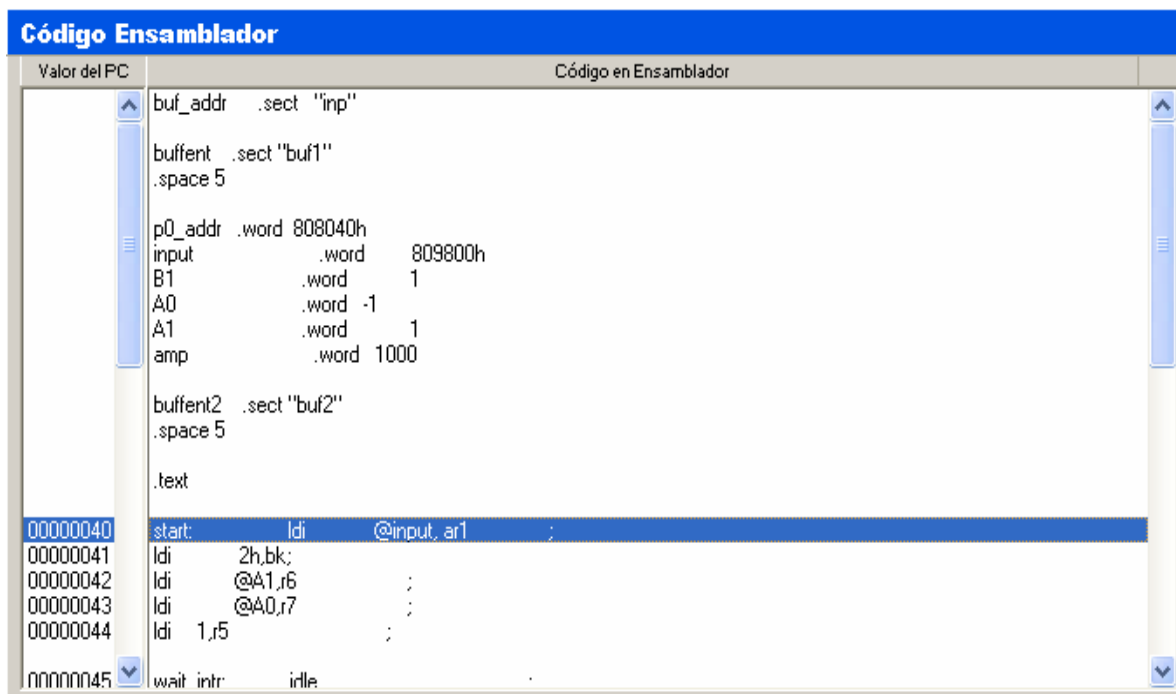


Figura 3.4. Memoria del DSP

En la ventana de ejecución se puede visualizar el código ensamblador, además del valor del contador de programa correspondiente a cada instrucción (en la sección 3.6.2. se explica el proceso de asignación de los valores del contador de programa a las instrucciones). En ella se pueden realizar diferentes tipos de ejecución del programa ensamblador, previamente cargado este en memoria.

### 3.3. FORMATOS

El paso siguiente ha sido la realización y definición de formatos que se manejan en el simulador. Estos formatos son: hexadecimal, binario, entero con signo, punto flotante y notación científica.

- Formato hexadecimal: Comprendido entre 0 y FFFFFFFF para posiciones de memoria y registros de 32 bits. Para los registros de precisión extendida (40 bits), el rango va desde 0 hasta FFFFFFFF.
- Formato binario: Es el mismo rango que para formato hexadecimal, si se transforma a binario.
- Formato entero con signo: Comprendido entre -2147483648 y 2147483647, que es el rango representable con 32 bits.
- Formato punto flotante: Este rango está dividido en dos zonas delimitadas por cuatro valores en punto flotante de 32 bits. Todo ello está ilustrado en la figura 3.5.
  - Máximo valor positivo: El máximo valor representable sin desbordamiento positivo es 6,8056469327705772E+038.
  - Mínimo valor positivo: El mínimo valor positivo sin desbordamiento positivo bajo es 5,877471754111438E-39.
  - Valor negativo de mayor magnitud: El valor negativo más grande será - 6,805646932770577200000E+038.
  - Valor negativo de menor magnitud: El valor negativo más pequeño sin desbordamiento negativo bajo será: -5,8774724547606697E-039.



- Notación científica: El rango es el mismo que para entero con signo, salvo que los números están representados en notación científica. Ejemplo: si se quiere representar el número -3457 en formato notación científica será: -3.457E3.

### 3.3.1. CAMBIOS DE FORMATO.

A continuación se explican las equivalencias entre los distintos formatos y cómo se han hallado estos equivalentes:

- Conversión hexadecimal - entero con signo: Como ya se ha comentado anteriormente en la sección 2.2.4.1, los enteros con signo representan mediante 32 bits, en complemento a dos si el número es negativo. De estos 32 bits, 31 corresponden al número entero al que vamos a convertir más un bit de signo. De esta manera, la equivalencia entre valores hexadecimales y formato entero con signo es :

$$\S \text{ 7FFFFFFFh } \mathfrak{B} \text{ } 2^{31}-1$$

$$\S \text{ 00000000h } \mathfrak{B} \text{ } 0$$

$$\S \text{ FFFFFFFFh } \mathfrak{B} \text{ } -1$$

$$\S \text{ 80000000h } \mathfrak{B} \text{ } -2^{31}$$

- Conversión hexadecimal - binario: El equivalente binario de una cifra hexadecimal se obtiene mediante 4 bits, cuyo valor puede oscilar desde 0000b = 0h hasta 1111b = Fh. De esta manera se puede obtener fácilmente la conversión binario – hexadecimal y viceversa.
- Conversión binario - entero con signo: Se realiza de la misma manera que para la conversión hexadecimal – entero con signo, es decir, se ha de tener en cuenta que para convertir un número binario a decimal si es negativo, se tiene que hacer el complemento a dos del valor para obtener el decimal correspondiente.

