Rendimiento de los comandos

Nos corresponde analizar las instrucciones lógicas junto con las operaciones de rotación y desplazamiento y las instrucciones de bifurcación

Analicemos primeramente las instrucciones lógicas de que está provisto el 68000. La instrucción AND opera con un "Y" lógico el fuente con el destino, dejando el resultado en el destino y afectando en consonancia a los bits N y Z del registro de estado. Existen varios modos de direccionamiento posibles, pero el fuente o el destino por lo menos deben ser un registro de datos. Además se permiten los atributos de datos; así, por ejemplo, si D0=1010 1010 y D1=1111 0000, y ejecutamos la instrucción:

```
AND D1,D0
```

entonces resulta D0=1010 0000.

Éste es un sencillo ejemplo del empleo de una instrucción AND. En este caso, hemos tomado los cuatro bits menos significativos poniendo a cero los bits de máscara que están en D1, y hemos hecho que los MSB (bits más significativos) aparezcan en su estado original mediante una máscara de bits puestos a 1. Una versión más explícita de esta operación incluiría quizá la instrucción ANDI. Ésta acepta el modo de direccionamiento inmediato como operando fuente, y los modos alterables de datos como destino. Si empleamos ANDI en el ejemplo anterior tendríamos:

```
ANDI #$F0,D0
```

Las restantes instrucciones lógicas son OR y ORI para la operación OR lógica, EOR y EORI para la OR exclusiva, y la instrucción NOT. Estas instrucciones, respecto a los modos de direccionamiento, son parecidas a la instrucción AND, siendo idéntica la forma en que es afectado el código de condición. La manipulación de los bits con instrucciones lógicas es de gran importancia para controlar, por ejemplo, los bits que hacen de *flags*, o bien una palabra o las líneas de entrada/salida digitales para periféricos o equipos externos.

Cuando el operando de datos es un bit único, el 68000 tiene un conjunto de instrucciones para manipulación de cuatro bits que pueden ser utilizadas en lugar de las instrucciones lógicas. Estas instrucciones comprueban el estado de un bit específico (numerado de 0 a 31, en un registro de datos, sólo para palabras largas) o un byte de la memoria. Afectan también al bit Z del SR según sea el estado de ese bit. Por tanto, el bit Z se convierte en una memoria invertida de un bit respecto al bit especificado. La instrucción BTST puede usarse del modo siguiente. Si, por ejemplo, D0=XXXX XXXX XXX1 0000 (en binario, siendo X un 0 o un 1), entonces:

```
BTST #4,D0
```

comprobará el bit cuatro y pondrá Z a cero (el bit del ejemplo no es cero). Ahora bien:

```
BTST #3,D0
```

pondría Z a uno. Todos los códigos de condición permanecen inalterados.

Las restantes instrucciones afectan al bit que se comprueba. Y son:

```
BSET    comprueba y pone el bit a 1
BCLR    comprueba y pone el bit a 0
BCHG    comprueba y cambia el valor del bit
```

Así si ejecutamos:

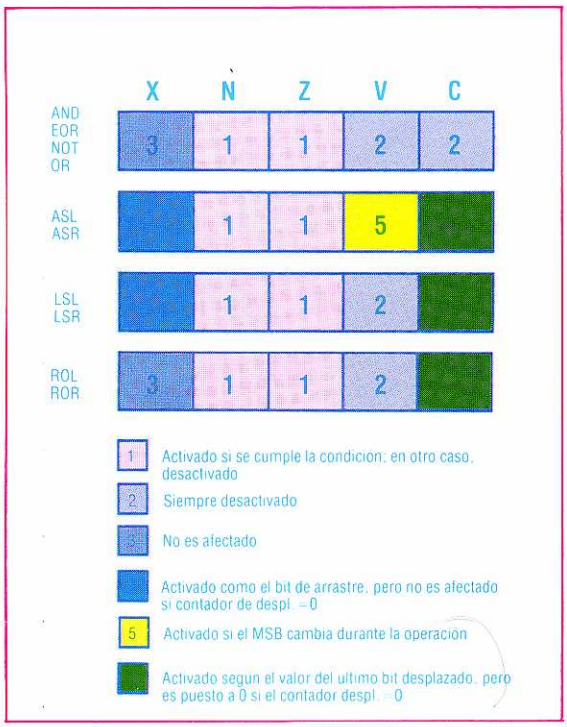
```
BCHG    #4,D0
```

en el ejemplo anterior (donde D0 está puesto a 1) entonces D0 se convertiría en D0=XXXX XXXX XXX0 0000 y Z se pone a 0, que indica el estado antes del cambio.

