3C TIC *(Edición núm. 18)* Vol.5 – Nº 3
Septiembre – diciembre '16, 33- 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

Recepción: 18 de abril de 2016
Aceptación: 19 de septiembre de 2016
Publicación: 29 de septiembre de 2016

ALGORITMO DE BOOTH EN FORMA DE DÍGITOS SIGNADOS

BOOTH ALGORITHM IN SIGNED-DIGIT REPRESENTATION

Jesús Ayuso Pérez[1]

1. Compositor musical y desarrollador software. Licenciado en Ingeniería Informática por la Universidad Carlos III de Madrid (UC3M). E-mail: ayusoperez@terra.com

Ayuso Pérez, J. (2016). Algoritmo de Booth en forma de dígitos signados. 3C TIC: Cuadernos de desarrollo aplicados a las TIC, 5(3), 33-43. DOI: <http://dx.doi.org/10.17993/3ctic.2016.53.33-43/>.

3C TIC ***(Edición núm. 18)*** *Vol.5 – Nº 3*
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

RESUMEN

El algoritmo dado por Andrew Donald Booth en 1950 para la multiplicación puede interpretarse de la forma clásica: una manera de reducir las secuencias de 1s consecutivas existentes en la representación binaria de un número, o puede mirarse más desde el prisma en que lo hacen técnicas como la representación NAF, donde se entiende que, en lugar de apostar por una representación binaria del número se tiene una representación ternaria, que cumple la propiedad (o se espera que cumpla) de tener más 0s que la representación binaria clásica. En ambos enfoques el aumento de rendimiento se obtiene debido a que, en el esquema algorítmico de Booth, cuando el número contra el que se opera tiene un 0 en la *i-esima* posición. En ese *i-esimo* paso algorítmico nos ahorramos los cálculos (o parte de ellos), de ahí que en este documento vayamos a optimizar esos casos donde, según el prisma por el que miremos, compactamos las secuencias consecutivas de 1s o la representación ternaria tenga más 0s que la binaria.

ABSTRACT

The algorithm given by Andrew Donald Booth in 1950 for multiplication, can be interpreted in the classic way: a way to reduce existing sequences consecutive 1s in the binary representation of a number. Or you can look more from the prism they do techniques such as NAF representation: where it is understood that instead of opting for a binary representation of the number, it is a ternary representation, which meets the property (or is expected to meet) having more 0s than classical binary representation. In both approaches, the performance increase is achieved because, in the algorithmic scheme Booth, when the number against which it has a 0 in the ith position operates in the i-th algorithmic step, we saved the calculations (or part thereof). Hence in this paper we are going to optimize those cases where, according to the prism through which we look, we compact sequences of consecutive 1s or 0s ternary representation has more than binary.

PALABRAS CLAVE

Booth, algoritmo, multiplicación, inverso, dígitos signados.

KEYWORDS

Booth, algorithm, multiplication, inverse, signed-digit.

3C TIC *(Edición núm. 18)* Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

1. INTRODUCCIÓN

Como adelantábamos, el concepto de Booth basa su potencia en la posibilidad de reducir las operaciones que componen cierto cálculo eliminando secuencias de 1s consecutivos en la representación binaria, o usando una representación ternaria de un mayor ratio de 0s, apoyándose en la operación inversa, a la que compone el cálculo, para absorber el efecto algebraico de operar por la secuencia completa de 1s. Lo que Booth no notó en su algoritmo fue lo que ocurría ante una secuencia de 1s que consta de un único 1. Claro, si es un solo 1, no es una secuencia de 1s (tal vez sí, depende de cómo se defina), y en ese caso, no tratado por Booth, su algoritmo decrementa el rendimiento.