#### 3.3.1.1 Conversión a punto flotante.

A continuación se explica la conversión entre el formato decimal y el formato punto flotante IEEE 754. Para ello se formula el siguiente teorema:

Para cada  $x \in \mathbb{R}$ , (de signo  $s$ ) existe un par  $(m \in \mathbb{R}, \exp \in \mathbb{Z})$  que verifica:



$$x = (-1)^s * 1.m * 2^{exp}.$$

**Ecuación 3.1**

Lo que quiere decir este enunciado es que para cada número real, existe un número en formato punto flotante (en el simulador la representación en punto flotante está limitada por los 32 bits de la precisión simple).

Demostración:

Se calculan  $m$  y  $exp$ ; y así al mismo tiempo:

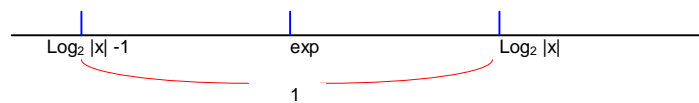
- Se demuestra que existen, es decir, que el teorema es cierto.
- Se proporciona un método para calcularlos.

a) Cálculo de  $exp$ .

$$|x| = 1.m \cdot 2^{exp} \Rightarrow \left\{ \begin{array}{l} |x| \geq 1.0 \cdot 2^{exp} = 2^{exp} \Rightarrow exp \leq \log_2 |x| \\ |x| < 1.0 \cdot 2^{exp} < 2 \cdot 2^{exp} = 2^{exp+1} \Rightarrow exp+1 > \log_2 |x| \Rightarrow exp > \log_2 |x| - 1 \end{array} \right\} \Rightarrow (\log_2 |x|) - 1 < exp \leq \log_2 |x|$$

**Ecuación 3.2**

Hay que tener en cuenta que en cualquier intervalo  $(r-1, r]$ , con  $r \in \mathbb{R}$ , hay sólo un entero de  $\mathbb{Z}$ .



**Figura 3.6**

Así pues,  $exp$  es el mayor entero de  $\mathbb{Z}$  que está entre los reales  $\log_2 |x| - 1$  y  $\log_2 |x|$

Conclusión: la forma de encontrar  $exp$  es:

1º: Calcular :  $\log_2 |x| = \frac{\ln |x|}{\ln 2}$

**Ecuación 3.3**

Lo más probable es que sea un real no entero.

2º: El  $exp$  buscado es el mayor de los enteros menores que  $\log_2 |x|$ , es decir:

$$exp = INT(\log_2 |x|)$$

#### Ecuación 3.4

b) Cálculo de  $m$ : (FRAC es la fracción)

$$|x| = 1.m \cdot 2^{exp} \Rightarrow 1.m = \frac{|x|}{2^{exp}} \Rightarrow m = FRAC\left(\frac{|x|}{2^{exp}}\right)$$

#### Ecuación 3.5

#### Corolarios:

1º.-

$|x| = 1.m \cdot 2^{exp} \Rightarrow$  En formato IEEE 754:

$$1.a) \text{ Campo mantisa } = m = FRAC\left(\frac{|x|}{2^{exp}}\right)$$

$$1.b) \text{ Campo exponente } = exp + \text{Exceso} = exp + (2^{n-1} - 1) = INT(\log_2 |x|) + (2^{n-1} - 1)$$

2º.-

$$|x| = 1.m \cdot 2^{exp} \Rightarrow |x| = (1 + 0.m) \cdot 2^{exp} = 2^{exp} + 0.m \cdot 2^{exp}$$

↓

La mayor potencia de 2 que se puede restar de  $|x|$  es  $2^{exp}$ .

↓

El primer bit significativo de  $|x| = b_{n-1} b_{n-2} \dots b_1 b_0 . b_{-1} b_{-2} b_{-3} \dots$  es  $b_{exp}$ .

Para ilustrar mejor este teorema, seguidamente se exponen varios ejemplos:

Ejemplo 1: convertir el número -2.5675e15 al formato IEEE 754 de 32 bits:

1) Expresar el número positivo y se iguala a una potencia de dos.

$$2.5675 \cdot 10^{15} = 2^{\text{exponente}}$$

Despejamos el exponente:

$$[\log(2.5675) + 15 \cdot \log(10)] / \log(2) = \text{exponente}$$

donde:

$$\text{exp} = 51.189$$

Se aproxima al valor inmediatamente inferior, o sea al 51. Esto se hace siempre. (Si fuera negativo, por ej. el -81.6, sería el -82)

Con esto se consigue una aproximación al número deseado. Para obtener el número exacto se hace lo siguiente:

$$2.5675 \cdot 10^{15} = x \cdot 2^{51}$$

donde:  $2^{51} = 2251799813685248$

y 'x' es el factor que, multiplicado a  $2^{51}$ , hace que se obtenga el número que buscamos:

$$x = 2567500000000000 / 2^{51} = 1.14019904629003576701$$

2) Pasar a binario el número 1.14.

La principal ventaja de este método es que sólo hay que hallar la parte decimal del número (0.140199...) porque la parte entera es 1 y va a ser el bit implícito o el de ahorro para el IEEE754.

$$0.14019904629003576701(\text{dec}) = 0.00100011111001000001011(\text{bin})$$

Con esto se obtiene la mantisa, ahora se calcula el exponente.

En la notación IEEE754 el exponente se pone en exceso, por lo que:

$$2^{(n-1)} - 1 + \text{exponente} = 2^7 - 1 + 51 = 127 + 51 = 178$$

3) Pasar todo a la notación IEEE754:

El primer bit es el de signo, que es el de un número negativo, y por tanto se pone a 1. Los siguientes 8 bits son los del exponente, por lo que se pone el 178 en binario en esos 8 bits y el resto (23 bits) son la mantisa que recuerda que se pone con ahorro de bit, eso quiere decir que el primer bit significativo de la mantisa se omite.

Quedaría así:

1 10110010 x00100011111001000001011

(en la x estaría un 1, pero como es el bit implícito, se descarta).

Ahora agrupándolos de 4 bits en 4 bits se obtiene el nº en hexadecimal

1101 1001 0001 0001 1111 0010 0000 1011

D 9 1 1 F 2 0 B

Es decir:

-2.5675e15 (dec) = D911F20B (ieee754)

Ejemplo 2: Convertir el número C19E0000 en formato IEEE754 a su equivalente decimal:

Primero se pasa de hexadecimal a binario:

C19E0000 = 1100 0001 1001 1110 0000 0000 0000 0000

Donde el signo es 1, el exponente 10000011 y la mantisa 001 1110 0000 0000 0000 0000.

