Capítulo 8. Aritmética del Procesador

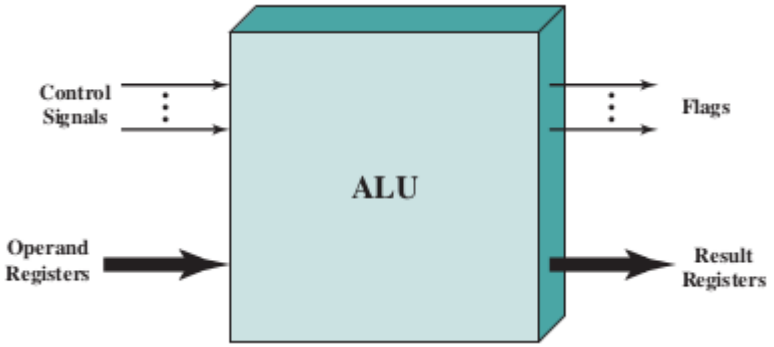
Las dos preocupaciones principales de la aritmética de una computadora son la manera en que se representan los números (el formato binario) y los algoritmos utilizados para las operaciones aritméticas básicas (suma, resta, multiplicación y división) tanto para números enteros como de punto flotante.

Los números de punto flotante se expresan como un número (significando) multiplicado por una constante (base) elevada a una determinada potencia entera (exponente). Los números de punto flotante pueden ser utilizados para representar números muy grandes o muy pequeños.

La mayoría de los procesadores implementan el estándar IEEE 754 para la representación y aritmética de números de punto flotante. En el IEEE 754 se definen formatos para 32 y para 64 bits.

8.1 La Unidad Aritmética Lógica (ALU)

La ALU es la parte de la computadora que efectúa las operaciones aritméticas y lógicas sobre los datos. El resto de los elementos del sistema – unidad de control, registros, memoria, I/O – están ahí para llevar datos hacia la ALU para que ésta los procese y luego regresar los resultados.



Como el resto de los componentes electrónicos de la computadora, la ALU se basa en el uso de dispositivos simples de lógica digital que pueden almacenar dígitos binarios y realizar operaciones simples de lógica booleana.

La figura anterior indica, en términos generales, cómo está interconectada la ALU con el resto del procesador. Los datos se presentan a la ALU en registros, y los resultados de una operación se almacenan en registros. Estos registros son localidades de almacenamiento temporal dentro del procesador que están conectados a la ALU mediante líneas de señal. La ALU puede también establecer banderas como resultado de una operación. Por ejemplo, una bandera de desbordamiento se activa si el resultado de una operación excede la longitud del registro en el cual se debe almacenar. Los valores de estas banderas también se almacenan en registros dentro del procesador. La unidad de control provee señales para controlar la operación de la ALU.

8.2 Representación de enteros

En el sistema binario, números arbitrarios pueden ser representados simplemente con los dígitos 0 y 1 y un punto. Ej. -1101.0101 = - 13.3125.

Sin embargo, para almacenar estos números en una computadora no contamos con el lujo de tener signos de menos o puntos decimales. Únicamente se pueden utilizar dígitos binarios para representar a los números. Si nos limitamos a números enteros no negativos, entonces la representación es simple, se utiliza el equivalente binario.

Representación Signo-Magnitud

Existen varias convenciones alternativas utilizadas para representar enteros tanto positivos como negativos, donde todas incluyen el utilizan el bit más significativo como bit de signo. Si el bit es 0, entonces el número es positivo, si el signo es 1, entonces el número es negativo.

La forma más simple de representación es la conocida como Signo-Magnitud. En una palabra de n bits, los n-1 bits de más a la derecha contienen la magnitud del entero.

$$+18 = 00010010 \qquad -18 = 10010010$$

El caso general es:

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

El problema con signo-magnitud es que las sumas y restas requieren de la consideración de ambos signos y sus magnitudes relativas. Otra desventaja es que existen dos representaciones para el número 0. Debido a estas desventajas, la representación signo-magnitud no se utiliza comunmente al implementar la porción entera de la ALU.