Es de notar que todas estas instrucciones realizan las operaciones de comprobación y alteración de bits en una sola instrucción. Esto puede ser importante en un entorno de multiprogramación, donde la posibilidad de ser interrumpido entre dos instrucciones puede conducir a resultados impredecibles. Una observación final sobre las instrucciones lógicas es que aún si se aplica la comprobación, no es necesario realizar acción alguna a partir de esa información. Se puede emplear la instrucción como manipulador de bits.

Ante todo, veamos lo que sucede cuando un patrón de bits es desplazado a la izquierda o a la derecha. Si D0 contiene 0000 0000 0000 1000, el desplazamiento de este modelo tres lugares a la derecha dará 0000 0000 0000 0001. El contenido de D0 ha sido dividido por ocho (o sea, 2^3), en otras palabras, un desplazamiento hacia la derecha corresponde a una división por dos por cada lugar que se desplace a la derecha. Además, se sigue de esto

Estado al día
Las instrucciones lógicas, de desplazamiento y de rotación afectan el contenido del registro de estado de la manera que aquí se ilustra. Las instrucciones de desplazamiento aritmético difieren de los desplazamientos lógicos sólo en el empleo que hacen del flag V para indicar cambios posibles en el bit de signo del operando



que un desplazamiento a la izquierda equivale a una multiplicación por idéntica cantidad.

Obsérvese que, en nuestro ejemplo, hemos supuesto que se rellenan con ceros los nuevos dígitos binarios (por la izquierda en los desplazamientos a la derecha, y viceversa). Esto se deberá cambiar si deseamos conservar el signo de nuestro número cuando hacemos un desplazamiento a la derecha. Así, por ejemplo, si D0=1111 1111 1111 0000 (−16, en decimal) un desplazamiento a la derecha de tres lugares daría D0=1111 1111 1111 1110, que es −2 en decimal. Aquí hemos puesto unos por la derecha para poder mantener el signo negativo del número. En general, podemos decir que para desplazamientos a la derecha el signo se conserva introduciendo bits del mismo signo que el bit que indica el signo (en el operando de datos es el bit más significativo). Para desplazamientos a la izquierda, introduciremos siempre ceros en los bits menos significativos de la palabra. Si deseamos conservar el signo del operando de esta manera entonces la operación de desplazamiento se denomina *desplazamiento aritmético*.

En el 68000 estas instrucciones de desplazamiento aritmético son ASL para el desplazamiento aritmético a la izquierda (*Arithmetic Shift Left*) y ASR para el desplazamiento aritmético a la derecha (*Arithmetic Shift Right*). Si el destino es un registro de datos, entonces podemos desplazar hasta ocho bits empleando el modo inmediato para el número de desplazamientos, o podemos servirnos del contenido de otro registro de datos como contador de desplazamientos hasta 31 bits en una palabra larga. Así, por ejemplo:

```
ASR     #8,D0
```

será el desplazamiento máximo en el modo inmediato, y:

```
ASL     D1,D0
```

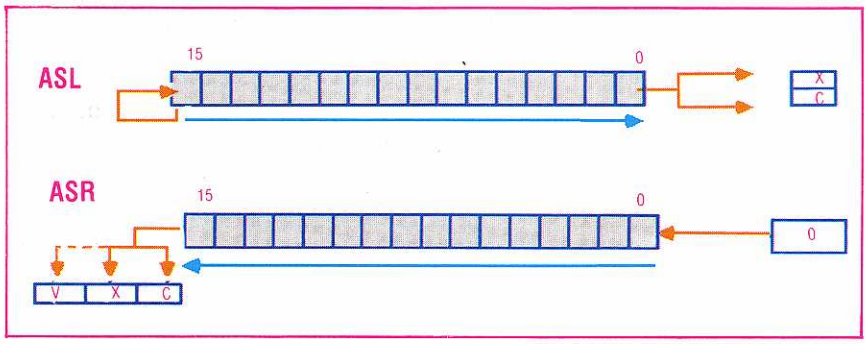
nos permitirá desplazar D0 a la izquierda un número de lugares indicado por el contenido de D1 (esto es, hasta 31 lugares). Si, no obstante, deseamos desplazar una posición de memoria, entonces la única opción posible será desplazar un lugar cada vez. Por ejemplo, ASR JUAN desplazará la posición de memoria indicada por JUAN un lugar a la derecha.

