3.8 Construcción de una ALU básica

En este punto veremos como por medio de compuertas lógicas y multiplexores, se pueden implementar las operaciones aritméticas básicas de una ALU. Esencialmente en este punto implementaremos la aritmética de suma entera sin signo, eslabón funcional que como vimos en los puntos anteriores, permite implementar las operaciones de suma, resta y multiplicación en varios esquemas de codificación.

En primera instancia se verá una implementación básica, para luego introducir algunas optimizaciones de diseño que mejoran su prestación, medida como el tiempo requerido para generar una salida estable a partir de las entradas y la señalización de control. Entendiéndose que este último punto es proporcional al número de compuertas lógicas por las cuales deben propagarse las señales de entrada para generar la salida.

Los eslabones a utilizar en este diseño son los que se muestran a continuación con sus respectivas tablas de verdad:

1. AND gate ($c = a \cdot b$)

a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

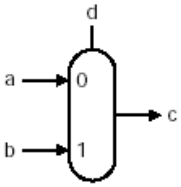
2. OR gate ($c = a + b$)

a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter ($c = \bar{a}$)

a	c = ā
0	1
1	0

4. Multiplexor
(if d = = 0, c = a;
else c = b)



d	c
0	a
1	b

Figura c_a 1: *Eslabones lógicos básicos*

Con ellos construiremos una ALU que implemente suma entera sin signo, suma y resta utilizando codificación complemento dos y las operaciones lógicas AND y OR sobre palabras de largo n. Para esto comenzaremos construyendo una ALU de 1–bit, para por medio de su concatenación lograr una ALU de n-bits.

3.8.1 ALU de 1 bit

En primera instancia se implementa una ALU que lleve a cabo las operaciones lógicas ***and*** y ***or***., como se muestra en la siguiente figura.

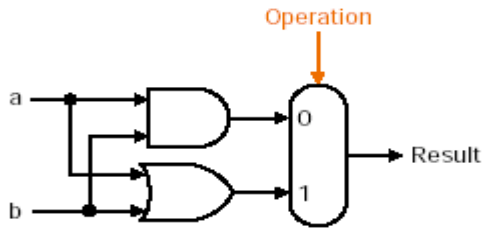


Figura c.a.2: *ALU de un bit con operaciones lógicas de (and, or)*

Este esquema elemental, con dos compuertas y un multiplexor, permite establecer algunas consideraciones extensibles a diseños más complejos. Se distinguen:

- las líneas de datos de entrada y salida (***a,b,Result***),
- la lógica combinacional que implementan las operaciones lógicas y aritméticas (en este caso las compuertas ***and***, ***or***),
- la lógica de control interno (en este caso el ***mux***)
- las líneas de control que determinan cual es la operación que debe implementar la ALU (***Operation***).

Recordar que el encargado de dar la señalización de control en la CPU es la **Unidad de Control**, por tanto directa o indirectamente estas señales provendrán de esta unidad.

Implementación de Suma en 1 – bit

La implementación de las operaciones aritméticas a este nivel implica analizar el bloque que implementa la suma en 1 – bit. La idea es que la concatenación de estos bloques permita generalizar la suma entera sin signo en n-bit. Por lo tanto, se debe llevar en consideración la señal de carry de entrada (***Carryin***), acarreo generado en la etapa de suma previa, y también la señal carry de salida (***Carryout***), cuyo destino es el bloque que implementa la suma del siguiente bit más significativo. Del punto de vista de entrada-salida este bloque se ve como sigue:

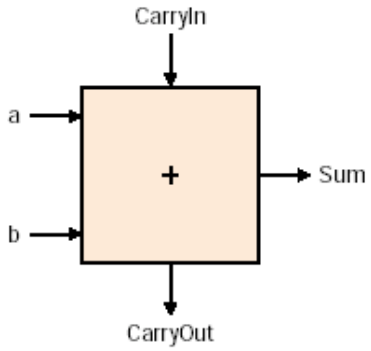


Figura c.a.3: *Entrada-salida bloque suma en 1-bit*

Siendo las entradas del bloque (a, b y CarryIn) las tablas de verdad que caracterizan las funciones lógicas de **Sum** y **Carryout** vienen dadas por:

a	b	Carryin	Carryout	Sum
0	0	0	**0**	**0**
0	0	**1**	**0**	**1**
0	**1**	0	**0**	**1**
0	**1**	**1**	**1**	**0**
1	0	0	**0**	**1**
1	0	**1**	**1**	**0**
1	**1**	0	**1**	**0**
1	**1**	**1**	**1**	**1**