Por otra parte, si lo enfocamos más desde el prisma de la representación ternaria, más concretamente la representación NAF, eliminamos ese efecto negativo o ese caso no tratado por Booth. No obstante, tampoco es una solución que pueda terminar de satisfacernos siempre, o de encajarnos en cualquier situación, ya que la gran virtud de la representación NAF, que es que garantiza el óptimo, es decir: no existe una representación del número que tenga mayor número de 0s, es también su punto débil, ya que para garantizar esa optimalidad se puede volver un cálculo bastante pesado, de cara a agilizar o a ser aplicado el concepto de Booth a operaciones ligeras. En cambio, de la manera elegante e implementativa con la que Booth dio a conocer su concepto, su algoritmo puede ser incrustado sin ese coste de tener que calcular otra representación de los números.

2. METODOLOGÍA

Bien, vamos a tratar de ilustrar exactamente la problemática del concepto de Booth. Para que se vea mejor, vamos a intentar mostrarlo con una operación sencilla: de la adición, en nuestro caso algebraico, suma entre enteros.

Antes de nada, repasemos la tabla 1, dada por Booth para reducir el número de operaciones necesarias, apoyándonos en esa invertibilidad de la primitiva con la que se construye nuestra operación:

Tabla 1. Tabla de acciones de Booth.

bit menos significativo	bit extra	Interpretación	Acción
0	**0**	intermedio cadena de 0s	ninguna
0	**1**	final cadena de 1s	operación
1	**0**	comienzo cadena de 1s	operación inversa / inverso misma operación
1	**1**	intermedio cadena de 1s	ninguna

Partiendo de la tabla anterior, el algoritmo de Booth, como figura en la referencia bibliográfica '*Booth algorithm operations addition and subtraction*', aplicado a la suma de dos números, *a* y *b*, de longitud *n* quedaría:

3C TIC (Edición núm. 18) Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

```
result = a;

bitExtra = 0;

for(int i = 0; i < n; i++)  {

        switch(actionBooth(b[i], bitExtra) {

                case ( 0 1 ):

                        result = successor(result, i);

                        break;

                case ( 1 0 ):

                        result = predecessor(result, i);

                        break;

        }

        bitExtra = b[i];

}
```

Figura 1. Algoritmo de suma 2.
Fuente: elaboración propia.

Pues bien, imaginemos que en el algoritmo anterior nos encontramos con un valor de *b* que respeta la siguiente distribución:

```
01010101... 01010101
```

Si nos fijamos, el algoritmo de Booth, tal y como él lo definió, al operar contra un número cuya distribución de 1s fuera esa relación intercalada de 1s y 0s, lo que haría sería: comienza el algoritmo, el bit extra de Booth, contiene un 0 y el primer bit del número es 1, luego actúa en función de lo que dice la tabla de Booth para: '**1 0**', es decir, comienzo de secuencia de 1s; luego opera por la operación inversa. Acto seguido, en la siguiente iteración, el bit extra de Booth vale 1, y el siguiente bit del operando es 0, luego se actúa para '**0 1**', con lo que se opera de manera natural. Y así constantemente.

Volvamos ahora al algoritmo inicial de suma, descrito en '*Booth algorithm operations addition and subtraction*', basado en la primitiva de sucesor. Tenía este esquema:

```
result = a;

for(int i = 0; i < n; i++)  {

        if(b[i] == 1) result = successor(result, i);

}
```

Figura 2. Algoritmo de suma 1.
Fuente: elaboración propia.

3C TIC (Edición núm. 18) Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

Si nos fijamos, el anterior algoritmo únicamente opera en caso de que el *i-esimo* bit del operando sea 1. Luego nos encontramos con que el algoritmo definido por Booth, para este caso, acaba realizando justo el doble de operaciones que el esquema clásico. Si lo miramos desde el prisma de que, el concepto de Booth se basa en el uso de una representación ternaria (como se hace en la representación NAF). Podríamos explicarlo de manera que el algoritmo de Booth acaba representando el número binario con esa relación intercalada de 1s y 0s de la siguiente manera:

`1-11-11-11-1… 1-11-11-11-1`

Donde vemos claramente que, para ese caso, el concepto de Booth no trabaja con un número que tiene más 0s en su representación, sino justo lo contrario, acaba eliminando todos los 0s que había en la representación del operando.