La fórmula es:

$$(-1)^s * 1, \text{mantisa} * 2^{(e-127)}$$

Que el signo sea 1 significa, por tanto, signo negativo.

El exponente 10000011 es 131, como se representa en exceso a  $2^{(n-1)}-1$  hay que restar 127 para volver a tener el exponente real, que es por tanto 4.

En la mantisa, se van multiplicando hacia la derecha los bits por  $2^{-1}$ ,  $2^{-2}$ ,  $2^{-3}$ ,  $2^{-4}$ , ...).

$0 \cdot 0,5$

$0 \cdot 0,25$

$1 \cdot 0,125$

$1 \cdot 0,0625$

$1 \cdot 0,03125$

$1 \cdot 0,015625$

-----

0,234375 en decimal

Como está normalizada (se supone un 1 a la izquierda de la coma) es en realidad 1,234375. Resumiendo:

$$(-1)^1 \cdot 1,234375 \cdot 2^4 = -1,234375 \cdot 16 = -19,75$$

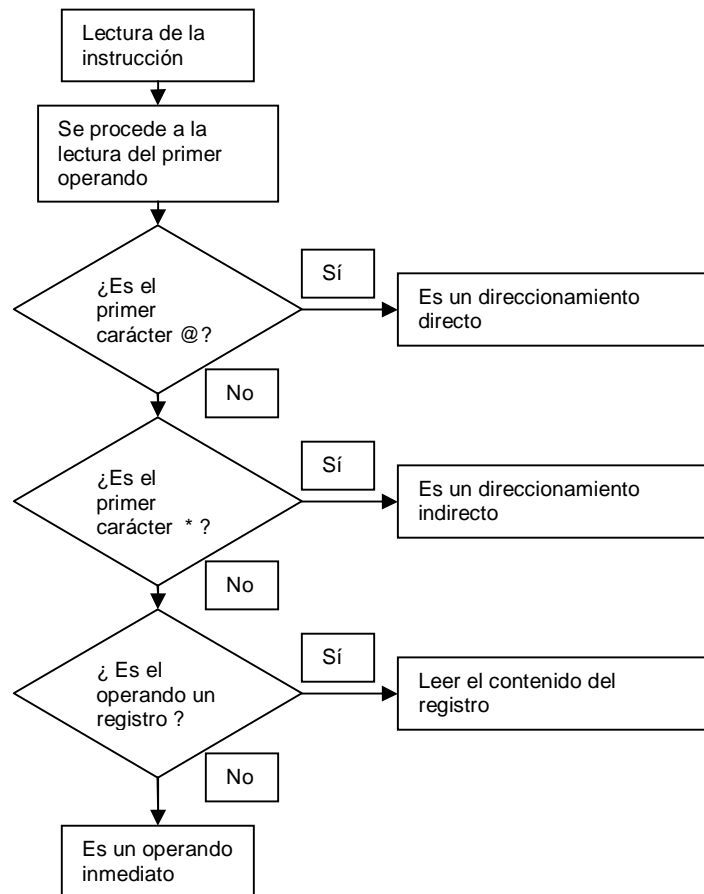
Hay que resaltar que al cero binario o al hexadecimal, siguiendo la metodología que se ha realizado, le correspondería el número flotante más pequeño representable en punto flotante. Para que no haya problemas en este aspecto, se debe redondear este valor a cero, estableciendo así una equivalencia con todos los formatos.

Con estos ejemplos se ha explicado la conversión de formato entre formato punto flotante y hexadecimal (o binario).

### 3.4. MODOS DE DIRECCIONAMIENTO

Como ya se explica en la sección 2.2.6, los modos de direccionamiento hacen referencia a la forma en que la CPU accede a los datos que necesita. El DSP TMS320C30 dispone de seis modos de direccionamiento.

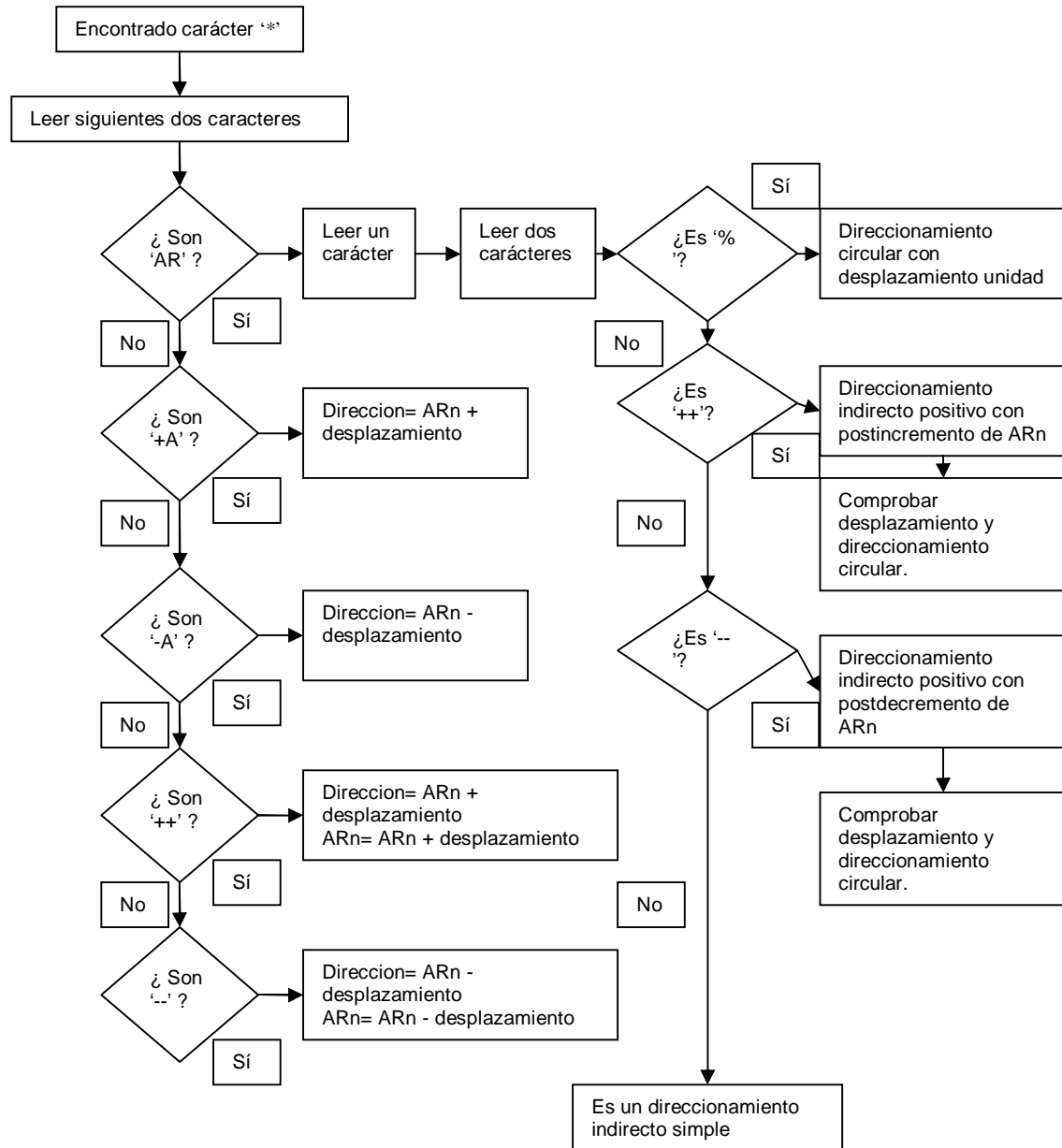
No se van a explicar en este apartado los mecanismos de los modos de direccionamiento porque ya se han comentado en la sección 2.2.6. A continuación se explicará el algoritmo de reconocimiento de operandos que utiliza el simulador, el cual interpreta el modo de direccionamiento utilizado en cada momento.



