3. Unidad Aritmética Lógica (ALU)

Abordaremos los aspectos que permiten la implementación de la aritmética de un computador, atributo funcional de la Unidad Aritmética Lógica (ALU). Primero se revisitará lo relacionado a la forma de representar los números como una trama de bit, para posteriormente ver los algoritmos sobre dichas codificaciones, que permiten implementar las operaciones aritméticas de forma consistente. Veremos que la selección de los esquemas de codificación esta fuertemente vinculada a la eficiencia en la implementación de su aritmética.

Se utilizara como base el conocimiento adquirido sobre lógica digital (compuestas combinaciones y secuenciales). La idea es poder introducir las consideraciones de diseño que permitan implementar una ALU.

Otras consideraciones de diseño son las interfaces que tiene esta unidad. Esta unidad puede ser vista del punto de vista de sus interfaces de entrada-salida como:

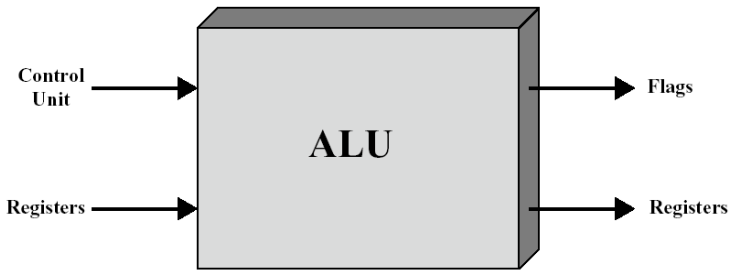


Figura a.1: Interfaces de la Unidad Aritmética Lógica

Las señales de entrada son:

- las líneas de control que determinan la operación a implementar;
- las líneas de datos de entrada correspondiente a los argumentos de la operación (registros de entrada)

Las señales de salida son:

- las líneas de datos donde se retorna el resultado de la operación implementada (registro de salida);
- los indicadores de estado (o **flags**), que indican la validez e información adicional de la operación (desbordamiento, acarreo, signo, zero, etc)

Como veremos las señales de flags son cruciales en la implementación de instrucciones de salto condicional. En este capitulo veremos en detalle los conceptos de acarreo, desbordamiento y su dependencia a la codificación considerada.

Definición de OVERFLOW

"El desbordamiento ocurre cuando el resultado de una operación aritmética implementada por la ALU cae fuera del rango de codificación definido para la salida".

3.1 Codificación Entera

Para el caso de codificación entera existen distintos formatos los cuales se describen a continuación.

Representación Entera sin Signo

Dada una palabra (o código) binaria $a_{n-1}, a_{n-2}, \ldots, a_0$ la representación decimal viene dada por:

$$A = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

Todos los bit de la palabra representan su magnitud, por siguiente el rango de codificación en n – bit va de 0 a $2^n - 1$

Representación signo-magnitud

Es la estrategia más sencilla para representar números enteros correlativos tanto positivos como negativos. La convención es que de una palabra de n bit, el bit más significativo (extremo izquierdo) represente el signo y los restantes n-1 bit representen la magnitud. El caso general el código $a_{n-1}, a_{n-2}, \ldots, a_0$ codifica:

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

Esta codificación presenta dos problemas de diseño:

- La unidad que implemente operaciones aditivas debe llevar en consideración el bit de signo de ambos argumentos, y en términos de ello, operar sus magnitudes. Es decir se necesita lógica de decodificación y la implementación de dos funciones lógicas sobre las magnitudes.
- Del punto de vista de la representatividad numérica, pues existen dos codificaciones para el valor 0.

$$+0_{10} = 0 \quad 0000000_2$$

$$-0_{10} = 1 \quad 0000000_2$$

Representación Complemento Dos

Este esquema de codificación permite muchas simplificaciones al nivel del diseño de la ALU y aborda adecuadamente el problema de la codificación del 0. Al igual que la codificación signo-magnitud, el bit más significativo representa el signo, pero la representación de los bits restantes es diferente.

Representación enteros positivos:
Cuando el bit más significativo es cero $a_{n-1} = 0$, la representación de los restantes n-1 bit es equivalente al esquema signo- magnitud. El rango de codificación es de 0 a 2^{n-1}-1.

Representación enteros negativos:
Del mismo modo $a_{n-1} = 1$ indica que se representa un número negativo, pero la magnitud se define como:

"el *valor entero sin signo de n bit que se debe sumar a la palabra* $a_{n-2}, a_{n-3}, \ldots, a_0$ *para generar la codificación entera sin signo de* 2^{n-1} *(es decir* ***100....0₂*** *, palabra de n-bit)*".