Tabla c.a.1: *tabla de verdad ALU – 1 bit*

De los conocimientos adquiridos de lógica digital es sabido que las funciones lógicas que caracterizan Carryout y Sum se pueden implementar con los bloques básicos (and, or y not). En particular para ***Carryout*** su caracterización en mini- términos es:

$$\mathbf{Carryout = (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b) + (a \cdot b \cdot CarryIn)}$$

El último mini- término es verdadero cuando los otros lo son, por tanto es redundante, luego la expresión en mini-términos más compacta puede escribirse como:

$$\mathbf{Carryout = (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b)}$$

Y la implementación de esta expresión sería como se muestra en la siguiente figura.

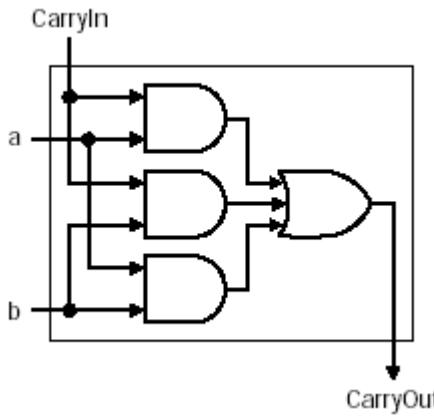


Figura c.a..4: *Diagrama de compuertas que implementan la función lógica* ***Carryout.***

De la misma manera la forma más compacta en mini términos para la función lógica ***Sum*** viene dada por:

$$Sum = (a \cdot \overline{b} \cdot \overline{C}arryIn) + (\overline{a} \cdot \overline{b} \cdot CarryIn) + (\overline{a} \cdot b \cdot \overline{C}arryIn) + (a \cdot b \cdot CarryIn)$$

Para este caso la implementación en suma de productos implica el uso de 4 compuertas **and** de 3 entradas y una compuerta ***or*** de 4 entradas. Finalmente la implementación de una ALU de 1 bit que lleva a cabo suma y las operaciones lógicas and y or, se puede ver en la ***Figura c.a.5***. En este caso el multiplexor utilizado debe ser de 4 es a 1, por lo tanto tiene dos bits de control para direccionar la salida.

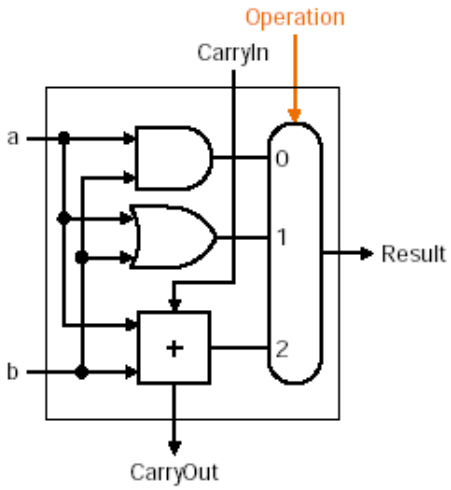


Figura c.a.5: *Alu de 1bit que implementa suma, y las operaciones lógicas de and-or.*

3.8.2 ALU de 32-bit

Es directo observar que la concatenación de estas ALU de 1 bit permite implementar una ALU con suma entera sin signo y las operaciones lógicas de and y or en n-bit. En particular, la implementación de una ALU en 32 bit se presenta en la siguiente figura:

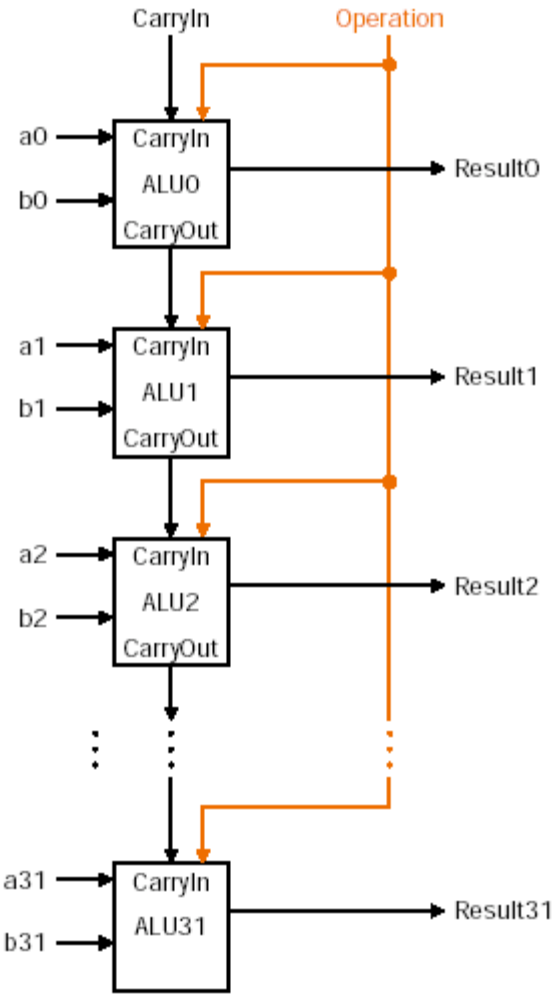


Figura c.a.6: *Alu de 32 bit que implementa suma, y las operaciones lógicas de and-or.*

En este diseño los bloques tienen una dependencia por medio de las señales CarryOut->CarryIn, para implementar la suma entera. La señal de control es la misma para cada uno de ellos (***Operation***). En esta etapa se puede ver que la implementación de las operaciones lógicas es menos costosa del punto de vista de tiempo, que las operaciones aritméticas. Si el tiempo que implica una salida estable en las compuertas elementales es T, entonces el tiempo requerido para implementar una operación lógica es T (independiente de los bits de palabra que permite la ALU). Por otro lado, el tiempo requerido para implementar una suma es (2T)xN. Este tiempo es equivalente al tiempo requerido por la última unidad aritmética de 1-bit (bit más significativo) en tener la señal ***CarryIn*** estable, más el tiempo en que este bloque genera salidas estables.

Veremos que existen técnicas de diseño lógico que permiten obtener las señales de ***CarryIn*** de los bloques de bits más significativos con anterioridad y de esta manera tener mejores tiempos de respuesta.

3.8.3 Implementación de suma – resta en complemento dos

En el capítulo anterior vimos que la aritmética sin signo, permite implementar la suma y resta complemento dos, donde sólo se debe implementar la lógica del operador de complemento (complemento booleano +1) y la lógica de control para los casos de **overflow**.

El siguiente esquema muestra el bloque elemental de una ALU de un bit que permite llevar a cabo dichas operaciones. Respecto de la ALU de suma entera sin signo, se ve que existe una señal de control adicional (**Binvert**) que es la encargada de direccionar si la salida de la operación aritmética es (a+b) o (a-b)