Pensemos ahora que, en muchos casos, para muchos valores de ese supuesto *b*, podemos encontrarnos con un 1 suelto en alguno de los *n* bits que lo representan. Dándose ese problema, no constantemente, como ocurre en el peor caso expuesto, pero sí en muchas ocasiones. Pues bien, para evitar esa casuística, la de estar ante un único 1, se puede modificar el algoritmo de Booth, de manera que sea capaz de detectar esas situaciones y, ante ellas, se realice una sola operación (1 suma para el caso que nos ocupa), como haría el algoritmo clásico. Para ello simplemente lo que se hace es:

1. Llevar 2 bits extras, en lugar de uno.

2. Contemplar 2 bits del operando, en lugar de uno. Serán el *i-esimo* y, dependiendo de si estamos en un recorrido *right-to-left* o *left-to-right*, el *i-esimo* + 1 o el *i-esimo* - 1, respectivamente.

Sabiendo lo anterior, se puede fácilmente modificar la tabla 1 dada por Booth, para absorber esos casos, y realizar como decimos sólo una operación. La tabla, para un recorrido de derecha a izquierda, y entiendo que lo que denotamos como *x* toma un valor perteneciente al conjunto {0, 1}, sería:

Tabla 2. Tabla ampliada de acciones de Booth.

2 bits menos significativos	bits extras	Interpretación	Acción
x 0	0 x	intermedio cadena de 0s	ninguna
x 0	1 0	cadena de un 1 dejada atrás	ninguna
x 0	1 1	final cadena de 1s	operación
0 1	0 x	cadena de un 1	operación
1 1	0 x	comienzo cadena de 1s	operación inversa / inverso misma operación
x 1	1 x	intermedio cadena de 1s	ninguna

Fuente: elaboración propia.

3C TIC (Edición núm. 18) Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

Con lo anterior, describimos un nuevo algoritmo de suma de dos números, *a* y *b*, de longitud *n*, como:

```
result = a;
bitExtra1 = 0;
bitExtra2 = 0;
for(int i = 0; i < n - 1; i++)  {
        switch(actionBooth(b[i + 1], b[i],
                        bitExtra1, bitExtra2) {
                case ( x 0 1 1 ):
                case ( 0 1 0 x ):
                        result = successor(result, i);
                        break;
                case ( 1 1 0 x ):
                        result = predecessor(result, i);
                        break;
        }
        bitExtra2 = bitExtra1;
        bitExtra1 = b[i];
}
switch(actionBooth(0, b[n - 1], bitExtra1, bitExtra2) {
        case ( x 0 1 1 ):
        case ( 0 1 0 x ):
                result = successor(result, i);
                break;
        case ( 1 1 0 x ):
                result = predecessor(result, i);
                break;
}
```

Figura 3. Algoritmo de suma 3.
Fuente: elaboración propia.

3C TIC (Edición núm. 18) Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

Véase que, para no acabar indexando la estructura que contiene los bits de los que consta nuestro operando *b*, realizamos nuestro bucle hasta *n - 1*, en lugar de hasta *n*, como estábamos haciendo hasta ahora. Por ello, al salir del bucle, tenemos un último *switch* donde tratamos la última iteración en la cual sabemos que no existe bit *(n + 1)-esimo*.

Pero lo destacable del algoritmo anterior es la forma en la que, con una simple modificación del algoritmo dado por Booth, conseguimos evitar ese peor caso donde no mejoramos el rendimiento de las técnicas clásicas, sin tener que recurrir a representaciones ternarias precalculadas, como sería en caso de utilizar la representación NAF.

Entremos ya en el caso de mirar el concepto de Booth como una representación ternaria de los números, en lugar de representarlos en binario, como se ha hecho siempre en el mundo de la Computabilidad. En concreto, con la alternativa óptima, la representación NAF. Primero daremos una posible implementación para calcular esa representación ternaria de un operando.

La representación Non-Adjacent Form, NAF, como decimos, lo que hace es llevarse un entero de la forma binaria habitual a un sistema ternario de dígitos signados donde se cumple la propiedad, y es la propiedad culpable de su interés, pues es la responsable del incremento del rendimiento, de que: la representación NAF de un entero *b* tiene menos dígitos diferentes de cero que su representación binaria.