**Figura 3.7. Modo de interpretación de los direccionamientos**

En la figura 3.7 se muestra cómo el simulador interpreta los modos de direccionamiento cuando se lee una instrucción. El direccionamiento relativo al PC se implementa si la instrucción lo admite y se leen los operandos adecuados para este direccionamiento (etiqueta o dirección).

Para el caso del direccionamiento indirecto, se profundizará un poco más en el algoritmo:



**Figura 3.8. Interpretación del direccionamiento indirecto**

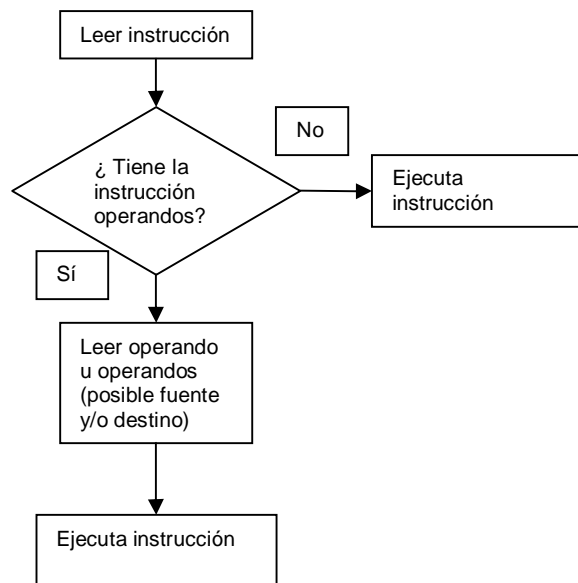
La figura 3.8 muestra cómo se interpreta el direccionamiento indirecto según los caracteres que se van leyendo. Cuando se utiliza el direccionamiento circular es necesario que previamente se haya cargado un valor positivo en el registro BK, de lo contrario, habrá un error. El direccionamiento a bit invertido no se ha implementado.

### 3.5. SUMARIO DE INSTRUCCIONES Y EL FLUJO DE CONTROL DEL PROGRAMA.

Como ya se ha explicado en la sección 2.2.8. el sumario de instrucciones, no se va a profundizar más en el tema. Tan sólo se comentará el modo de funcionamiento de ciertas instrucciones y la manera de interpretarlas.

Básicamente se han implementado todas las instrucciones del ensamblador del DSP TMS320C30, excepto las siguientes: NORM, RND, IACK, SWI, SUBRB, SUBRF, SUBRI, TRAPcond y operaciones básicas con interbloqueo. Las interrupciones (excepto la del puerto serie) no se han realizado en el simulador, ni tampoco las operaciones con interbloqueo por no considerarlas necesarias para el objetivo final.

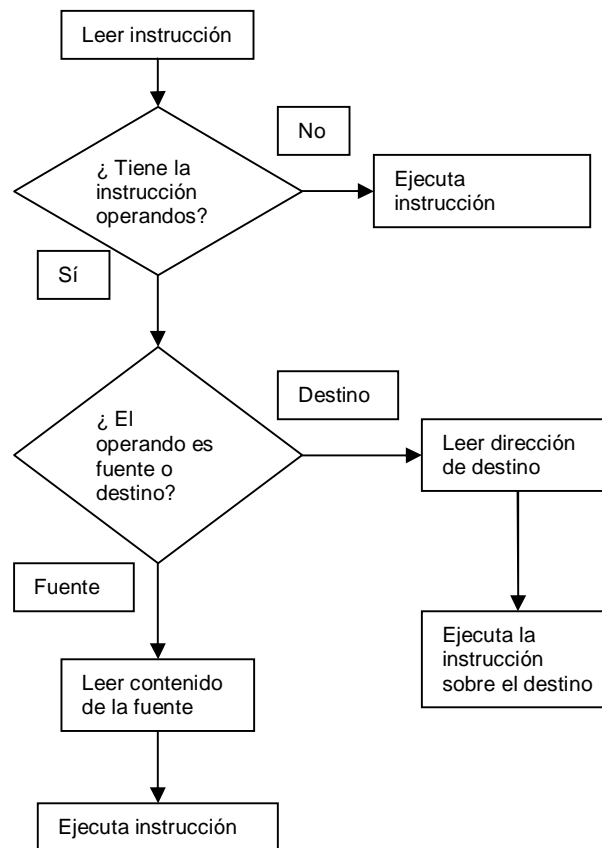
La funcionalidad de cada instrucción está descrita en la sección 2.2.8 y por ello no se entra en detalles, sólo se describe a continuación el modo de operar del simulador. Para una instrucción genérica, el comportamiento es el siguiente:



**Figura 3.9. Interpretación genérica de una instrucción**

En el caso de que la instrucción tenga un operando, el modo de operar se muestra en la figura 3.10.