El rango de codificación va desde -1 a -2^{n-1}.

$$\hat{1}11\ldots111_2 \quad -1_{10}$$
$$\vdots$$
$$\hat{1}00\ldots000_2 \quad -2^{n-1}{}_{10}$$

Una expresión analítica consistente con las definiciones viene dada por:

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

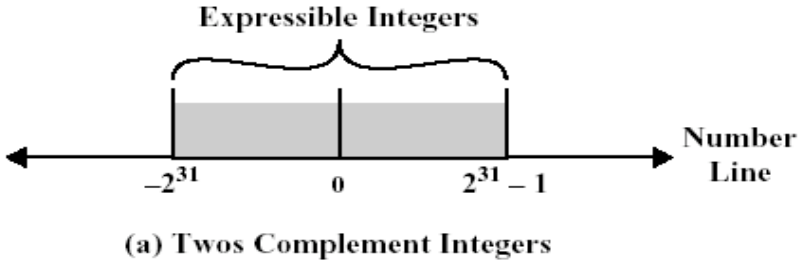


Figura a.2: *Rango de representatividad de la codificación complemento dos para una palabra de 32 bit.*

3.2 Operaciones Aditivas

Codificación entera sin signo

La suma en esta codificación es la generalización de la suma en base 10 la cual se ilustra en el siguiente diagrama.

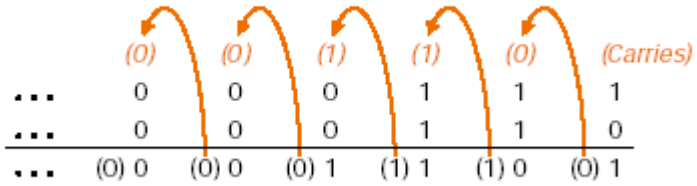


Figura a.3: Esquema de la implementación de suma en codificación entera sin signo

El paso inductivo se debe a la dependencia del bit de acarreo, y se deriva de la propiedad básica que $2^{n+1} = 2^n + 2^n$ lo que se manifiesta en un acarreo al siguiente bit de la representación.

En el caso que ocurra un acarreo en el bit más significativo, implica que dicho número escapa del rango de codificación 0 a $2^n - 1$. Por consiguiente estamos frente al escenario de **overflow.** Se verá que este evento de acarreo no se interpreta como **overflow** en otros esquemas de codificación.

Codificación entera complemento dos

Este esquema de codificación fue diseñado de tal forma que la suma binaria (consistente con la codificación entera sin signo) pueda ser utilizada de manera consistente, es decir no se necesite alambrar lógica adicional para implementar las operaciones de suma y resta en la ALU. Esta es la propiedad que ha transformado a este formato en un estándar para la representación entera.

El siguiente esquema, **Figura a.4**, muestra una representación circular de la estrategia de codificación e ilustra, al mismo tiempo, las operaciones de substracción, adición y sus restricciones en el ámbito de la representatividad numérica.

- Incrementar en una unidad se puede demostrar que equivale a rotar la referencia a favor de las manecillas del reloj. Con esta convención, se puede incrementar adecuadamente un número negativo en todos los casos (caso crítico pasa de –1 a 0 de forma consistente) y números positivo salvo para el caso 2^{n-1}-1 ($01..1_2$) pues pasa a la codificación -2^{n-1}(ver figura). En este caso puntual ocurre un desbordamiento (**overflow**), pues el valor resultante no se puede representar con n-bits en este esquema de codificación.

- De manera análoga decrementar un numero implica una rotación en sentido contrario, contexto en el cual el caso crítico ocurre con -2^{n-1} ($10..0_2$).

Las adiciones y substracciones en general se pueden visualizar como rotaciones en más de una unidad, donde por consiguiente, los escenarios de argumentos críticos (**overflow**) aumentan.

