Al pie de la letra

Prosiguiendo nuestro estudio de las instrucciones del 68000 de Motorola, consideraremos la instrucción de comparación

Estrechamente relacionada con la instrucción SUB tenemos la instrucción CMP (comparación), cuya importancia estriba en que, sea cual sea el área de programa en que estemos trabajando, siempre habrá elementos de decisión en el diseño de ese programa. Consideremos el siguiente diseño en alto nivel, que incluye un elemento *pseudo-code* de decisión:

```
If inputchar='N' then
   cleararray
end
```

Aquí la decisión If se codificará así en la fase de implementación del proyecto:

```
CMP.B  #'N',D0     Compara el byte de entrada
                   con N
BNE    NOTSAME     Va a NOTSAME si es
                   diferente
```

Cuando se ejecuta la instrucción CMP, la fuente es restada del destino y se cambian sólo los códigos de condición, sin que sea afectado el destino (como ocurre, por ejemplo, con la instrucción SUB). La bifurcación condicional, BNE, que sigue provocará una bifurcación a la etiqueta NOTSAME si el resultado de la recta de D0-'N' no es cero.

Veamos otros dos ejemplos que emplean atributos diferentes de datos:

```
CMP.W SPEED,D3   Compara contenido palabra de
                 posición SPEED con D3
CMP.L D1,D2      Compara pals. largas completas
                 contenidas en D1 y D2
```

Se notará que los campos de destino en estos ejemplos son registros de datos, y que de hecho puede utilizarse cualquier modo de direccionamiento como modo de direccionamiento fuente. Si es necesario direccionar cualquier otra forma de destino, se usan diferentes instrucciones CMP. Por ejemplo:

```
CMPA   BETTY,A3   Compara el contenido de la
                  posición BETTY con A3
```

En realidad es posible el uso de cualquier modo de direccionamiento fuente. Sin embargo, la forma inmediata que sigue sólo permite modos alterables de datos.

```
CMPI   #3,(A4)    Compara cualquier A4 que se
                  apunte con 3
```

Una última variante de CMP digna de mención es la instrucción CMPM. Por ejemplo, CMPM (A2)+,(A3)2 comparará el contenido de las posiciones de memoria apuntadas por A2 y A3 y después posincrementará los punteros. Por ello, esta instrucción puede ser utilizada para comparar, por ejemplo, palabras clave almacenadas con caracteres que están en un buffer de entrada, y todo eso también en una palabra. He aquí un ejemplo:

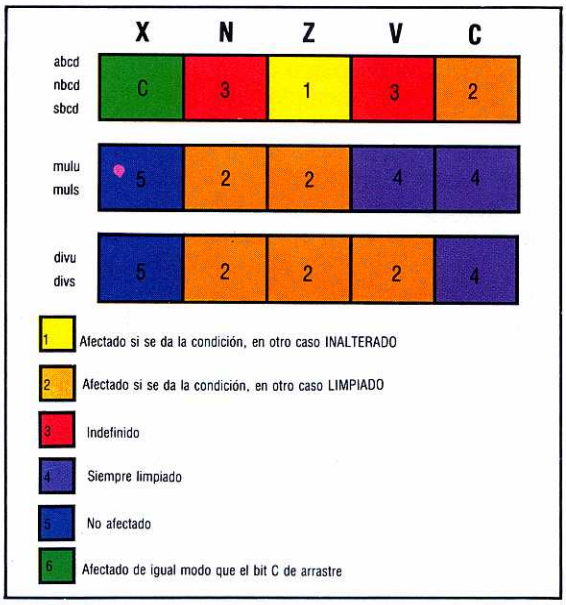
```
LEA     KEYS,A2      Establece el primer puntero
                     hacia las palabras claves
LEA     BUFFER,A3    Y el puntero del buffer
CHECK   CMPM.B       Compara las dos posiciones de
        (A2)+,       memoria
        (A3)+
BEQ     CHECK        Va comparando hasta que
                     sean diferentes
```

El programa en total ocupa sólo siete palabras.

Otro aspecto ulterior es que la instrucción CMPM sólo se permite con el modo postincremento de direccionamiento; así, si se desea cualquier otra forma, entonces un operando ha de ser cargado en el registro de datos en primer lugar y emplear la instrucción CMP.

Dos sencillas instrucciones aritméticas son NEG y EXT. La instrucción NEG (negación) resta de 0 el registro de datos del operando, es decir, forma el valor negativo en complemento a dos del contenido del registro de datos (no se permite ningún otro modo de direccionamiento). Así, por ejemplo, si D0=1111 1010 (−6, en decimal) y ejecutamos NEG.B D0, ocurrirá que D0 contendrá 0000 0110 (6, en decimal). Empleos típicos de esta instrucción incluirán la obtención del valor absoluto de un operando de datos comprobando primeramente el valor negativo y después dándole esa forma negativa. Existe también una forma ampliada de la instrucción, que incluye el bit X, llamada NEGX.