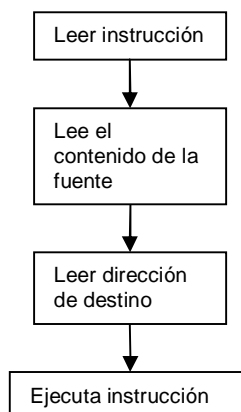




**Figura 3.10. Interpretación de instrucciones de un operando**

Entre las instrucciones de un operando se encuentran las siguientes: PUSH, POP, PUSHF, POPF, ROL, ROR, ROLC, RORC, CALL, RPTB, RPTS, IDLE, NOP, BR y Bcond.

En cuanto a las instrucciones de dos operandos, el modo de operación es:

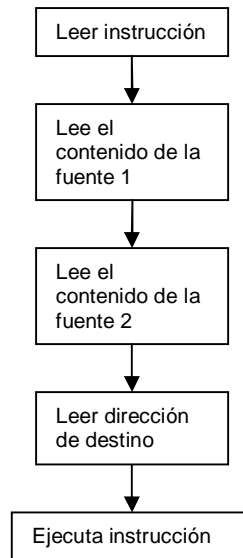


**Figura 3.11. Interpretación de instrucciones de dos operandos**

Entre las instrucciones de dos operandos se encuentran las siguientes: FLOAT, FIX (aunque estas dos admiten un solo operando también), CMP, CMPF3, CMPI, CMPI3,

TSTB3, ABSI, ABSF, ADDC, ADDF, ANDN, ASH, CMPF, CMPI, LSH, MPYF, MPYI, NEGI, NEGF, NOT, OR, ROL, ROLC, SUBC, SUBF, SUBI, SUBB, TSTB y XOR.

Las instrucciones de tres operandos pueden tener dos operandos fuente (o un solo operando fuente y un operando *cuenta*) y un operando destino. La forma de interpretarlas es la siguiente:

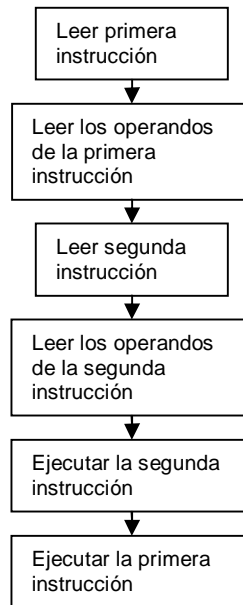


**Figura 3.12. Interpretación de instrucciones de tres operandos**

Las instrucciones de tres operandos son las siguientes: ADDC3, ADDF3, ADDI3, AND3, ANDN3, ASH3, CMPF3, CMPI3, LSH3, MPYF3, MPYI3, OR3, SUBB3, SUBI3, TSTB3 y XOR3.

Para instrucciones de ejecución paralela, el simulador las interpreta como una sola, siendo necesario primero leer los operandos correspondientes a las dos instrucciones antes de ejecutarlas. El procedimiento llevado a cabo es el que se comenta en la figura 3.13.

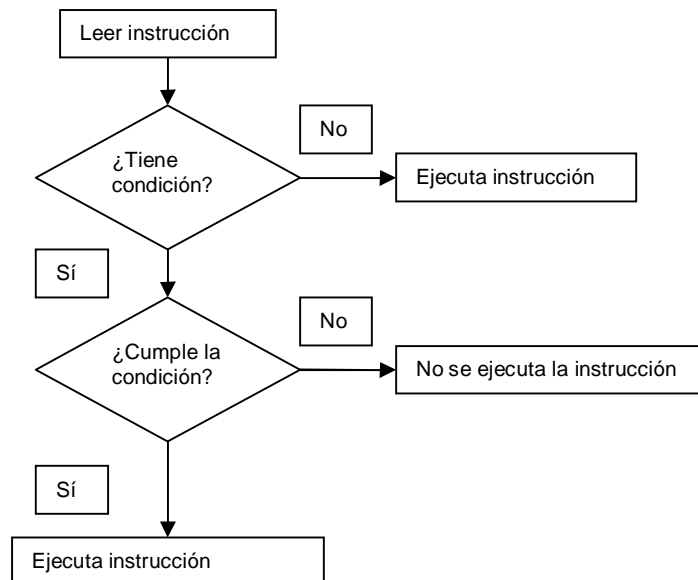
Las instrucciones de ejecución paralela se describieron completamente en la sección 2.2.8.



**Figura 3.13. Interpretación de instrucciones de ejecución paralela**

Se debe resaltar que con este planteamiento, el registro de estado ST (que modifica su contenido con cada instrucción), verá modificado su contenido con la primera instrucción de las dos en ejecución paralela. Esto es precisamente lo que se ha buscado, ya que así lo refleja el manual [Texas, 1].

En las instrucciones de ejecución condicional, el modo de operación es el siguiente:



**Figura 3.14. Interpretación instrucciones condicionales**

Las instrucciones de control de flujo fueron comentadas en la sección 2.2.7. Los saltos retardados no se han implementado, ni la instrucción TRAPcond.

La instrucción IDLE, que es la de salto a la interrupción del puerto serie 0, tiene un comportamiento similar al de la función CALL, excepto que la primera desactiva la bandera de habilitación global de interrupciones (GIE). Este comportamiento es diferente que en el DSP TMS320C30, ya que la instrucción IDLE hace que el sistema espere una interrupción (y pone GIE a 1), pero en el simulador saltará directamente a la rutina de atención de la interrupción, y por tanto, se comportará como una instrucción CALL.