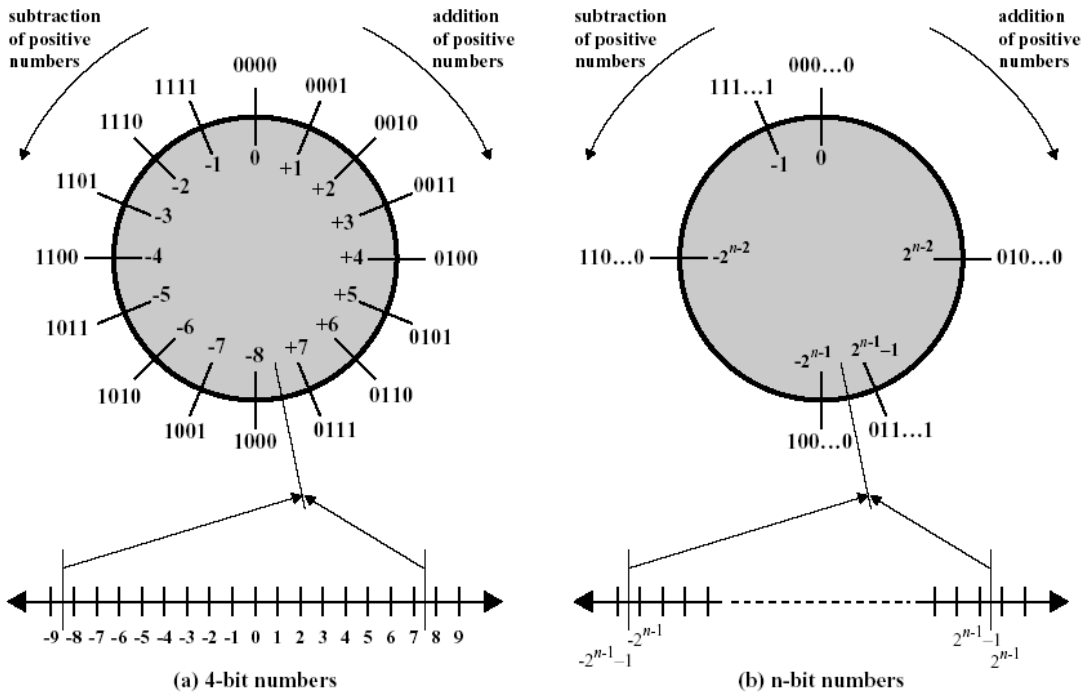


Figura a.4: *Representación circular de los esquemas de codificación complemento a dos y visualización de sus operaciones aritméticas básicas (suma, resta)*

Condiciones de Overflow

Del análisis se deriva que:

- al sumar dos números complemento dos de distinto signo, el resultado nunca genera un **overflow** pues la magnitud del número resultante es estrictamente menor que la de cada uno de sus argumentos, y por consiguiente representable.
- Sin embrago, al sumar números de igual signo, el **overflow** se da cuando, como consecuencia de estas rotaciones, el resultado corresponde a un numero de signo contrario al de sus argumentos. Esta condición define la lógica de control que debe implementar la ALU para detectar estos eventos.

Correctitud del algoritmo de Suma

En este punto demostraremos la correctitud del algoritmo de suma entera sin signo en el escenario de representatividad numérica complemento dos, distinguiendo adecuadamente los casos críticos (**overflow)**. (Visto en cátedra)

(<u>Propuesto 1</u>) Demuestre la correctitud del algoritmo de suma entera sin signo, cunado ambos argumentos son negativos. en los escenarios que corresponda y especifique los casos de **overflow**

Por lo tanto se demuestra la correctitud de la aritmética entera sin signo en el caso de codificación complemento dos salvo en los escenarios de **overflow**. Este sólo ocurre al operar argumentos del mismo signo y se manifiesta, sí y solo sí, el resultado de la operación aritmética es de signo contrario al de sus argumentos. Esto se visualiza en los siguientes ejemplos:

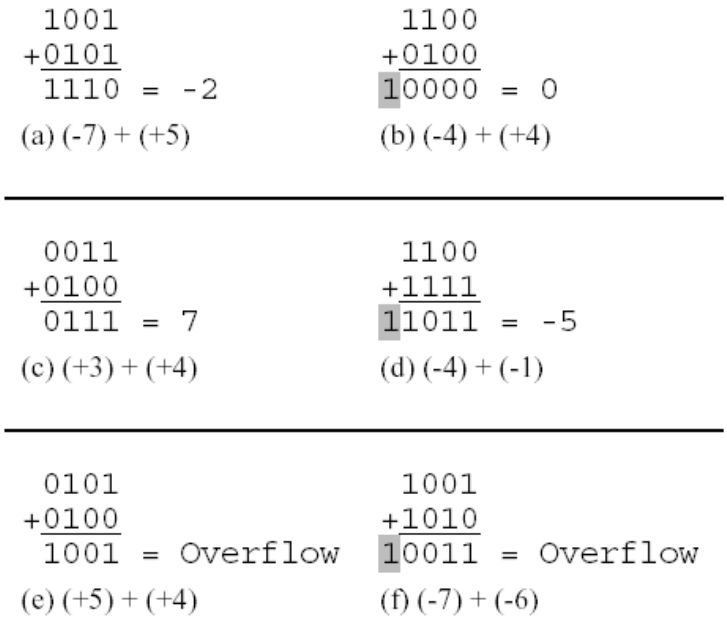


Figura a.6: *Suma de números en complemento dos*

Adicionalmente veremos que el complemento de un número se obtiene con lógica elemental, lo que implica que sólo con la implementación de la suma (en codificación entera sin signo) se obtienen las operaciones de substracción y adición complemento dos

Negación

En el contexto de la codificación complemento dos el complemento de un número se obtiene siguiendo la siguiente metodología:

1. Determinar el complemento bit a bit de la codificación de la palabra.
2. A la palabra resultante se incrementa en una unidad (aritmética entera sin signo).

La correctitud se vio formalmente en cátedra, pero de forma complementaria presentamos un argumento alternativo.

Si $a = (a_{n-1}, a_{n-2}, \ldots, a_0)$ es la expresión bit a bit de la palabra y su complemento b. Si notamos como $\overline{a}_i$ el complemento Booleano del bit a_i se tiene por definición que:

$$A = -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i \qquad \text{y} \qquad B = -2^{n-1} \cdot \overline{a}_{n-1} + \sum_{i=0}^{n-2} \overline{a}_i 2^i + 1$$

La expresión de B es consistente con los puntos 1 y 2, sólo si al incrementar en una unidad no se produce **overflow**. Es decir es válido salvo cuando *a* codifica el valor -2^{n-1}, pues su complemento *Booleano* es la palabra (0111...1) que al incrementarla produce **overflow**. Esto tiene sentido pues el

rango de representatividad numérico es asimétrico, -2^{n-1} a 2^{n-1}-1, y se esta tratando de obtener el complemento del valor -2^{n-1}.

Obviando tal caso, sólo basta probar que B es precisamente el complemento de A, o equivalentemente que $A + B = 0$

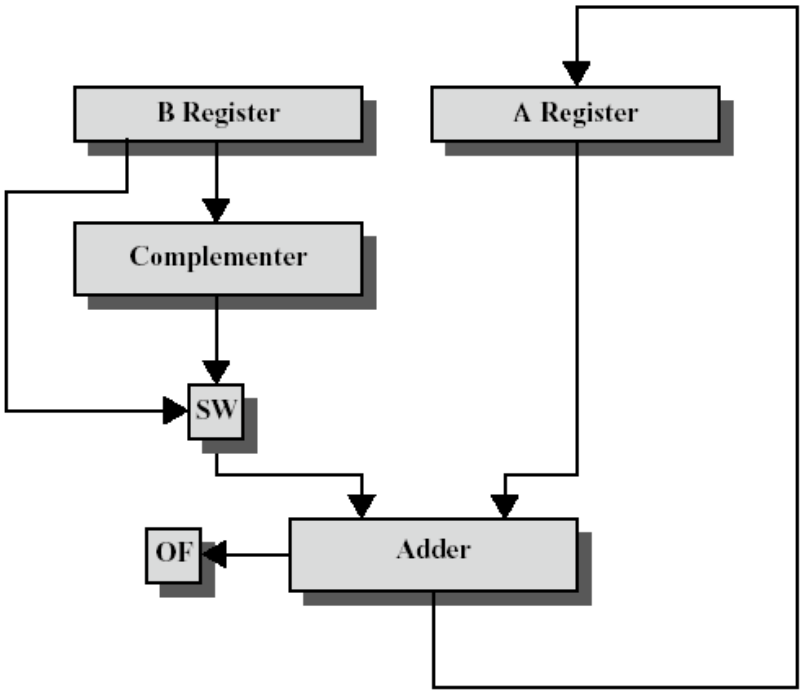
$$A + B = -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i + -2^{n-1} \cdot \overline{a}_{n-1} + \sum_{i=0}^{n-2} \overline{a}_i 2^i + 1$$

$$A + B = -2^{n-1} \cdot (\overline{a}_{n-1} + a_{n-1}) + \sum_{i=0}^{n-2} (\overline{a}_i + a_i) 2^i + 1$$

$$A + B = -2^{n-1} + (2^{n-1} - 1) + 1 = 0$$

De esta forma se tiene un algoritmo muy sencillo para obtener el completo de una palabra binaria, utilizando compuertas **not** y una unidad aritmética tradicional.