Representación de complemento a dos

El complemento a dos difiere de signo-magnitud en la manera en que se interpretan los bits de magnitud. Sin embargo, al igual que en signo-magnitud, la representación en complemento a dos se entiende mejor definiéndola en términos de una suma ponderada de bits. Considere el entero de n-bits A representado en complemento a dos. Si A es positivo, entonces el bit más significativo es cero. El resto de los bits representan la magnitud del número de la misma manera que en signo-magnitud.

El número cero se identifica como un entero positivo y por tanto tiene un bit de signo 0 y bits de magnitud todos en 0. Se observa que el número de enteros positivos que se pueden representar es [*0, 2^(n-1)-1*].

Para los números negativos, el bit de signo es 1. El resto de los *n-1* bits puede tomar cualquiera de los *2^(n-1)* valores. Por tanto, el rango de enteros negativos que puede ser representado es [*-1, -2^(n-1)*].

Twos Complement $A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$

La siguiente tabla compara las representaciones signo-magnitud y complemento a dos para enteros de 4 bits. Aunque la representación en complemento a dos puede parecer extraña desde la perspectiva humana, se observará que facilita las operaciones aritméticas más importantes, la suma y la resta. Por esta razón, es casi universalmente utilizada para representar enteros en los procesadores.

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

Conversión entre diferentes longitudes de bit

En ocaciones es deseeable almacenar un entero de n-bits en un formato de m-bits, donde m > n. En una representación signo-magnitud, esto se logra fácilmente: simplemente mueva el bit de signo a la posición de más a la izquierda y rellene con ceros.

+18	00010010	8 bits
+18	0000000000010010	16 bits

Este procedimiento no funcionará para la representación en complemento a dos de enteros negativos, en lugar de ésto la regla para enteros en complemento a dos es mover el bit de signo a la posición más significativa y rellenar los bits adicionales con copias del bit de signo.

-18	=	11101110
-18	=	1111111111101110

Es importante mencionar que estas las representaciones presentadas en esta sección se conocen como representaciones de punto-fijo. Lo anterior debido a que se considera que el punto binario está fijo a la derecha del bit menos significativo.

8.3 Aritmética entera

Negación

En signo-magnitud la regla para efectuar la negación de un número es muy simple: invertir el bit de signo. En complemento a dos, la negación de un entero se forma con las siguientes reglas:

1. Tome el complemento booleano de cada bit del entero (incluyendo el bit de signo).
2. Tome el resultado como un entero sin signo y sume 1.

Este proceso de dos pasos se conoce como la **operación complemento a dos**:

+18	=	00010010	(twos complement)
bitwise complement	=	11101101	
		+ 1	
		11101110	= −18
−18	=	11101110	(twos complement)
bitwise complement	=	00010001	
		+ 1	
		00010010	= +18

Existen dos casos especiales a considerar: A= 0 y A = -128:

0	=	00000000	(twos complement)
bitwise complement	=	11111111	
		+ 1	
−128	=	10000000	(twos complement)
bitwise complement	=	01111111	
		+ 1	
		10000000	= −128

Nótese que para el caso de A = -128, al realizar la operación complemento a 2 se obtiene el mismo número. Una anomalía como la mencionada es hasta cierto punto inevitable. El número de combinaciones de bits en una palabra de n-bits es 2^n, que es un número par. Como deseamos representar números positivos, negativos y 0, existe un número desigual de números positivos y negativos que se pueden representar. De ahí que no se pueda obtener el valor de negado de -128 puesto que no se puede representar en complemento a 2 utilizando únicamente 8 bits.