La instrucción EXT se emplea para ampliar con el bit signo del operando de datos el tamaño del operando más grande. Así, si se ejecuta EXT.W D0



Cambio de estado
Los efectos de las instrucciones DIV, MUL y BCD sobre el registro de estado son los que aquí se ilustran. Es importante observar que en algunos casos la ausencia de una condición no se traduce en la limpieza del flag correspondiente, que quedará inalterado y, si se emplea de testio, puede dar una falsa lectura (la que corresponde a una operación previa)

donde D0=1111 0101 (FA, en hexa) dará D0=FFFA (en hexa). Esta instrucción es útil cuando se trabaja con números de precisión múltiple, en especial cuando se incluyen operandos más grandes de datos, como pueden ser las instrucciones de multiplicación y división.

Interpretación de patrones de bits

Antes de seguir con el examen de instrucciones más complicadas, debemos considerar el empleo de los modelos binarios de bits en un operando de datos de un byte. Por ejemplo, cuando D0=1001 0110 podemos suponer que se trata de un entero con un valor de –106 (recuerde que el valor se obtiene haciendo una inversión y sumando la unidad). Sin embargo, si el número no tiene signo y se asume el intervalo global binario con enteros positivos, el valor sería 96 (en hexa), o sea 150 en decimal (9×16+6×1). Los números sin signo son útiles si sabemos que sólo vamos a necesitar un intervalo grande positivo. Por ejemplo, el intervalo de direcciones de un ordenador puede considerarse como un intervalo de números positivos sin signo, y a condición de que no operemos con esos números mediante operadores con complemento a dos, así pueden tomarse.

Pero hay todavía una tercera interpretación del modelo de bits dado; es el sistema llamado BCD (*binary coded decimal*: decimal codificado en binario). Se trata de una codificación muy adecuada en la que todo grupo de cuatro dígitos binarios es considerado como el código de un dígito decimal. En este caso, nuestro ejemplo (D0=1001 0110) tendría en BCD el valor de 96 (1001=9 y 0110=6).

Debemos hacer tres importantes observaciones sobre esta codificación. Primera, lo fácil que es convertir números incluso muy grandes de decimal a BCD, o viceversa. Por ejemplo, el decimal 9 631 en BCD se escribe 1001 0110 0011 0001. La conversión, como se ve, es muy sencilla.

La segunda observación es que también resulta fácil definir la precisión de los números codificados de esta manera. La tercera se refiere a que codificamos dígitos del 0 al 9, por lo que los códigos sobrantes no tienen validez (es decir, los códigos que van desde el 1010, que es 10 en decimal, al 1111, que es 15 en decimal).

La importancia de estas observaciones se hará evidente cuando estudiemos las distintas instrucciones aritméticas del 68000. Veámosla por un momento en la operación de multiplicar. Si multiplicamos dos operandos de 16 bits, el modelo de bits resultante puede tener 32 bits de longitud. Usted mismo puede comprobarlo, pero entienda que para una palabra de n bits de longitud, el producto del número más grande, $2^{(n-1)}$, será $2^{2(n-1)}$, o sea, dos veces la longitud de la palabra original.

Para la instrucción de multiplicar en binario, disponemos por ello de dos operandos de 16 bits que dan como resultado 32 bits. Puesto que podemos operar con números con y sin signo, tenemos dos instrucciones independientes, la MULU (*multiply unsigned*: multiplicación sin signo) y la MULS (*multiply signed*: multiplicación con signo). Ambas instrucciones multiplican los dos operandos y ponen un resultado de 32 bits en el registro de datos de destino. De nuevo se observará que sólo se permiten modos de direccionamiento de datos para el operando fuente.

Así, por ejemplo, MULU #20,D0 cuando D0=XXXX 0003 (la X significa aquí 0 o 1) dará D0=0000 003C. Se notará que se emplea el registro entero de datos de 32 bits aun cuando no sea necesario en este ejemplo particular.

Del mismo modo, cuando D0=XXXX FFFF, MULU #$10,D0 dará como resultado D0=000F FFF0 pero con MULS se obtendría D0=FFFF FFF0. Esto es así porque el resultado es de signo ampliado a todos los 32 bits. Se observará que es posible comprobar fácilmente estos resultados dado que el multiplicador de 10 hexa equivale a un desplazamiento a la izquierda de 4, o a una multiplicación por 16.