El resto de instrucciones de control de flujo (BcondD, BRD, DBcondD, CALL, CALLcond, RETIcond, RETScond, RPTB y RPTS) realizan las funciones descritas en la sección 2.2.7.

## **3.6. LAS DIRECTIVAS DE ENSAMBLADOR Y LA UBICACIÓN EN MEMORIA.**

Como ya se ha comentado en la sección 2.5.2, el ensamblador y el linker del DSP TMS320C30 crean ficheros objeto que pueden ser ejecutados por el procesador. El formato en el que están estos ficheros objeto es el llamado COFF.

El COFF hace la programación modular más fácil, ya que hace pensar en términos de bloques de código y datos llamados secciones. Una sección es un bloque de código o datos que ocupa un espacio contiguo, en el mapa de memoria del C30. Cada sección de un fichero objeto está separado de las otras secciones del mismo fichero.

Los ficheros objeto COFF siempre contienen tres secciones por defecto:

- .text : Usualmente contiene código ejecutable.
- .data : Usualmente contiene datos inicializados
- .bss: Usualmente reserva espacio para variables inicializadas

En el simulador del DSP TMS320C30 no se han implementado todas las directivas de ensamblador existentes, solo las que se han considerado necesarias para la realización de prácticas de laboratorio. Las directivas implementadas son las siguientes:

- .usect : reserva el espacio en la sección adecuada, pero no cambia en qué sección están los datos o el código siguientes.
- .text : en esta sección (como ya se ha comentado), se ubica el código ejecutable.
- .sect : permite crear secciones con un nombre propio, creando el número de secciones que el programador cree oportuno.
- .space : palabras de 32bits. Reserva los bits especificados en la directiva, haciendo que la etiqueta que lo acompaña apunte al inicio del espacio reservado.
- .float : inicializa uno o más constantes en punto flotante de precisión simple de 32 bits.
- .long : inicializa uno o más enteros de 32 bits.
- .word : inicializa uno o más enteros de 16 bits.
- .set: iguala un valor a un símbolo.

Cuando se carga un programa en ensamblador del DSP TMS320C30, se realizan los siguientes pasos:

- 1) Ubicación y lectura de las secciones.
- 2) Numeración de las instrucciones del código según el contador de programa.

### ***3.6.1. UBICACIÓN Y LECTURA DE LAS SECCIONES***

El primer paso a seguir cuando cargamos un programa en ensamblador es ubicar las secciones en la memoria. Las partes de memoria posibles donde se pueden ubicar las secciones son: SRAM, RAM0 y RAM1. Por otro lado, las secciones de memoria a ubicar son las siguientes:

- .text
- .sect
- .stack (pila del sistema, en el simulador se reserva el espacio para la pila sin necesidad de utilizar la directiva).

En el caso de que se elija ubicar las tres secciones en una misma parte de memoria, el programa las colocará por este orden: primero .text, seguidamente .sect, luego se ubica la sección COM\_DATA (ver sección 2.2.10.1), y por último la pila del sistema.

### **3.6.1.1. Ubicación de la sección .text en memoria**

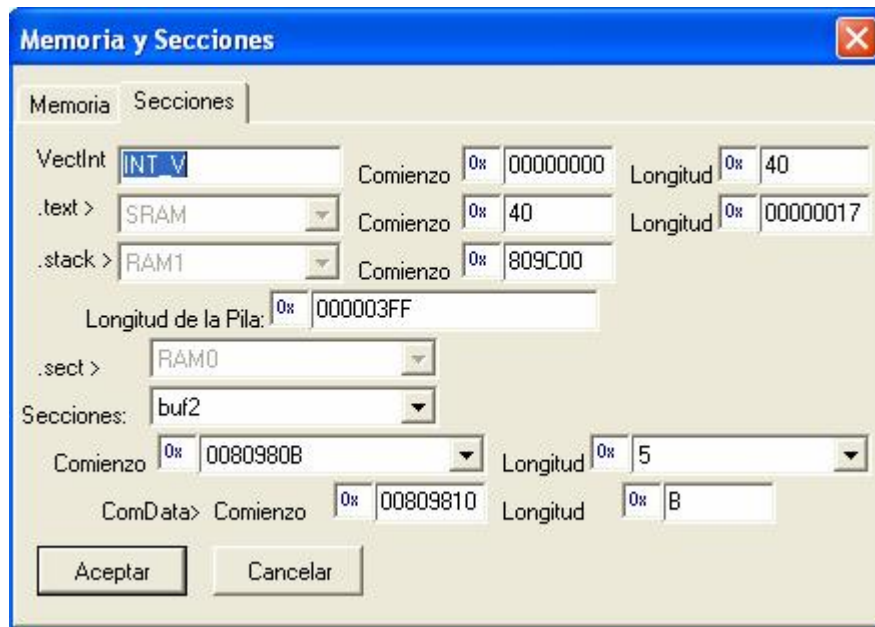
Cuando se carga un fichero en formato ensamblador en el simulador, la primera operación que éste realiza es contar el número de líneas de instrucciones de código que tiene el programa. Una vez contabilizadas, se reserva el espacio de memoria correspondiente al tamaño calculado y se marca el espacio ocupado para que el usuario lo visualice y no escriba en él. (Se marca cada posición de memoria con la etiqueta “EEEEEEEE”). Por otro lado, la dirección de comienzo de la sección y su tamaño aparecen reflejados después de la carga del fichero y la ubicación en memoria, en la ventana llamada “Memoria y Secciones”.

### **3.6.1.2. Ubicación de la sección .sect en memoria**

Como se comenta en el manual de usuario (capítulo 5), una de las reglas de utilización del simulador consiste en que las secciones .text y .sect deben estar claramente diferenciadas. El ensamblador del DSP permite la utilización de directivas de ensamblador para que usen espacio de memoria dentro de la sección .text. No va a ser este el caso, ya que es requisito imprescindible para el buen funcionamiento del simulador que primero se declaren las secciones y dentro de ellas todas las posibles constantes, variables o espacios reservados que se desee utilizar.

En caso de que la sección .sect coincida en la misma parte de la memoria que la sección .text (SRAM, RAM0 ó RAM1), ésta irá ubicada justamente después del espacio reservado por la sección .text.