Suma y Resta

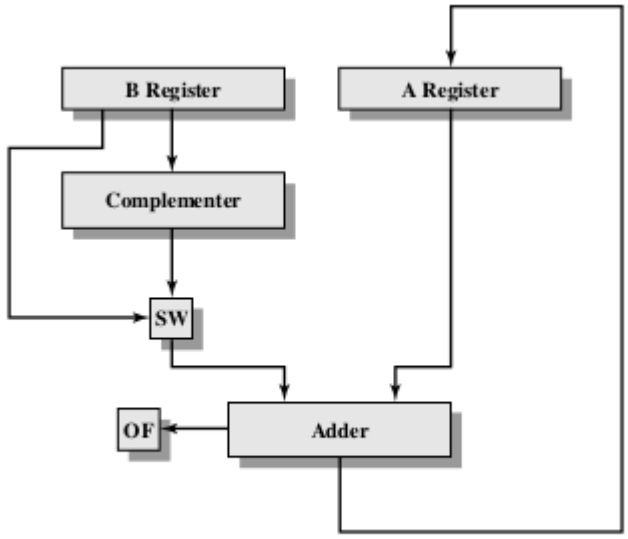
1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Para la **suma** de enteros en complemento a dos se procede como si los números fueran enteros sin signo. Si el resultado de la operación es positivo, se obtiene un número en complemento a dos positivo. Si el resultado de la operación es negativo, se obtiene un número negativo en complemento a dos. Note que en algunas instancias existe un ***bit de acarreo*** que sale de la longitud de la palabra y, por tanto, se ignora.

En cualquier suma, el resultado puede ser más grande que el que se puede almacenar debido al tamaño de la palabra. A esta condición se le conoce como ***desbordamiento (overflow)***. Cuando ocurre un desbordamiento, la ALU debe indicar este suceso de manera que el resultado de la operación no sea utilizado.

Para detectar el desbordamiento se observa la siguiente regla: si dos números son sumados y ambos son positivos o ambos son negativos, entonces el desbordamiento ocurre si y sólo si el resultado tiene el signo opuesto.

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 (c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100



OF = Overflow bit
SW = Switch (select addition or subtraction)

Para la **resta** se sigue la siguiente regla: tome el complemento a dos (negación) del sustraendo y súmelo al minuendo. Así, la resta se logra utilizando la suma. La regla del desbordamiento sigue aplicando. En la figura de la arriba se muestran algunos ejemplos de restas.

En la figura a la derecha se suguieren los caminos de datos y elementos de hardware necesarios para lograr la suma y la resta. El elemento central es el sumador binario, al cual se le presentan dos números a sumar y produce un resultado y un indicador de desbordamiento. El sumador binario trata a los números como enteros sin signo. Para la suma, ambos números se presentan al sumador desde dos registros, designados en la figura como A y B. El resultado puede ser almacenado en alguno de estos registros o en un tercero. El indicador de desbordamiento se almacena en una bandera de 1 bit (0 = no desbordamiento, 1 = desbordamiento). Para la resta, el substraendo (registro B) se pasa primero por un complementador a dos, de manera que su complemento a dos es presentado al sumador.

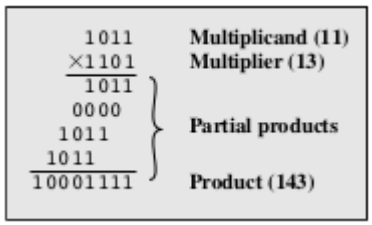
Se necesita una señal de control (SW) para indicar si se utiliza o no el complementador, dependiendo de si la operación es una suma o una resta.

Multiplicación

Comparada con la suma y la resta, la multiplicación es una operación compleja. Se han utilizado una gran variedad de algoritmos en varias computadoras. En esta sección se comienza por el ejemplo más simple de multiplicación de enteros sin signo para posteriormente pasar a la multiplicación de números en representación complemento a 2.

Enteros sin signos

En la figura se ilustra la multiplicación de enteros binarios sin signo. El proceso implica la generación de productos parciales, uno por cada dígito en el multiplicador, que posteriormente son sumados para producir un producto final. Los productos parciales se definen facilmente: si el bit del multiplicador es 0, el producto parcial es 0; cuando el bit del multiplicador es 1, el producto parcial es igual al multiplicando. El producto total se produce sumando los productos parciales. Para esta operación, cada producto parcial sucesivo está desplazado una posición hacia la izquierda con respecto al anterior. La multiplicación de dos enteros binarios de n bits resulta en un producto con longitud de hasta 2n bits (ej. 11 x 11 = 1001).