Una consideración final sobre la instrucción para multiplicar se refiere a los códigos de condición. Los flags N y Z se activan según sea el resultado, pero los bits V y C son puestos a 0. Aunque no es posible que con operandos de palabras y resultados de una palabra de longitud obtengamos un desbordamiento, es interesante conocer si el resultado de palabra se desbordaría, porque entonces se sabrá fácilmente si el resultado puede ser truncado a una palabra en caso de necesidad.

Veamos un fragmento típico de programa que emplea la instrucción MULS. Supongamos que se desea convertir caracteres ASCII representativos de números decimales en palabras binarias. La primera operación será convertir el carácter de entrada en binario y sumar seguidamente este patrón de bits en el acumulador binario. Antes de efectuar la suma tenemos que asegurarnos de que las entradas anteriores fueron multiplicadas por 10 (pues se trata de entradas decimales). Así, por ejemplo, un posible programa sería éste:

```
SUB.B  #'0',D0    Forma el binario del carácter decimal
MULS   #10,D5     Multiplica la suma previa por 10 decimal
ADD.W  D0,D5      Suma el nuevo valor
```

En este ejemplo el carácter de entrada está en D0 y la nueva palabra binaria que representa en binario los caracteres de entrada está en D5.

Examinemos ahora la instrucción de dividir, DIVU (para números sin signo) y DIVS (para números con signo). Ambas instrucciones toman el entero de 32 bits que está en el registro de datos de destino y lo dividen por el operando fuente de 16 bits. El resultado se pone en el registro de datos de destino y en los 16 bits inferiores colocando cualquier resto en los 16 bits más significativos. Por ejemplo, si D0=0000 0005 (en hexa):

```
DIVU   #3,D0
```

dará D0=0002 0001 (1 con resto 2). Observe que el operando destino es un operando completo de 32 bits, lo que significa que primero puede necesitarse una palabra larga simple, o una larga EXT.L ampliada con signo. Dado que es posible el desbordamiento (p. ej., dividiendo por la unidad una palabra entera de 32 bits obtendremos como resultado algo más de 16 bits), se tendrá en cuenta oportunamente el bit de desbordamiento.

Un sencillo algoritmo que nos permita contar todos los números de una lista de palabras terminados en cero podría ser el siguiente:

```
                ORG $1000
        LEA     LIST1,A0
PUNTERO                         establece
                                  dirección
        CLR.L   D0              borrado total
        CLR.L   D1              contador bucle
LOOP    ADDQ    #1,D1           cuenta núm.
                                  palabras
        ADD.W   (A0)+,D0        y suma
        TST     (A0)            comprueba
                                  fin de lista
        BNE     LOOP
        DIVU    D1,D0           divide suma
                                  por contador
        TRAP    0
LIST1   DC.W    1,2,3,4,0
```

Se notará que se han empleado borrados de largas palabras en la parte de inicialización de este programa, ya que la suma será tratada como operando completo de 32 bits. En D1 se coloca un contador y la palabra apuntada A0 se añade a D0 con el direccionamiento de postincremento. Seguidamente se comprueba por medio de la instrucción TST la siguiente palabra que está en LIST1. Esta instrucción se limita a colocar códigos de condición listos para la instrucción BNE, y no altera operando alguno.

La instrucción TST es necesaria porque la instrucción anterior ADD afectará los códigos de condición según sea el resultado de sumar los operandos de datos en D0, y no según el valor de los datos cargados por el puntero A0. La instrucción BNE (B*ranch if Not Equal to zero:* bifurcar si no es igual a cero) provocará la vuelta a LOOP mientras no sea cero el siguiente elemento de la lista. Cuando ocurra que es igual a cero, D0 contendrá la suma y D1 el número de elementos no cero (el contador de datos).

Cuando se haya ejecutado el bucle (LOOP) cuatro veces, los valores contenidos en los registros de datos serán:

```
D0=0000 000A (en decimal,10)
D1=0000 0004
```

Por tanto, una vez ejecutada la instrucción DIVU, entonces D0 contendrá 0002 0002 (10 ÷4=2 y me llevo 2).

Una última observación sobre las instrucciones para dividir es que una división por cero provocará un *trap* (una interrupción software en el monitor del sistema), ya que un número infinito ciertamente no es representable en 16 bits. Si no queremos caer en esta "trampa", hemos de establecer un testigo cuando el divisor sea cero.