Se observará, por el dibujo del control de estado (página contigua), que se afectan todos los códigos de condición del SR (los bits C y X señalan los bits descartados por el desplazamiento, y el bit V indica cualquier cambio del bit de signo en desplazamientos a la izquierda). Los restantes bits, el N y el Z, reciben el valor acorde con el resultado.

Puede que usted se pregunte por qué se atribuye tanta importancia a estos desplazamientos aritméticos, considerando sobre todo que el 68000 está provisto de instrucciones para multiplicar y dividir. La razón está en que el tiempo de ejecución de estos dos tipos de operaciones es diferente. Una operación MULT tarda 70 ciclos de reloj y una ASL o ASR tan sólo tarda 6+2n ciclos, donde "n" es el número de desplazamientos ejecutados. Así, el tiempo de ejecución de un desplazamiento aritmético está entre los ocho ciclos para un solo desplazamiento y los 68 ciclos para un desplazamiento completo de 31 bits. De esta manera, para un reducido número de desplazamientos, las instrucciones de desplazamiento serán considerablemente más eficientes que las instrucciones de multiplicar o dividir. En el caso extremo de una instrucción de dividir por dos ¡se conseguiría una rapidez 19 veces mayor!

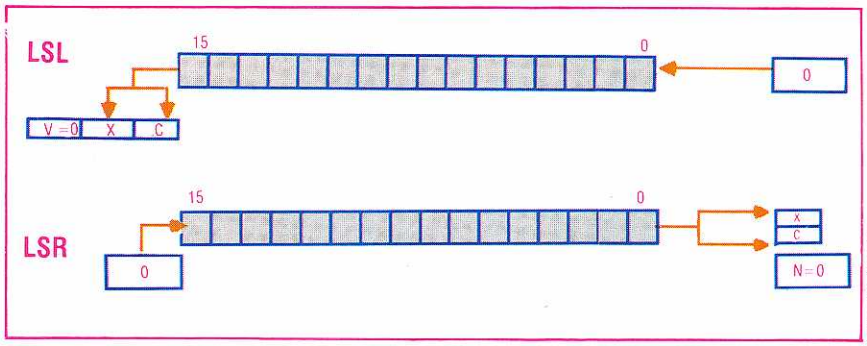
Obsérvese que también disponemos de instrucciones de *desplazamiento lógico*, LSL y LSR, que no conservan el signo del operando de datos e introducen siempre ceros en los nuevos dígitos binarios. Las restricciones de direccionamiento de los desplazamientos aritméticos son también válidas en los desplazamientos lógicos, y el bit V siempre se pone a cero. El desplazamiento lógico se utiliza provechosamente para establecer o comprobar operandos de datos que contengan subgrupos o campos más pequeños.

Finalmente en este conjunto de instrucciones de desplazamiento toca el turno al grupo de instrucciones rotatorias. Son sin duda una reliquia histórica,

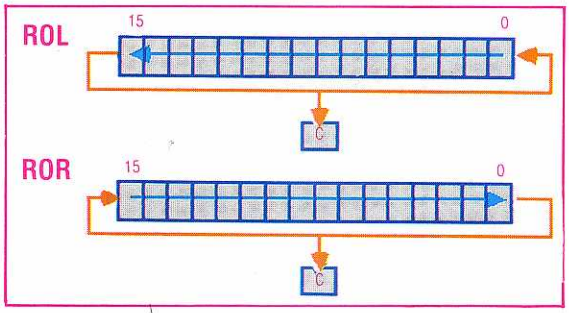


Desplazamiento aritmético
Cuando se desplazan los bits a la derecha (ASR), la operación aritmética copia el bit 15 en el bit 14, pero el contenido del bit 15 permanece inalterado. El bit de signo se respeta. El bit 0 se copia tanto en el bit C como en el X del registro de estado (SR). Obsérvese que cuando se efectúa un desplazamiento a la izquierda, los ceros se copian en el bit 0, y el bit V se pone a 1 si el cont. del bit 15 ha cambiado