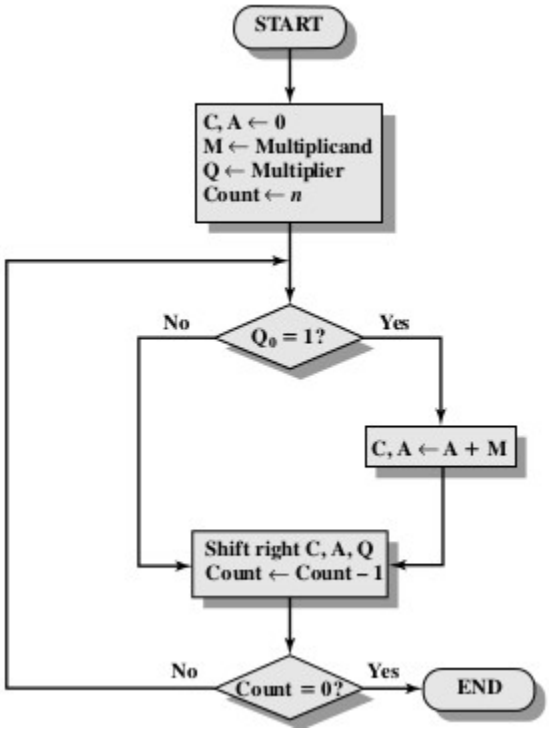


Hay varias cosas que se pueden hacer para hacer el cálculo más eficiente. Primero, se puede hacer la suma parcial de cada producto obtenido sin esperar hasta el final. Esto elimina la necesidad de almacenar dichos productos y, por tanto, se necesitan menos registros. Segundo, se puede ahorrar algo de tiempo en la generación de los productos parciales. Para cada 1 en el multiplicador, se requiere una operación suma y un desplazamiento; para cada 0, únicamente se requiere el desplazamiento.

A continuación se presenta el diagrama de bloques de dicha implementación y un ejemplo de su funcionamiento. El multiplicador y el multiplicando se cargan en los registros Q y M respectivamente. Se utiliza un registro adicional A que se inicializa en 0. Por último, existe también un registro de 1-bit C, inicializado en 0, que almacenará el bit de acarreo resultado de la suma.

La operación del multiplicador es la siguiente. Se lee cada bit del multiplicador uno a la vez:

1. Si Q_0 es 1, entonces se suma el multiplicando a A, el resultado se guarda en A y se guarda el bit de acarreo (si existe) en C. Posteriormente, todos los bits de los registros C, A y Q se desplazan una posición a la derecha: C -> A_{n-1}, A_0 -> Q_{n-1}, Q_0 se pierde.
2. Por otro lado, si Q_0 es 0, sólo se efectúa el desplazamiento a la derecha.



3. 3. El proceso se repite hasta leer todos los bits del multiplicador, el resultado queda almacenado en los registros A y Q.

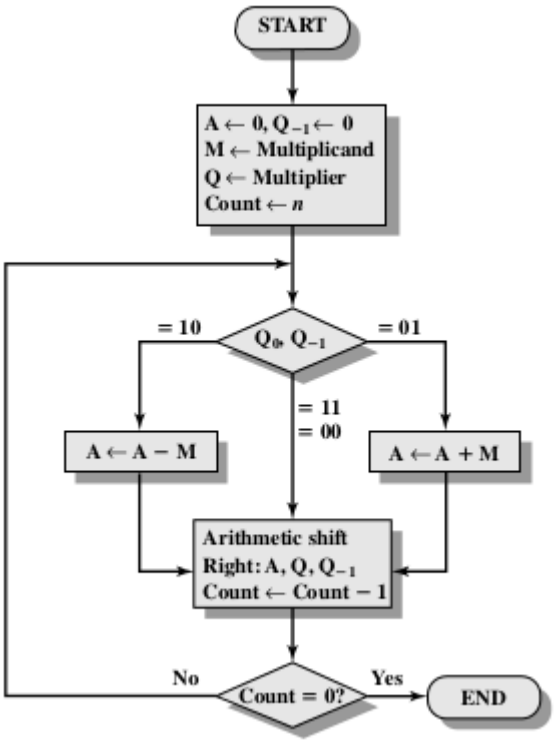
C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Multiplicación en complemento a 2

Para la multiplicación de enteros positivos se puede utilizar el mismo esquema puesto que su representacion es igual a los enteros sin signo. Sin embargo, no funcionará para enteros negativos, ya sea que tanto multiplicador y multiplicando sean negativos, o que sólo uno de ellos lo sea.