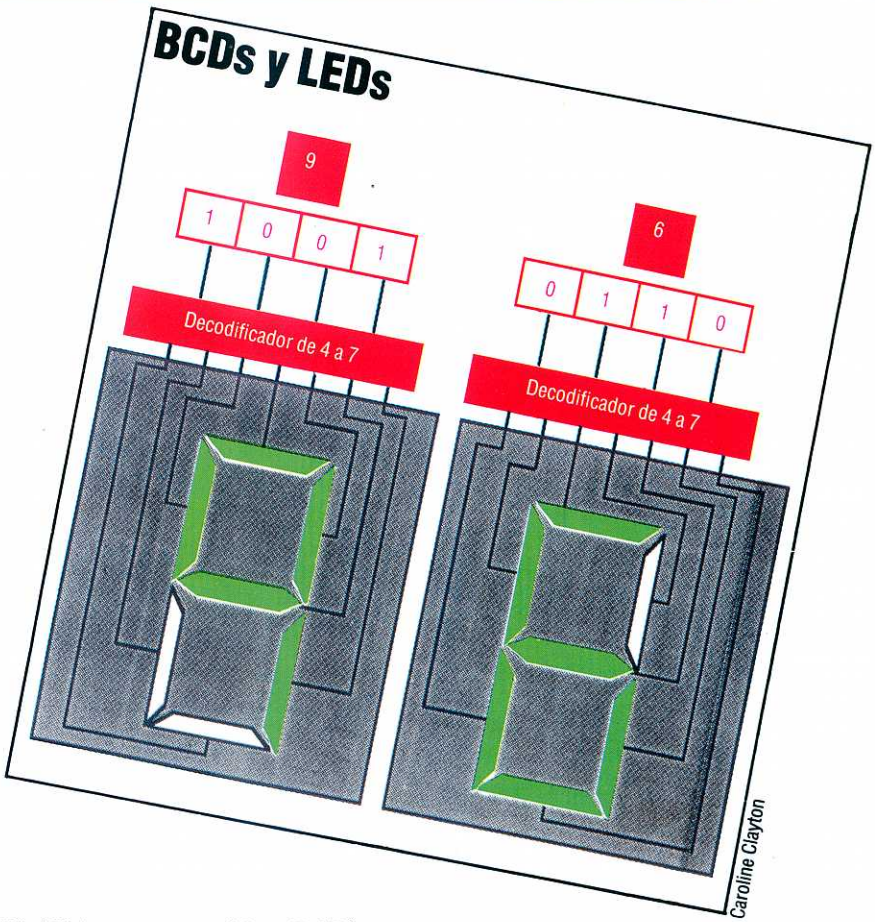
Por ejemplo:

```
TST     D1
BEQ     ERROR
DIVU    D1,D0
```

Ya hemos visto la conveniencia del BCD para la representación de números decimales. Pero se ha de notar que un dígito BCD válido corresponde a un dígito en hexa (4 = 0100 binario = 4 hexa = 4 BCD), por lo que las constantes BCD equivalen a constantes hexa. Por ejemplo:

```
MOVE.B      #$54,D0   pone 54 BCD en D0
```

Pero la analogía acaba aquí. Al sumar dos dígitos BCD en aritmética binaria, por ejemplo, obtendremos una respuesta incorrecta:




```
0100 1001       (49 en BCD) suma
0000 0001       binaria
                (1 en BCD)
---------
0100 1010
```

Obtenemos un dígito BCD ilegal como dígito BCD menos significativo. Esto no debe preocuparnos, sin embargo, dado que el 68000 posee un grupo de instrucciones BCD para sumar (ABCD), restar (SBCD) y negar (NBCD).

Nuestro estudio de las instrucciones BCD se limitará a la instrucción ABCD, que suma el byte fuente (dos dígitos BCD) al byte destino, con el bit X, poniendo la suma en el destino. Los únicos modos de direccionamiento permitidos son los pares de registros de datos y los pares de registros de direcciones predecrementadas. Así, por ejemplo, si D0=44 BCD y D1=01 BCD, entonces tras ejecutar

```
ABCD    D0,D1
```

D1 contendrá 45 BCD. Si el bit X que está en el SR estaba activado (equivalente al arrastre para BCD), el resultado sería 46. Igualmente, si sumamos 1 a 99 en D0, el resultado sería 00 con el bit activado en el registro de estado. Con esto se puede colegir que si bien estamos limitados a operandos de un byte, podemos ampliar fácilmente la precisión de nuestros cálculos empleando el bit X para llevar el arrastre a componentes del byte más significativo.

Los códigos de condición X, C y Z se activan para todas las instrucciones BCD. Sin embargo, es de notar que la instrucción NBCD (Negar en BCD) permite modos de destino alterables de datos preferentemente a pares predecrementados o simples datos.

Precisión BCD
Lo mismo que para operar con enteros con y sin signo, el 68000 posee un juego de instrucciones para la manipulación de datos tomados en forma BCD (decimal codificado en binario). Esto es especialmente útil cuando se necesita una precisión aritmética total. Pero también ofrece otras ventajas en distintas aplicaciones. Por ejemplo, en el manejo de visualizaciones LED de siete segmentos. El hardware que decodifica los datos en BCD emplea un decodificador de líneas de 4 por 7, como en el dibujo, resultando más sencillo que la decodificación de números binarios