Una implementación del algoritmo de transformación NAF para obtener la representación ternaria de *b* con una longitud de *n* bits podría ser:

```
k = 0
result = 0;
tmp = b;
while(TRUE)  {
        if(tmp[0] == 1) {
                if(tmp[1] == 0) {  // tmp[1] == 0 && tmp[0] == 1
                        result[(2 * n) - 2]  = 1;
                        result[(2 * n) - 1]  = 0;
                } else {  // tmp[1] == 1 && tmp[0] == 1
                        tmp = successor(tmp, 0);
                        result[(2 * n) - 2]  = 1;
                        result[(2 * n) - 1]  = 1;
                }
        }
        tmp =>> 1;        // DESPLAZAMIENTO DE 1 BIT A LA DERECHA
```

3C TIC (Edición núm. 18) Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

```
        if(tmp == 0) break;

        k++;

        result =>> 2;     // DESPLAZAMIENTO DE 2 BITS A LA DERECHA

}

return k;
```

Figura 4. Algoritmo de cálculo de representación NAF.
Fuente: elaboración propia.

En el algoritmo expuesto en la figura 4, entenderemos que la variable *result* tiene un tamaño de *2 * n*, ya que para almacenar 1 dígito ternario requeriremos de 2 bits. Además, como vemos, devuelve dicha representación ternaria en la variable *result* y también retorna el valor de un entero *k*. Ese entero *k* determinará el número de dígitos ternarios de los que constará nuestra nueva representación del operando. Destacamos que etiquetamos el algoritmo con el nombre *calcNAF*.

Otro detalle a resaltar es que, el anterior algoritmo de cálculo de la representación NAF está asociado al recorrido *left-to-right*. En nuestro caso, como vamos a utilizar un recorrido *right-to-left*, vamos a requerir de un posterior ajuste. Algo así:

```
k = calcNAF(b, b_naf);

for(int i = k + 1; i < n; i++)

        b_naf =>> 2;     // DESPLAZAMIENTO DE 2 BITS A LA DERECHA
```

Figura 5. Algoritmo de ajuste NAF para recorrido right-to-left.
Fuente: elaboración propia.

Como aventurábamos, intentar calcular la representación ternaria en operaciones muy ligeras, por ejemplo, de alguno de los 2 operandos de los que consta una operación de suma de 2 enteros, no tiene demasiado sentido, ya que el coste de obtener dicha representación puede ser ya igual o más pesado que la propia operación de adición. Sin embargo, imaginémonos que la operación de suma se produce dentro de un contexto modular, y además, que tendremos que realizar cientos o miles de sumas y restas bajo un módulo (como es el caso de, por ejemplo, operaciones como la exponenciación modular). Entonces, representar dicho módulo *m* en por ejemplo NAF, sí podría tener mucho sentido. Por ello, expondremos una solución que detalla cómo conjugar esa técnica con el algoritmo de reducción basado en Booth. Tendríamos que la suma de dos números, *a* y *b*, de longitud *n*, módulo *m*, quedaría:

```
tmp = a;

result = b;

k = calcNAF(m, m_naf);

        for(int i = k + 1; i < n; i++)

        m_naf =>> 2;     // DESPLAZAMIENTO DE 2 BITS A LA DERECHA
```

3C TIC (Edición núm. 18) Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

```
int i = 0;

for(; i < k; i++)  {

        switch(actionBooth(m_naf[1], m_naf[0]) {

                case ( 0 1 ):

                  if( ! (tmp[0] & 0x01))  // SI ES IGUAL A 0

                   result = predecessor(result, i);

                  break;

                case ( 1 1 ):

                 result = successor(result, i);

                 if(tmp[0] & 0x01)

                   result = successor(result, i);

                 break;

                default:

                 if(tmp[0] & 0x01)

                   result = successor(result, i);

                 break;

        }

                tmp =>> 1;       // DESPLAZAMIENTO DE 1 BIT A LA DERECHA


        m_naf =>> 2;    // DESPLAZAMIENTO DE 2 BITS A LA DERECHA

}

for(; i < n; i++)  {

        if(tmp[0] & 0x01)

                result = successor(result, i);

        tmp =>> 1;       // DESPLAZAMIENTO DE 1 BIT A LA DERECHA

}
```

Figura 6. Algoritmo de suma modular 3.
Fuente: elaboración propia.