Existen un número de formas para resolver el problema, pero la más utilizada es el algoritmo de Booth debido a su eficiencia. El diagrama de bloques del algoritmo de Booth se muestra a continuación. Al igual que antes, el multiplicando y el multiplicador se almacenan en los registros M y Q. También existe un registro de 1 bit ubicado lógicamente a la derecha del bit menos significativo, Q_0 del registro Q, al cual se designa como Q_{-1}. El resultado de la multiplicación aparecerá en los registros A y Q. A y Q_{-1} son inicializados en 0. Al igual que antes, los bits del multiplicador se leen uno a la vez, sólo que esta vez también se analiza el bit a su derecha, es decir se analiza Q_0 y Q_{-1}. Si los dos bits son iguales, entonces todos los bits de A, Q y Q_{-1} se desplazan hacia la derecha 1 bit. Si los bits son diferentes, entonces el multiplicando se suma o se resta del registro A, dependiendo de si los dos bits son 0-1 o 1-0. Posteriormente a la suma o resta, ocurre el desplazamiento. En cualquier caso, el desplazamiento es tal que el bit de más a la izquierda de A, A_{n-1} , no sólo se desplaza hacia A_{n-2} sino que también se mantiene en A_{n-1}. Esto se requiere para mantener el signo del número y se le conoce como un **desplazamiento aritmético**.



A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	First cycle
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	Third cycle
0001	0101	0	0111	Shift	Fourth cycle

En la figura de la izquierda se muestran los pasos del algoritmo de Booth para la multiplicación de 7 x 3. El algoritmo de Booth es correcto por la siguiente razón: considere un multiplciador que consiste de un bloque de 1s rodeado por 0s, por ejemplo 00011110. Como sabemos la multiplicación puede ser lograda sumando de manera apropiada copias desplazadas del multiplicando:

$$\begin{aligned} \mathrm{M} \times (00011110) &= \mathrm{M} \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= \mathrm{M} \times (16 + 8 + 4 + 2) \\ &= \mathrm{M} \times 30 \end{aligned}$$

El número de operaciones se puede reducir a 2 si observamos lo siguiente:

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = 2^{n+1} - 2^{n-K}$$

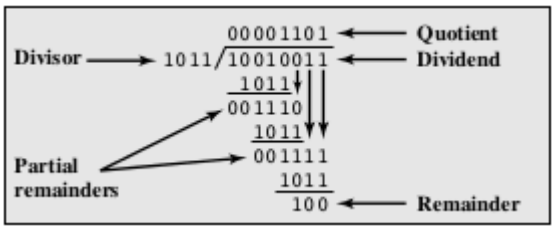
$$\begin{aligned} \mathrm{M} \times (00011110) &= \mathrm{M} \times (2^5 - 2^1) \\ &= \mathrm{M} \times (32 - 2) \\ &= \mathrm{M} \times 30 \end{aligned}$$

Así, el producto puede ser generado mediante una suma y una resta del multiplicando. Este esquema se extiende a cualquier número de bloques de 1s en el multiplicador, incluyendo el caso en el que un solo 1 es tratado como un bloque:

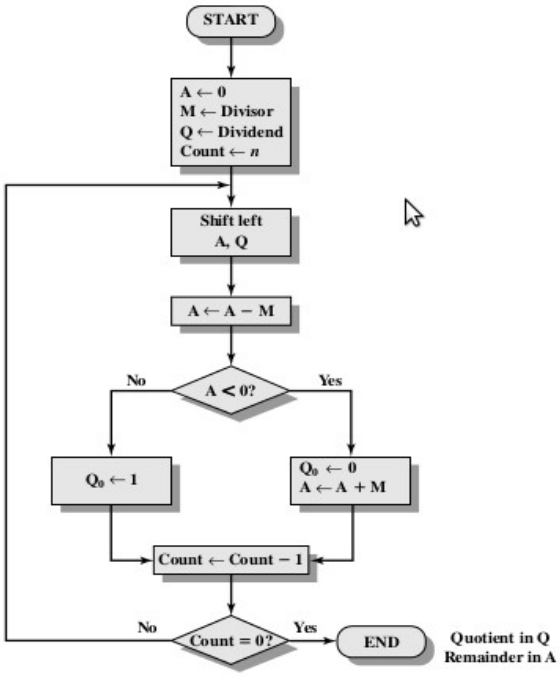
$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