Desplazamiento lógico
Las operaciones de desplazamiento lógico se limitan a introducir ceros bien sea en el bit 15 bien en el bit 0, según que el desplazamiento sea a la izquierda o a la derecha. Pero debe tenerse en cuenta que V se pone a cero durante la LSL, y N se pone a cero durante la LSR. El último bit que se desplaza fuera se almacena en C y en X en ambas operaciones



Caroline Clayton



Instrucciones de rotación
Las instrucciones ROTATE sólo hacen rotar a los operandos, poniendo el bit 0 en el bit 15 durante ROR y viceversa durante ROL. Obsérvese que el bit "intercambiado" se copia en el bit C del registro de estado durante la operación

dado que su principal finalidad es la comprobación de bits singulares de operandos de datos mediante la rotación y comprobación del estado resultante en el flag de condición. Naturalmente, en el 68000 disponemos de un conjunto completo de instrucciones para comprobar y modificar bits, y tardan casi el mismo tiempo que el conjunto de las rotatorias. Quizá pueda bastar aquí un sencillo ejemplo de rotación. ROR #3,D0 rotará el contenido de D0 tres lugares a la derecha, y dará al bit C un valor acorde con el último bit que ha dado la vuelta, yendo del menos significativo al más significativo.

Avancemos algo más para examinar las instrucciones de control de programas. Se trata de un importantísimo conjunto de instrucciones que controlan la secuencia de ejecución. Un grupo, denominado *bifurcaciones condicionales*, alterará el flujo secuencial normal de las instrucciones en función de una condición que se comprueba. Las *bifurcaciones incondicionales*, por su parte, siempre producirán una bifurcación o cambio del flujo normal secuencial de las instrucciones. Examinaremos primero estas últimas.

El modo habitual de ejecutar una bifurcación incondicional es:

```
BRA       NUEVAETIQ
```

donde el flujo de instrucciones continuará a partir de la posición denominada NUEVAETIQ una vez ejecutada la instrucción BRA. Si el desplazamiento de un byte puede guardarse en una palabra de ocho bits con signo, la instrucción resultante se codificará en una palabra. La mitad de la palabra contendrá el opcode BRA (60 en hexa) y el otro byte contendrá el desplazamiento de 16 bits con signo. Si el desplazamiento no es posible contenerlo de esta manera, o no se conoce todavía la dirección de bifurcación (es decir, el assembler no puede calcularla porque implica una referencia ulterior), entonces el byte de desplazamiento contendrá valor cero y la palabra siguiente contendrá el desplazamiento de 16 bits completo con signo.

Por lo general, la instrucción BRA será la adecuada; no obstante, en algunas ocasiones desearemos hacer algo más refinado cuando se calcula la dirección de la bifurcación. Por ejemplo, cuando se quiere bifurcar a una dirección contenida en una tabla con un índice establecido en un registro (recuérdese que esta manera de direccionar se denomina *indirecta con índice y desplazamiento*). Por desgracia, la instrucción BRA no acepta esta forma de dirección calculada, y así Motorola ha proporcionado la instrucción JMP (*jump*: saltar), donde es posible el cálculo de la dirección de bifurcación si es preciso.

Veamos un ejemplo de programa que ilustra las diferentes formas de bifurcación incondicional. La lista adjunta (*De un lugar a otro*) ilustra los principios en que se basa. Las instrucciones de salto en este ejemplo muestran un direccionamiento absoluto (hacia adelante y hacia atrás) e indirecto con indexación y desplazamiento. Otros modos de direccionamiento permitidos son el *indirecto simple* —por ejemplo, (A2)— y el *relativo al PC*.

Las instrucciones BRA muestran primero un ejemplo donde el desplazamiento de la dirección se contiene en la palabra de instrucción, y después dos ejemplos donde se utilizan palabras de extensión completa. En el primero de ellos (BRA FINISH) se ha utilizado una extensión completa de palabra aunque el desplazamiento puede ser contenido en un byte. Esto se debe a que el assembler no conoce la dirección de FINISH todavía, y por ello debe proveer una extensión completa de palabra para el desplazamiento. Si se sabe que el desplazamiento hacia adelante se ajusta a un byte con signo, entonces se puede obligar al assembler a que utilice la forma abreviada de la instrucción mediante el sufijo .S (*short*: breve). Así, nuestro ejemplo podría ser más adecuadamente escrito como BRA.S FINISH.