Entendemos que la función *calcNAF* deja la representación ternaria del módulo *m* en la variable llamada *m_naf*, además de devolver el número de dígitos ternarios de los que consta dicha variable. Como hemos adelantado, en nuestra implementación física, dicha variable se

3C TIC *(Edición núm. 18)* Vol.5 – Nº 3
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

acabará traduciendo por una variable con el doble de bits de nuestro operando, es decir, para nuestro caso *2 * n*. Pues bien, requerimos 2 bits por cada dígito binario del módulo que es representado en un dígito signado. De ahí que, como podemos observar en el código existente sobre este párrafo, al término de cada iteración desplazamos 2 bits el valor de *m_naf*, para encuadrar los 2 siguientes bits que contienen el valor del próximo dígito ternario, y en cambio, en el caso de la variable *tmp* únicamente nos movemos 1 solo bit, ya que ésta sí contiene una representación binaria.

Obviamente, se podría aplicar simultáneamente la mejora que soluciona el peor caso del algoritmo de Booth, con la anterior técnica de reducción de módulo basada en la representación NAF. En la descripción anterior sólo hemos contemplado el proceso de cancelación entre los incrementos de la adición y los decrementos de la reducción debido al módulo.

3. RESULTADOS Y DISCUSIÓN

El aplicar el concepto de Booth en cualquier operación que se construya sobre una sucesión de cómputos de otra operación invertible que compone a la primera, no siempre nos garantiza una mejora en el rendimiento, o un número menor de operaciones para converger al resultado final. Dependiendo de la distribución de los bits de los que constan nuestros operandos, podemos no siempre estar realizando el menor número de cálculos necesarios. Para evitar esto tendremos que afrontar la optimización en base a varios criterios: la sobrecarga de utilizar representaciones de números que garanticen una mayor distribución de 0s, como es el caso de NAF, y el evitar el fallo del algoritmo dado por Booth al afrontar un 1 individual entre los bits de los números.

En conclusión, la idea propuesta por Booth basa su rendimiento en un factor estadístico de la distribución de los 1s en la representación binaria de un número. Existen representaciones ternarias, que garantizan un mayor ratio de 0s, que la representación binaria, pero igualmente el rendimiento estará supeditado a esa probabilidad de que el operando se pueda expresar con un menor número de cruces entre la operación algebraica y su inversa, refiriéndonos, claro está, a la que compone a aquella que estamos calculando.

3C TIC ***(Edición núm. 18)*** *Vol.5 – Nº 3*
Septiembre – diciembre '16, 33 - 43
Área de Innovación y Desarrollo, S.L.
ISSN: 2254 – 6529
DOI: http://dx.doi.org/10.17993/3ctic.2016.53.33-43

4. REFERENCIAS BIBLIOGRÁFICAS

Ayuso, J. (2015). Booth algorithm operations addition and subtraction, *3C TIC*, 4(2), 113-119.

Ayuso, J. (2015). Booth algorithm modular arithmetic operations of addition and subtraction, *3C TIC*, 4(3), 222-229.

Booth, A.D. (1945). A method of calculating reciprocal spacings for X-ray reflections from a monoclinic crystal. *J. Sci. Instr*, Vol. 22, p. 74.

Booth, A.D. y Britten, K.H.V. (1947). *General Considerations in the Design of an Electronic Computer*. Princeton.

Booth, A.D. (1951). A signed binary multiplication technique. *Q.J. Mech. and Appl. Math,* 4(2), 236-240.

Burks, A., Goldstein, H. and Von Neumann, J. (946). *Logical Design of an Electronic Computing Instrument*. Princeton.

W. Reitwiesner, G. (1960). Binary Arithmetic, 231-308.