El algoritmo de Booth atiende al esquema anterior efectuando una resta cuando se encuentra el inicio de un bloque de unos, una transición 1-0, y una suma cuando se encuentra el fin del bloque, una transición 0-1. Los bloques de 1s y 0s son saltados completamente, así, el algoritmo de Booth efectúa menos sumas y restas que el algoritmo tradicional de multiplicación.

División



La división es un poco más compleja que la multiplicación pero se basa en los mismos principios generales, en particular, restas y desplazamientos. Al igual que en el caso anterior, la base del algoritmo es el enfoque de papel-y-lapiz. En la figura a la derecha se muestra un ejemplo de división para enteros binarios sin signo. Primero, los bits del dividendo son examinados de izquierda a derecha hasta que el set de bits examinados representa un número mayor o igual al divisor. Hasta que este evento ocurre, se ponen 0s en el cociente de izquierda a derecha. Cuando el evento ocurre, se coloca un 1 en el cociente y el divisor se resta del dividendo parcial. El resultado de dicha resta es el residuo parcial. De este punto en adelante, la división sigue un patrón cíclico. En cada ciclo, bits adicionales del dividendo se adjuntan al residuo parcial hasta que de nuevo se tiene un número mayor o igual al divisor. Como antes, el divisor se resta de dicho número y produce un nuevo residuo parcial. El proceso continúa hasta que se han terminado los bits del dividendo.



A continuación se muestra el diagrama de bloques para implementar el algoritmo mencionado. El divisor se coloca en el registro M, el dividendo en el registro Q. En cada paso, los registros A y Q se desplazan a la izquierda 1 bit. M se resta de A para determinar si A divide al residuo parcial. Si así es, se coloca un 1 en Q_0. De otra forma, se coloca un 0 en Q_0 y se debe sumar M de nuevo a A para restaurar el valor previo. La cuenta se decrementa y el proceso continúa por tantos bits como tenga el dividendo. Al final, el cociente está en el registro Q y el residuo en el registro A.

Para números en complemento a dos, el algoritmo es el mismo. Se asume que el divisor V y el dividendo D son positivos y que V < D. Si V es igual a D, entonces el cociente Q = 1 y el residuo R = 0. Si V > D, entonces Q = 0 y R = D. El algoritmo es el siguiente:

1. Cargue el complemento a dos del divisor en el registro M, el registro entonces contiene la negación del divisor. Cargue el dividendo en los registros A y Q. El dividendo se debe expresar como un número positivo de 2n-bits. Así para un el número de 4 bits 0111, el dividendo se guarda como 00000111.
2. Desplazar A y Q hacia la izquierda 1 bit.
3. Restar M a A.
4. Si el resultado es no negativo (el bit más significativo de A = 0), entonces poner un 1 en Q_0. Si el resultado es negativo, entonces poner un 0 en Q_0 y restaurar el valor previo de A.
5. Repetir los pasos 2 a 4 tantas veces como posiciones haya en Q.

6. El residuo está en A y el cociente en Q.

Para lidiar con los números negativos, considere que las magnitudes de Q y R no son afectadas por los signos de V y D; y que los signos de Q y R son derivables de los signos de V y D. Así, una manera de hacer la división en complemento a dos es convertir los operandos a valores positivos y, al terminar, tomar en cuenta los signos de entrada para obtener los signos correspondientes del cociente y el residuo.

$$
\begin{array}{llllll}
D = 7 & V = 3 & \Rightarrow & Q = 2 & R = 1 \\
D = 7 & V = -3 & \Rightarrow & Q = -2 & R = 1 \\
D = -7 & V = 3 & \Rightarrow & Q = -2 & R = -1 \\
D = -7 & V = -3 & \Rightarrow & Q = 2 & R = -1
\end{array}
$$

8.4 Representación de punto flotante

Pag. 350