Cada vez que se escriba una directiva .sect, se crea una nueva sección de memoria contigua a la que hubiera antes. Una vez cargado el fichero ensamblador y ubicadas las secciones en memoria, en la opción del menú “Memoria y Secciones”, aparecen reflejadas todas las secciones, su dirección de comienzo y longitud. Todo esto puede verse en la figura 3.15.



**Figura 3.15. Ventana de memoria y secciones.**

La forma de ubicar en memoria es la siguiente, según la directiva de ensamblador que empleemos después de utilizar la directiva `.sect`:

- Directiva `.space`: reserva espacio en la memoria. El espacio reservado estará a cero.
- Directiva `.word`: inicializa uno o más enteros. En este caso, se tienen dos opciones:
  - Si el contenido de la directiva `.word` es uno o varios valores enteros, estos valores se irán almacenando en memoria contiguamente.
  - Si el contenido de la directiva `.word` es una etiqueta de una directiva de ensamblador previamente declarada, se almacena en memoria la dirección de comienzo de la directiva previa. Ejemplo:

```
buf2 .word 1,4,6,4,1
```

```
input2 .word buf2
```

En este caso, en posiciones de memoria contiguas se escriben los valores enteros 1, 4, 6, 4 y 1 seguidos de otra posición de memoria que contenga la dirección de comienzo de `buf2`, es decir, la dirección correspondiente al primer '1' entero escrito.

- Directiva `.long`: el tratamiento es el mismo que para la directiva `.word`.
- Directiva `.float`: inicializa uno o más números en punto flotante.

- Directiva .usect: tiene el mismo efecto que si se declara una nueva sección y luego se utiliza la directiva .space.
- Directiva .set: reemplaza una etiqueta en el código por otro valor definido.

Cada sección .sect o .usect definida antes de la directiva .text se ubica en memoria contigua a la anterior sección. La diferencia con el linker del DSP TMS320C30 es que éste permite ubicar cada sección en una posición diferente de memoria, mientras que el simulador las coloca automáticamente en posiciones contiguas.

Una vez cargado el programa, es posible consultar la dirección de inicio y longitud de cada sección en la opción “Memoria y Secciones”. Además, es posible consultar también la correcta ubicación de las secciones directamente sobre la ventana de memoria.

En cuanto a la sección COM\_DATA, el programa la ubica automáticamente después de las secciones declaradas.

### **3.6.1.3. Ubicación de la pila del sistema en memoria**

Cuando se carga el programa en ensamblador, se pide introducir la ubicación de la pila del sistema (stack). Si se elige una zona de memoria (SRAM, RAM0 ó RAM1) donde también se encuentren las secciones .text y/o .sect, entonces la pila del sistema se sitúa contigua a estas secciones pero siempre al final.

Además, también se pide introducir la longitud de la pila del sistema, pudiendo haber dos posibles errores al introducirla:

- 1) Si la longitud de la pila es mayor que la longitud física de memoria, el programa visualiza un mensaje de error, teniendo que volver a introducir una longitud correcta.
- 2) Si la longitud de la pila es mayor que la longitud disponible después de ubicar las secciones .text y .sect (si las hubiere en esa zona de memoria), entonces automáticamente se reduce la longitud de la pila al valor máximo posible de los restantes.

Si en algún momento de la ejecución del programa, se excede los límites de la pila del sistema, recibiremos un mensaje de error de desbordamiento.



### 3.6.2. NUMERACIÓN DEL CÓDIGO

Una vez cargado el programa y ubicadas las secciones en memoria, se procede a numerar el código. Como es conocida la dirección de comienzo de la sección .text y su longitud, se debe identificar de nuevo cada instrucción (antes ya se hizo para contar la longitud de .text). Por cada instrucción encontrada, se asigna un número de línea correspondiente al valor del PC cuando se ejecute esta instrucción. Este número de línea puede verse en una columna contigua a la ventana de ejecución (ventana “Valor del PC”). Durante la ejecución, el valor del PC se modifica con el valor de línea correspondiente a la instrucción a ejecutar.

### 3.7. ENTRADA Y SALIDA DE INFORMACIÓN

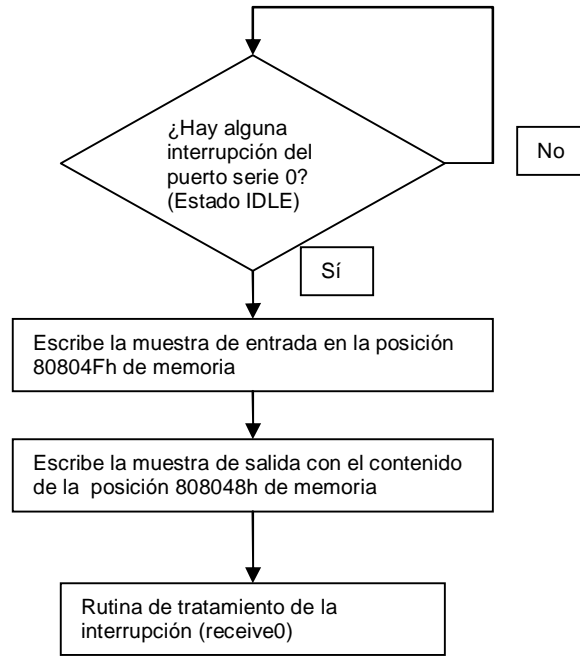
En esta sección sólo se profundizará en detalles del funcionamiento del simulador atendiendo a la entrada y salida de información. En las secciones 2.2.9. y 2.2.10. se explica con detalle el funcionamiento de la entrada y salida de información en el DSP TMS320C30 EVM.

Como se comenta en la sección 2.2.9.1, cada periférico puerto serie posee ocho registros de 32 bits, ubicados en una zona determinada del mapa de memoria del DSP, que pueden ser accedidos por software en lectura o escritura. En el caso del simulador sólo se utiliza el puerto serie 0, y sus registros están mapeados en memoria desde la posición 808040h hasta la posición 80804Fh. De todos estos registros interesa especialmente el registro de datos en recepción (DRR), ubicado en la posición 80804Ch y el registro de datos en transmisión (DXR), ubicado en la posición 808048h.