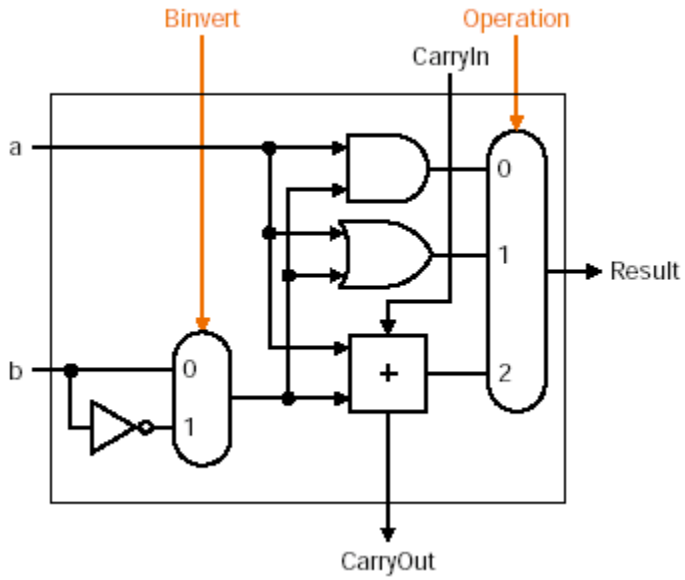


Figura c.a.7: *ALU de 1 bit para implementación de suma-resta en codificación complemento a dos.*

Al concatenar estos bloques de la forma mostrada en la ***Figura c.a.6*** es fácil ver la resta en complemento dos, en este caso (a-b), se logra vía el complemento booleano de b, lo que esta implementado en cada ALU de 1-bit y adicionalmente la señal ***CarryIn*** del bit menos significativo en 1, pues es equivalente a incrementar en 1 la suma. Específicamente, el resultado de dicho proceso sería:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a - b$$

De esta forma las señales de control de esta ALU de n-bit corresponden a las señales (Operation, Bintvert y CarryIn $_{\text{bit 0}}$). Las dos primeras tienen incidencia uniforme sobre cada bloque de 1-bit y la última incide sólo en el bloque del bit menos significativo. Se debe dejar en claro que en este análisis se ha despreciado la lógica de control para detectar los casos de **overflow**, tanto en el proceso de determinar el complemento de un número, como en la suma, escenarios ya caracterizados en nuestros estudios previos.

Dado que el último caso de **overflow** depende de los bits más significativos de los argumentos y del signo del resultado, esta lógica se puede implementar localmente en la ALU del bit más significativo como muestra la siguiente figura.

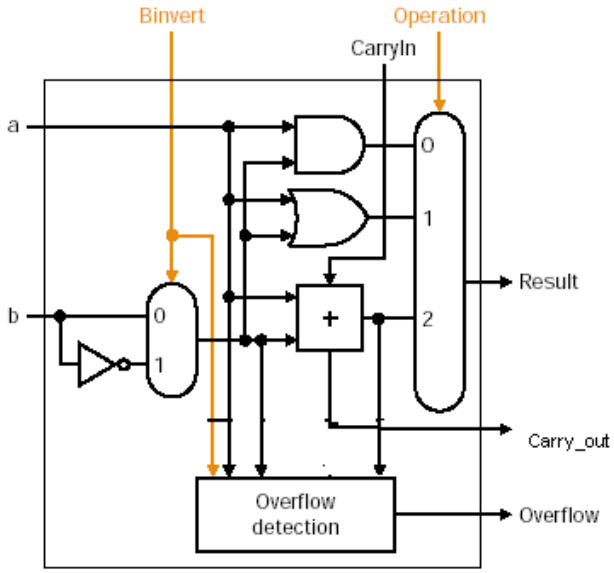


Figura c.a.8: *ALU del bit más significativo para implementación de suma-resta en codificación complemento a dos y detección de escenario de* ***overflow***. .

