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:

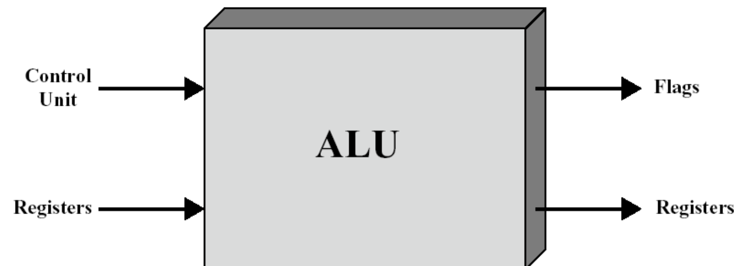

Figura a.1: Interfaces de la Unidad Aritmética Lógica

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 $a_{n-1}, a_{n-2}, ..., a_0$ 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 $a_{n-1}, a_{n-2}, ..., a_0$ 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.

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

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

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* $a_{n-2}, a_{n-3}, ..., a_0$ *para generar la codificación entera sin signo de* 2^{n-1} *(es decir* ***100....0**₂* *, palabra de n-bit)*".

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

$$\hat{1}11...111_2 \quad -1_{10}$$
$$\vdots$$
$$\hat{1}00...000_2 \quad -2^{n-1}{}_{10}$$

Una expresión analítica consistente con las definiciones viene dada por:

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

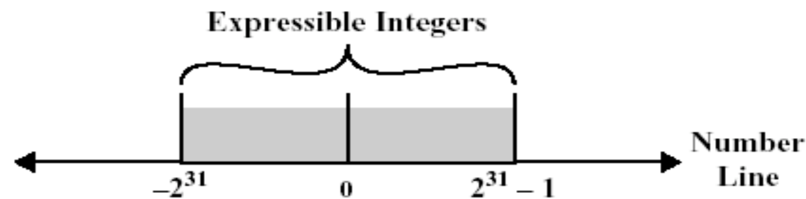

Figura a.2: *Rango de representatividad de la codificación complemento dos para una palabra de 32 bit.*

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.

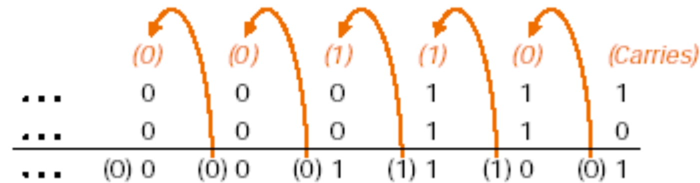

Figura a.3: Esquema de la implementación de suma en codificación entera sin signo

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.

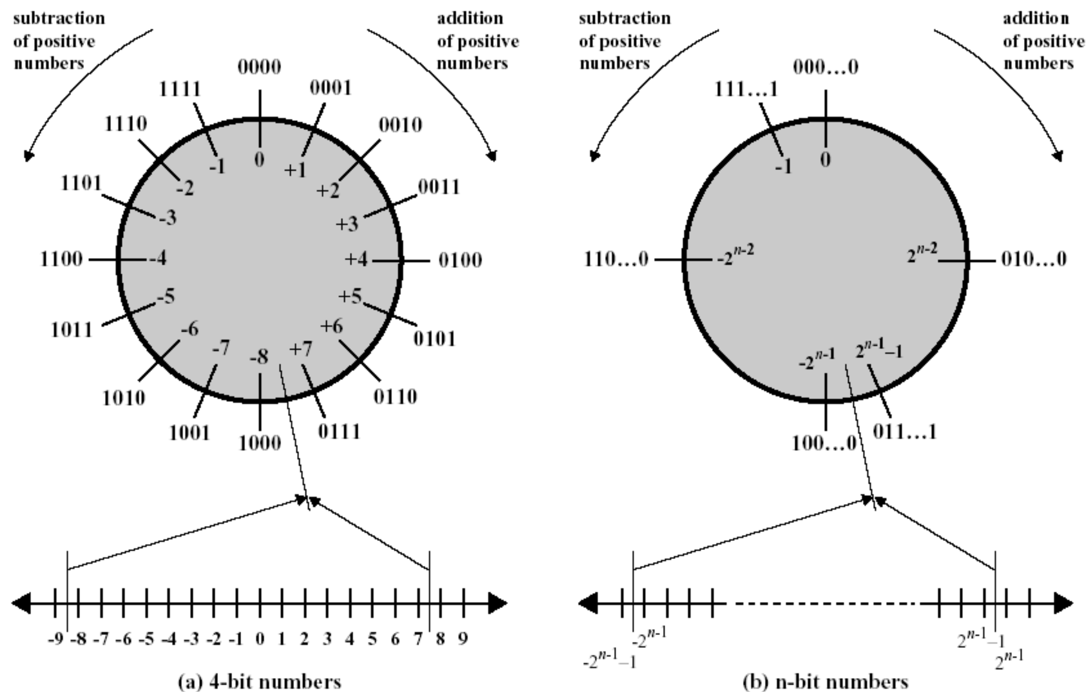

Figura a.4: *Representación circular de los esquemas de codificación complemento a dos y visualización de sus operaciones aritméticas básicas (suma, resta)*

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:

```
 1001                 1100
+0101                +0100
 1110 = -2           10000 = 0
```
(a) (-7) + (+5)   (b) (-4) + (+4)

```
 0011                 1100
+0100                +1111
 0111 = 7            11011 = -5
```
(c) (+3) + (+4)   (d) (-4) + (-1)

```
 0101                 1001
+0100                +1010
 1001 = Overflow     10011 = Overflow
```
(e) (+5) + (+4)   (f) (-7) + (-6)

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 $A + B = 0$

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

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

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

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.

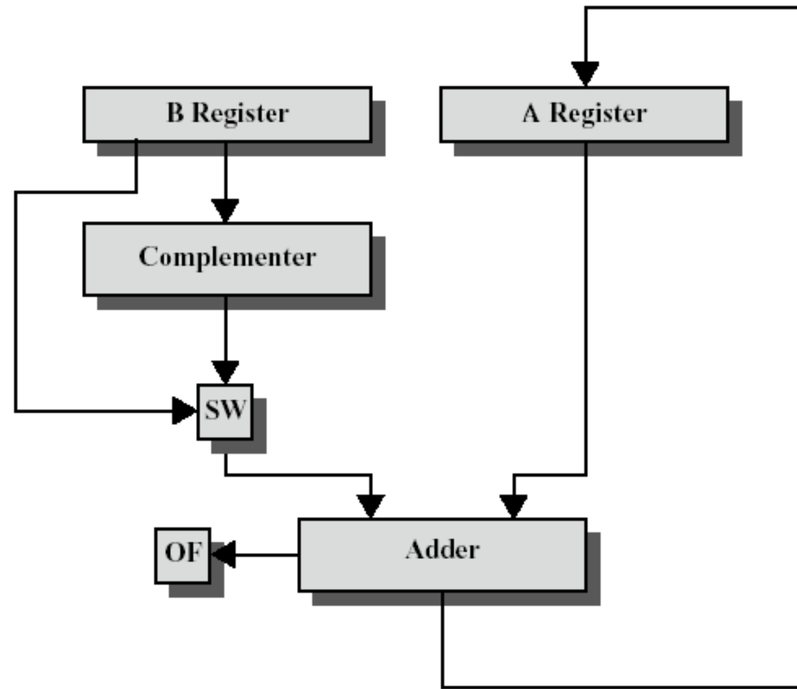

Figura a.7: Bloques funcionales para implementación de suma y resta