Cada vez que se cargue un programa en ensamblador, se pide la siguiente información:

- Frecuencia del conversor analógico-digital AIC (Hz): Es uno de los valores de la tabla 2.17. de la sección 2.2.10.3.
- Tiempo de simulación (segundos): Es el tiempo de simulación en segundos.
- Ficheros de entrada y salida: contienen la información de entrada y salida del DSP.

Esta información determina el número de interrupciones del puerto serie 0 que se realizan. Cada vez que se atiende a la interrupción (bloque de código ‘receive0’), se lee la muestra de entrada (posición 80804Fh de memoria, registro DRR) y se saca a la salida el contenido de la posición 808048h (registro DXR). El procedimiento es el siguiente:



**Figura 3.16. Interrupción del puerto serie 0.**

Es evidente que si en el programa ensamblador no hay rutina de atención a la interrupción, no se leerá nada de la entrada ni se escribirá en la salida.

Siguiendo lo descrito en la sección 2.2.10.3 sobre el control del AIC, se debe enviar a la salida un número entero de 14 bits. Por tanto, si en la posición 808048h hay escrito un valor entero mayor a 14 bits en el momento de la interrupción, se envía a la salida el mayor entero con signo posible de 14 bits (será 8092 en valor absoluto). En cuanto a la entrada, cada vez que hay una interrupción se escribe el valor de entrada (entero de 14 bits) en la posición 808048h.

### ***3.7.1. ESTUDIO TEÓRICO DE LA ENTRADA Y SALIDA.***

Debido a que a la entrada del sistema se tiene un fichero de muestras de enteros, se debe dar a este fichero un formato para que represente una señal deseada. Ya que no se cuenta con una fuente analógica, se supone que las muestras del fichero de entrada corresponden a la salida digital sobre un muestreo de la señal analógica de 17361 muestras por segundo, que corresponde con la frecuencia máxima de muestreo del ADC (*Analog-Digital Converter*).

Por tanto, según se elija la frecuencia de muestreo del ADC, se toman más o menos muestras:

- Si la frecuencia del ADC  $f = 17361$  Hz, se utilizan todas las muestras de entrada.
- Si  $f = 11574$  Hz, se toman menos muestras. En concreto ( $17361/11574 = 1,5 = 3/2$ ) se cogen dos de cada tres muestras del fichero.
- Si  $f = 8013$  Hz,  $17361/8013 \approx 2,2 = 11/5$ , se toman cinco de cada once muestras de entrada.
- Si  $f = 4006$  Hz,  $17361/4006 \approx 4,33 = 26/6$ , se toman seis de cada veintiséis muestras.
- Si  $f = 3360$  Hz,  $17361/3360 \approx 5,12 \approx 41/8$ , se toman ocho de cada cuarenta y una muestras.

Dada una señal de entrada senoidal a una frecuencia determinada, como la frecuencia de muestreo es de 17361 Hz, el número de muestras por periodo es:

$$\text{Número de muestras por periodo} = \frac{T_{\text{SEÑAL}}}{T_{\text{MUESTREO}}}$$

### Ecuación 3.6

$T_{\text{SEÑAL}}$  representa el periodo de la señal analógica de entrada, mientras que  $T_{\text{MUESTREO}}$  es la frecuencia máxima de muestreo del ADC, o sea, 17361 Hz. Para que haya una buena señal de salida se debe cumplir el teorema del muestreo (muestrear como mínimo al doble de velocidad que la frecuencia de la señal muestreada).

Ejemplo: se quiere generar una señal de entrada de frecuencia 500 Hz muestreada a 17361 Hz. El primer paso es conocer el número de muestras por periodo necesario:

$$N = \frac{1/500}{1/17361} = 34,772$$

Aproximamos el resultado a 35 muestras por periodo.

En el programa Matlab, se introduce lo siguiente:

```
x = [0:2*pi/34:2*pi];
y = sin(x);
plot(x,y);
```

Con esto se consiguen muestras equivalentes a un periodo de la señal senoidal de 500KHz muestreada a 17384 Hz. También se sabe que estas 35 muestras corresponden a  $(1/500\text{Hz})=0,002$  segundos.

Por tanto, conocidas el número de muestras que se tienen por periodo de señal, se multiplica este número de muestras por el tiempo de ejecución dividido entre el periodo. Así se obtiene el número de muestras requerido para el intervalo de tiempo elegido de ejecución:

$$\text{Nº de muestras totales} = \text{Nº de muestras por período} * \frac{\text{Tiempo de ejecución( seg)}}{\text{Duración de un período (seg)}}$$

**Ecuación 3.7**

En cuanto a cuantización de las muestras de salida, se supone que a la salida, el TMS320C30 EVM tiene un margen dinámico de la señal de salida de  $\pm 3$  Voltios (es un rango de salida posible de varios que se pueden obtener modificando el divisor de tensión de salida amplificador de salida). Para más detalles, consultar [Texas, 3].

Para representar correctamente los valores de la señal de salida, se debe multiplicar el número entero de salida (que representa una muestra) por un factor:

$$\text{Factor} = \frac{\text{Rango dinámico posterior}}{\text{Rango dinámico anterior}} = \frac{3 - (-3)}{8192 - (-8192)} = 0,0003662109375$$

**Ecuación 3.8**

El rango dinámico anterior es la diferencia entre los máximos valores enteros con signo representables de 14 bits (máximo positivo y máximo negativo).

## 3.8. LA EJECUCIÓN DEL PROGRAMA

Una vez cargado el programa, se tienen varios modos de ejecución, todos ellos limitados al tiempo de ejecución previamente elegido. Los modos de ejecución son los siguientes:

- Ejecución continua: En este modo el programa se ejecuta hasta que termina el tiempo de ejecución.
- Ejecución paso a paso: Es igual que el modo de ejecución continua, solo que se ejecuta una sola instrucción cada vez.

- Ejecución de bloque: Igual que la ejecución paso a paso excepto que los saltos a bloques de código se hace en ejecución continua.
- Ejecución hasta el cursor: Se realizará una ejecución continua hasta la instrucción donde se encuentra el cursor fijado.