El siguiente esquema muestra las unidades elementales que permiten la implementación de la adición para codificación entera sin signo y complemento dos.



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

Figura a.7: Bloques funcionales para implementación de suma y resta

3.4 Multiplicación Entera Sin Signo

Comenzaremos con el escenario de multiplicación entera sin signo, para extrapolar al escenario complemento dos y terminar discutiendo una implementación más eficiente de dicho algoritmo. En este caso, pese a que el operador es conmutativo, hace sentido del punto de vista de la implementación, distinguir entre el **multiplicando** y el **multiplicador**. Este proceso para efectos de una implementación algorítmica lleva en consideración que:

1. La multiplicación es una suma de productos parciales.

2. Los productos parciales se definen como. Cuando el bit n de **multiplicador** es 1, el producto parcial asociado a este es el **multiplicando** desplazado en n-1 posiciones a la izquierda. Viene de la propiedad que:

$$\sum_{i=0}^{N-1} a_i 2^i \cdot 2^k = \sum_{i=0}^{N-1} a_i 2^{i+k}$$

Este producto parcial debe ser almacenado en una palabra de al menos N+(n-1) bit.

3. De la propiedad distributiva de la multiplicación respecto a la suma, el resultado total corresponde a la suma de los productores parciales.

4. El producto de palabras de N bit se da como resultado en una palabra de 2N bit.

(Propuesto 2) Demuestre que el resultado de multiplicar argumentos de N bit siempre se puede representar en una palabra de 2N bits.

Flujo de Ejecución Secuencial

El siguiente esquema muestra una implementación de la multiplicación entera sin signo, utilizando sumadores, lógica de control, registros y registros de desplazamiento:

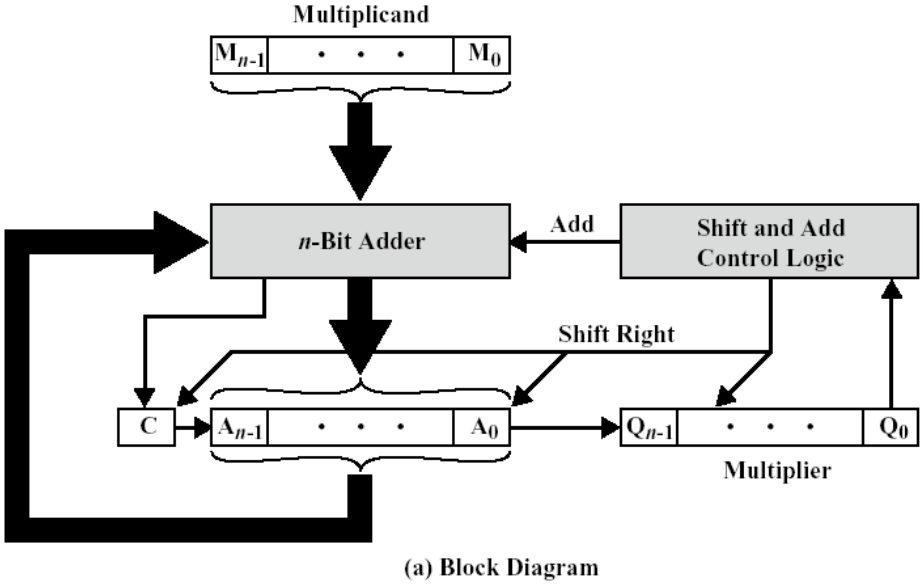


Figura a.8: *Unidades que implementan el algoritmo de multiplicación entera sin signo.*

Este esquema posee tres registros M el multiplicando, Q el multiplicador y A es el registro acumulador donde se van almacenando las sumas parciales.

Speudo-Algoritmo
La unidad de **desplazamiento y control** va analizando los bits del multiplicador, desde el menos significativo al más significativo, donde:

1. en el caso que el bit este activo, **la unidad de aritmética** suma la palabra M con A, retorna el resultado al acumulador A y la señal de carry a la celda **C**.
2. posteriormente tanto A como Q son desplazados hacia la derecha, pasando el bit C a la parte más significativa de A y el bit menos significativo de A a el más significativo de B.

De este proceso es fácil ver que al analizar los N bit del multiplicador, el resultado de la operación queda almacenado en la palabra de 2N bit producto de la concatenación de los registros A-Q. Donde el bit Q_0 esta asociado al término 2^0 y el bit A_{n-1} al término 2^{2n-1}.