Examinemos ahora el segundo grupo de instrucciones de bifurcación: las *bifurcaciones condicionales*. Este grupo se subdivide en tres subgrupos:

- Bifurcaciones de complemento a dos
- Bifurcaciones sin signo
- Control de bucles

Los dos primeros subgrupos tienen un formato común en las instrucciones:

```
Bcc ETIQ
```

donde cc se refiere al código de condición que se comprueba. Si esta condición es verdadera, entonces tiene lugar una bifurcación a ETIQ; en caso contrario se ejecuta la siguiente instrucción de la secuencia. La condición comprobada se muestra en el cuadro *Condiciones para bifurcar* (derecha).

La columna "Verdadero si" del cuadro es la condición aritmética resultante de la comparación por medio de CMP o la instrucción SUB (que, obviamente, se ejecuta inmediatamente antes de aplicarse la bifurcación condicional). En el caso del primer subgrupo de condiciones mostrado en el cuadro (las bifurcaciones del complemento a dos), el bit V, o flag de desbordamiento, se incluye en todo caso en la comprobación lógica y éste es el factor que determina la pertenencia a este subgrupo. Implica además la necesidad de una comprobación adicional del bit de desbordamiento para una corrección completa, por lo que es preciso examinar cuidadosamente las condiciones lógicas para las bifurcaciones indicadas en el *Manual del usuario*. Por ejemplo, una bifurcación BGE comprueba si N=V, y habrá una bifurcación si se cumple NOT N AND NOT

De un lugar a otro

```
                    WAYOFF EQU     $3FFF
                           ORG     $1000
100     3200       START  MOVE    D0,D1
1002    4EF8 1000         JMP     START     *Extensión pal. completa
1006    4EF8 101C         JMP     FINISH    *De nuevo pero hacia adel.
100A    4EE8 0005         JMP     5(A0)     *Indirecto con desplaz.
100E    4EF2 000C         JMP     12(A2,D0  *Indirecto con índice y
                                             desplazamiento

    *Versiones que siempre bifurcan

    *Desplaz. en la palabra del opcode si está en un byte

1012    60EC              BRA START         *Desplazamiento negativo
1014    6000 0004         BRA FINISH        *Hacia adelante, sin saberse
                                             cuánto hacia adelante
1018    6000 2FE5         BRA WAYOFF        *Da una pal. de desplaz.
                                             de 16 bytes completa
101C    3200       FINISH MOVE D0,D1
```

V (es decir, si no hay desbordamiento y no es negativo), o cuando ambos N y V son verdaderos (desbordamiento y negativo).

Veamos un ejemplo. Supongamos que se desea comparar los dos números con signo contenidos en D1 y D2 y bifurcar hacia MAYOR si D2 es más grande que D1. Si la condición se comprueba antes con una instrucción comparativa, se emplea la bifurcación condicional BGT como sigue:

```
CMP   D1,D2       *forma D2-D1
BGT   D2MAYOR     *bifurca a D2MAYOR si no es
                  *cero ni negativo y no hay
                   desbordamiento
```

El segundo subgrupo de bifurcaciones condicionales ilustrado en el cuadro trata los números sin signo y la comparación de los códigos de condición es relativamente más sencilla. Por ejemplo, para comparar el contenido de la posición LUISA con D1, con independencia de cualquier condición de desbordamiento, se puede ejecutar esta secuencia:

```
CMP   LUISA,D1    *forma D1 - LUISA
BEQ   IGUAL       *bifurca a IGUAL si z=1
```

Un segundo ejemplo de estas bifurcaciones condicionales se da cuando se codifica en ensamblador para la realización de un bucle. Por ejemplo, el ensamblador equivalente de:

```
FORI:=1 TO 5 hacer
(...parte principal del programa a ejecutar 5 veces...)
NEXT I
```

sería:

```
      MOVEQ  #5,D7   *establece contador bucle
LOOP  (...parte      princ.   prog. a ejecutar 5 veces...)

      SUBQ   #1,D7   *decrementa el contador
      BNE    LOOP    *repite hasta que D7=0
```

quedando codificado eficazmente en sólo tres palabras (dado que se han usado las instr. rápidas).