Ejemplo: El siguiente esquema muestra la forma en que son modificados los registros A, Q y M, por al algoritmo recién planteado, etapa a etapa:

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	

Figura a.9: *estado de los registros en la ejecución de la operación de multiplicación*

En el siguiente esquema se muestra el diagrama de flujo del algoritmo de multiplicación implementado:

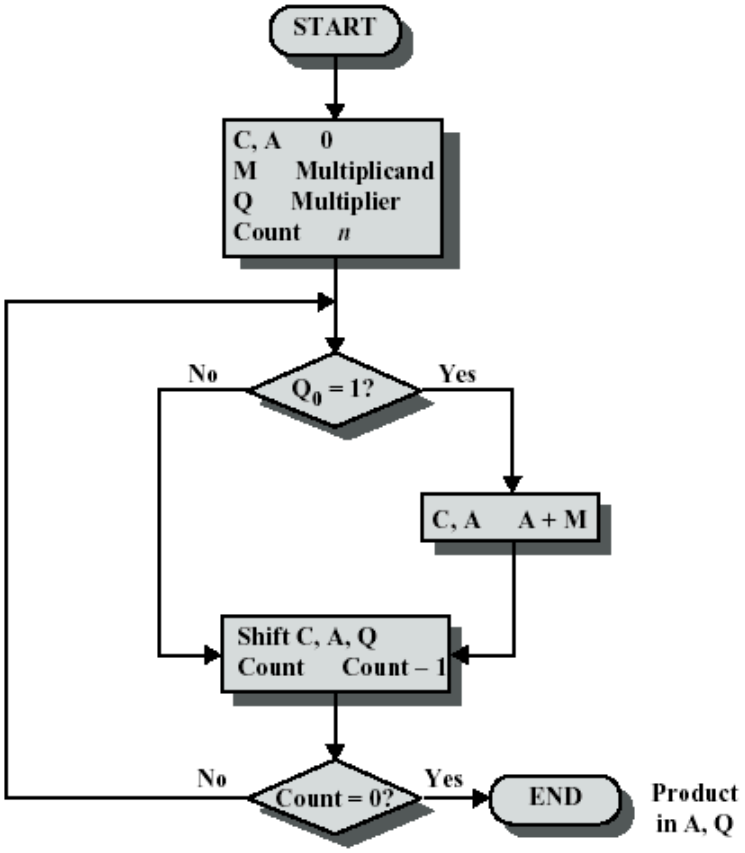


Figura a.10: *Flujo grama del algoritmo de multiplicación*

3.5 Multiplicación Codificación Complemento Dos

Una forma directa de implementar la multiplicación codificación complemento dos en N bit es pasar a la codificación entera sin signo de sus magnitudes en N bit, aplicar el algoritmo de multiplicación entera sin signo que genera una palabra de 2N bit y dependiendo de los signos de los argumentos (lógica de condición de signo), utilizar la regla de negación para generar un resultado en una codificación complemento dos de 2N bit.

(Propuesto 3) Demostrar que esta metodología es consistente y no se generan casos de **overflow** en la representación resultante de 2N bit.

Algoritmo de Booth

Se han encontrado maneras más eficientes de implementar esta multiplicación evitando pasar por las etapas de conversión entre esquemas de codificación antes mencionadas. Una de esas metodologías es el **Algoritmo de Booth**. Este algoritmo utiliza la las unidades funcionales y estructura presentes en la figura a.10.

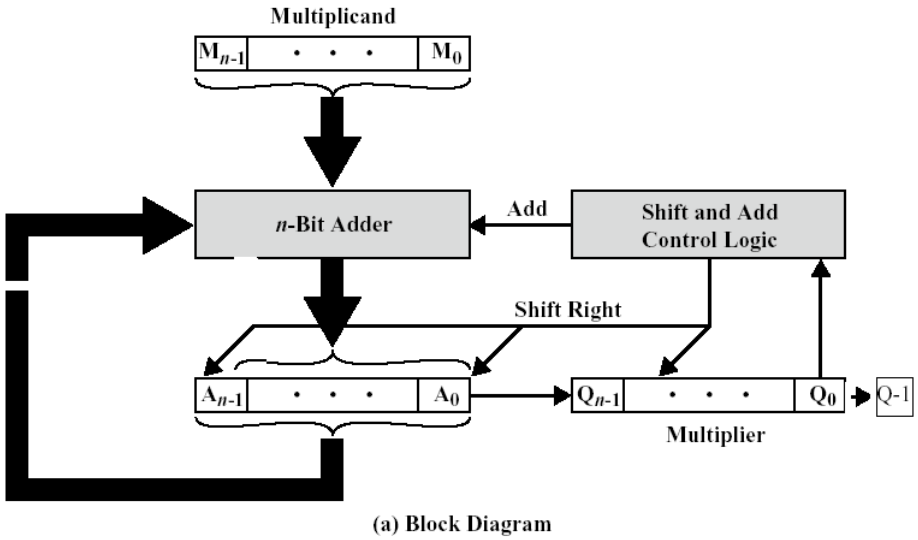


Figura a.10: *Unidades que implementan el algoritmo de multiplicación complemento dos.*

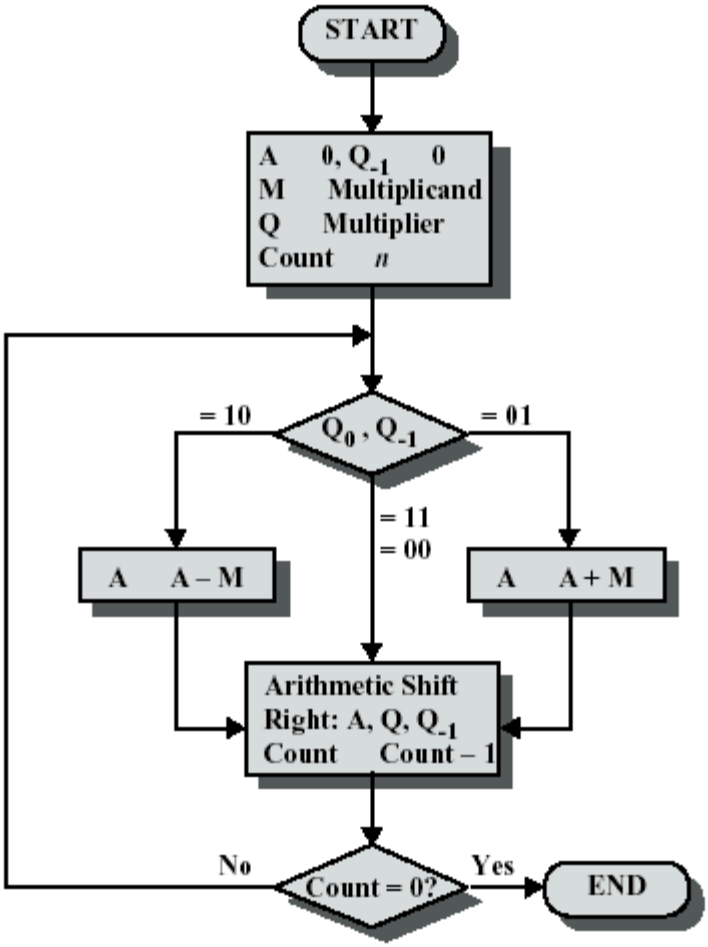


Figura a.11: *Flujo grama del algoritmo de multiplicación complemento dos utilizando el algoritmo de Booth*

Del punto de vista de operatoria hay dos aspectos que distinguen a este algoritmo.

- Las condiciones de operación que involucran el bit Q_{-1} (inicialmente en 0).
- (***desplazamiento aritmético***) El valor del bit A_{n-1} además de ser transferido al bit A_{n-2} permanece en el mismo estado. Es decir el desplazamiento no genera cambio de signo en el acumulador.

Speudo-Algoritmo

La metodología del algoritmo es la siguiente.

- Del registro Q, la **unidad de control de desplazamiento** analiza los bits Q_0-Q_{-1}. Cuando el valor de estos bits es **00** o **11** no se suma nada al acumulador A, si es **01** se suma el registro M al acumulador A y si es **10** se resta M al cumulador A.
- Posteriormente a cualquiera de los eventos anteriores se desplaza A, Q, Q_{-1} un bit a la derecha, con la consideración de desplazamiento aritmético en el registro A.

Al ejecutar n – veces esta metodología el resultado en codificación complemento dos de 2-n bit queda en los registros A-Q (ver flujo grama de la ejecución ***Figura a.11***)

Observaciones:

- La unidad aritmética debe implementar tanto sumas como restas complemento dos, problema ya resuelto utilizando esquemas como el de la **figura a.7**.
- El número de sumas y resta esta asociado a las transiciones de ceros y unos, que se puede demostrar que resulta en un método más eficiente, pues el número promedio de estas transiciones es menor que el número promedio de unos presentes en una palabra.