El tercer subgrupo de bifurcaciones condicionales está formado por una única instrucción, DBcc (*Decrement and Branch*: decrementar y bifurcar al cumplirse la condición cc). Esta instrucción es una ampliación del programa de control de bucles dado más arriba, pero codificando tanto el decremento como la bifurcación condicional en una sola instrucción. De hecho la instrucción imita el pseudo-code típico del PASCAL para el bucle reiterativo:

```
REPEAT
  (...cuerpo del bucle...)
UNTIL
  'c'=verdadero o Dn=-1
```

donde cc es una de las condiciones descritas en el segundo subgrupo ilustrado en el cuadro y Dn es un registro de datos empleado para retener el contador del bucle. Veamos cómo puede codificarse con la instrucción DBcc:

```
      MOVEQ #5,D1 *establece contador bucle
LOOP  (...cuerpo del programa...)
      DEBQ D1,LOOP *salida si el último 'cc' es 0
                   *o bien D1=1 (6 iteraciones)
```

Obsérvense con cuidado las condiciones de salida explicadas en los comentarios anteriores, y cómo además el sentido de la bifurcación es diferente respecto de la instrucción BEQ normal. Si la comprobación condicional no es necesaria, entonces puede emplearse una cc de F (de "falso") para dar el equivalente DBcc de un simple bucle con FOR. Por ejemplo:

```
      MOVEQ #4,D3
LOOP  (...cuerpo del bucle FOR...)
      DBF   D3,LOOP
```

Lo cual equivale a nuestro bucle FOR original de cinco iteraciones, y hasta ocupa la misma memoria (todas las instrucciones DBcc ocupan dos palabras). Algunos programadores opinarán, sin embargo, que nuestra codificación primera era más explícita.

Condiciones para bifurcar

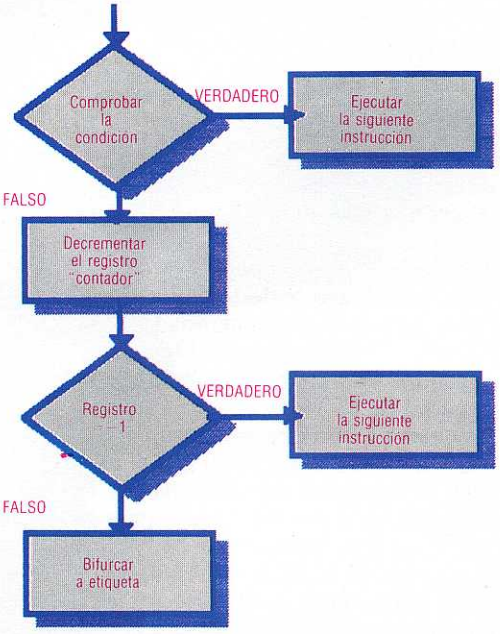
Las siguientes condiciones pueden comprobarse antes de una bifurc. hacia una dir. calculada por medio de la aritmética del complemento a dos

Condición cc	Verdadero si
GT mayor que	destino>fuente
LT menor que	destino<fuente
GE mayor o igual que	destino>=fuente
LE menor o igual que	destino<=fuente
VS desbordamiento	V=1
VC no hay desbord.	V=0

Las siguientes condiciones pueden comprobarse antes de realizar una bifurcación hacia una dirección calculada mediante enteros sin signo

EQ igual a cero	Z=1
NE no igual a cero	Z=0
MI menos	N=1
PL más	N=0
HI más alto que	C=Z=0
LS más bajo que	C o Z=1
CS arrastre activado	C=1
CC arrastre limpio	C=0

A condición de que...



Tiempo para comprobar
La DBcc proporciona al programador del 68000 varias y poderosas oportunidades de programación a alto nivel. El diagrama de flujo de la instrucción es el que aquí se muestra. Obsérvese que no sólo se limita a decrementar el registro contador (compararla con la instrucción DJNZ del Z80), sino que también comprueba la condición. A veces, sin embargo, podemos necesitar la ejecución de un bloque entero de código reiteradamente (como en el bucle FOR...NEXT del BASIC), pudiéndose emplear para este fin la instrucción condicionada FALSE (DBF)