Correctitud del Algoritmo de Booth (No formal)
Partiremos analizando el caso donde Q, ***multiplicador***, corresponde a la codificación complemento dos de un numero positivo, es decir $Q_{n-1}=0$.
Sin perdida de generalidad supongamos el siguiente caso:

$$M \times (001110) = M \times \left(2^3 + 2^2 + 2^1\right) \tag{1}$$

se puede demostrar utilizando la regla de las series geométricas que:

$$2^m + 2^{m-1} + \ldots + 2^k = 2^{m+1} - 2^k \tag{2}$$

es decir en nuestro caso que:

$$M \times (001110) = M \times \left(2^4 - 2^1\right)$$

Por lo tanto si el algoritmo va analizando los bit del ***multiplicador*** de la expresión (1), cuando encuentre el primer 1, transición 10, se resta el valor de el multiplicando con el desplazamiento correspondiente (1 en este caso) y al encontrar el ultimo bit de la trama, es decir la transición 01, le suma el valor del multiplicando con el desplazamiento correspondiente (4 desplazamientos en este caso).

Esta metodología se generaliza para cualquier bloque de unos, y por consiguiente para cualquier codificación entera positiva del ***multiplicador***.

Ejemplo $$M \times (000110110) = M \times \left(2^5 + 2^4 + 2^2 + 2^1\right) = \left(2^6 - 2^4 + 2^3 - 2^1\right)$$

Los desplazamientos sobre M que implica esta metodología son sobre una codificación complemento dos, resuelva los siguientes propuestos para validar la correctitud de estos.

(Propuestos 4) Encuentre el método para representar un número complemento dos de n-bit en uno de n+1bit. Muestre que es equivalente a la definición del desplazamiento aritmético en el proceso de añadir el bit más a la izquierda de la palabra resultante.

(Propuestos 5) Analice que significa un desplazamiento aritmético a la derecha (conservando el bit de signo) y un desplazamiento simple a la izquierda (insertando un cero en el bit menos significativo de la palabra) en una codificación complemento dos de n-bit.
En que casos esto corresponde a multiplicar/ dividir por 2, respectivamente?
(Ind. Vea separadamente los casos de codificación positiva y negativa)

Finalmente nos queda analizar el caso en el cual el multiplicador es negativo, i.e. inicialmente $Q_{n-1}=1$.

Sin pedida de generalidad supongamos el multiplicando de la forma. $Q = (111...110....)$ con un bloque de unos consecutivos que llega hasta el bit Q_{k+1} (es decir el primer bit de izquierda a derecha distinto de uno es el bit k <= n-2). Se tiene que:

$$Q = -2^{n-1} + 2^{n-2} + ...2^{k+1} + \sum_{i=0}^{k-1} a_i 2^i$$

de (2) se tiene que

$$Q = -2^{n-1} + 2^{n-2} + ...2^{k+1} + \sum_{i=0}^{k-1} a_i 2^i = -2^{n-1} + 2^{n-1} - 2^{k+1} + \sum_{i=0}^{k-1} a_i 2^i = -2^{k+1} + \sum_{i=0}^{k-1} a_i 2^i$$

Por lo tanto $M \times (111...110...) = M \times \left(-2^{k+1} + \sum_{i=0}^{k-1} a_i 2^i \right)$

Observando la operación resultante y asociándola con la codificación del ***multiplicador***, esto corresponde a restar M desplazado k+1 veces, correspondiente a la etapa k+1 del algoritmo (donde se observa la última transición 10), más la multiplicación de M por una codificación entera positiva donde ya probamos la consistencia de la regla de Booth. Por lo tanto esta metodología es consistente en el caso de un multiplicador negativo.

De esta forma se puede ver que el flujo grama ***Figura a.11***, implementa la multiplicación complemento dos de dos palabras de n-bit a una palabra de 2n-bit.

(Propuesto 7) Demuestre que durante la ejecución del **algoritmo de Booth**, al sumar o resta el contenido de M al acumulador A no se genera la condición de **overflow,** es decir, la representación resultante de A+M o A-M es representable por un número complemento dos de n-bit.

Ind: Re-visite el caso donde al sumar don palabras de n-bits no se genera la condición de **overflow** (complemento dos), y aplique tal análisis a este problema.

(Propuestos 8) Realice las modificaciones al algoritmo de Booth para implementar de manera consistente multiplicación entera sin signo de n-bit a 2n-bit.