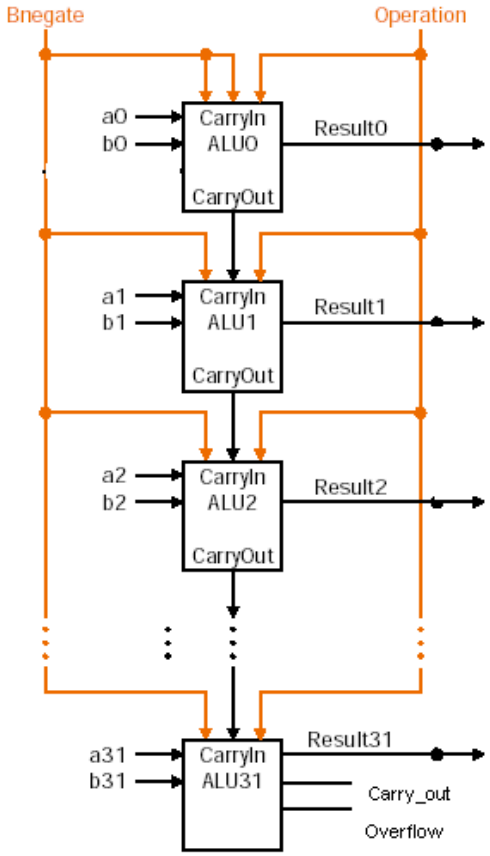


Figura c.a.9: *ALU de 32 bit para implementación de suma-resta complemento dos*

3.8.4 Implementación de la operación de comparación

Al igual que las operaciones aritméticas y lógicas básicas, las cuales están implementadas en todas las arquitecturas de CPU, las operaciones de comparación son un conjunto de instrucciones de igual importancia. Estas se utilizan para determinar la secuencia de ejecución de instrucciones en un programa. De esta forma la ALU debe tener lógica que permita implementar este tipo de operaciones.

Una manera directa de implementar una instrucción de comparación es restar los argumentos (en codificación complemento dos) y ver si el resultado es negativo o positivo. Luego la operación de comparación se reduce a implementar la operación (a-b) y sondear el bit de signo del resultado (bit más significativo en la codificación)

- Si el bit más significativo es **1**: => a<b
- Si el bit más significativo es **0**: => a>=b

Se debe notar que es la parte de la ALU del bit de signo la que posee la información para determinar esta condición (***set***), pues recibe los signos de los argumentos, determina el signo de la palabra de salida y la información del escenario de ***overflow***.

La siguiente figura muestra la ALU del bit más significativo con la señal de ***set*** que indica cuando el argumento a_0.. a_N es menor o igual que b_0.. b_N.

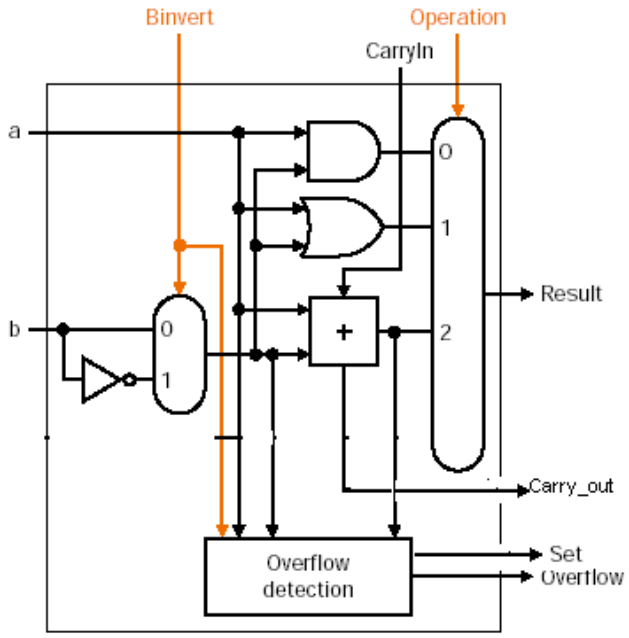


Figura c.a.10: *ALU del bit más significativo para implementación de suma-resta-comparación en codificación complemento a dos y detección de escenario de* ***overflow***. .

En primera instancia esto sería válido salvo en los escenarios de **overflow.** Sin embargo incluso en este caso es posible determinar cuál de los argumento es mayor.

(Propuesto 1) Muestre que incluso en el escenario de **overflow** con la información de los bits más significativos de los argumentos es posible determinar la señal de compasión **set**.

(Propuesto 2) Determine la lógica que implementa la unidad **overflow**.

Otra operación de comparación importante es la que permite decidir cuando los argumentos son iguales. Esto se puede llevar a cabo utilizando la operación substracción y sobre sus salidas utilizar una compuerta ***or*** de N-bit. De esta manera esta ALU tendrá un flag ***Zero***, que indica si la operación aritmética de substracción, independiente del escenario de **overflow**, tiene la codificación del cero, i.e:

$$zero = \overline{(result_{31} + result_{30} + ... + result_0)}$$

La implementación es la que muestra la siguiente figura:

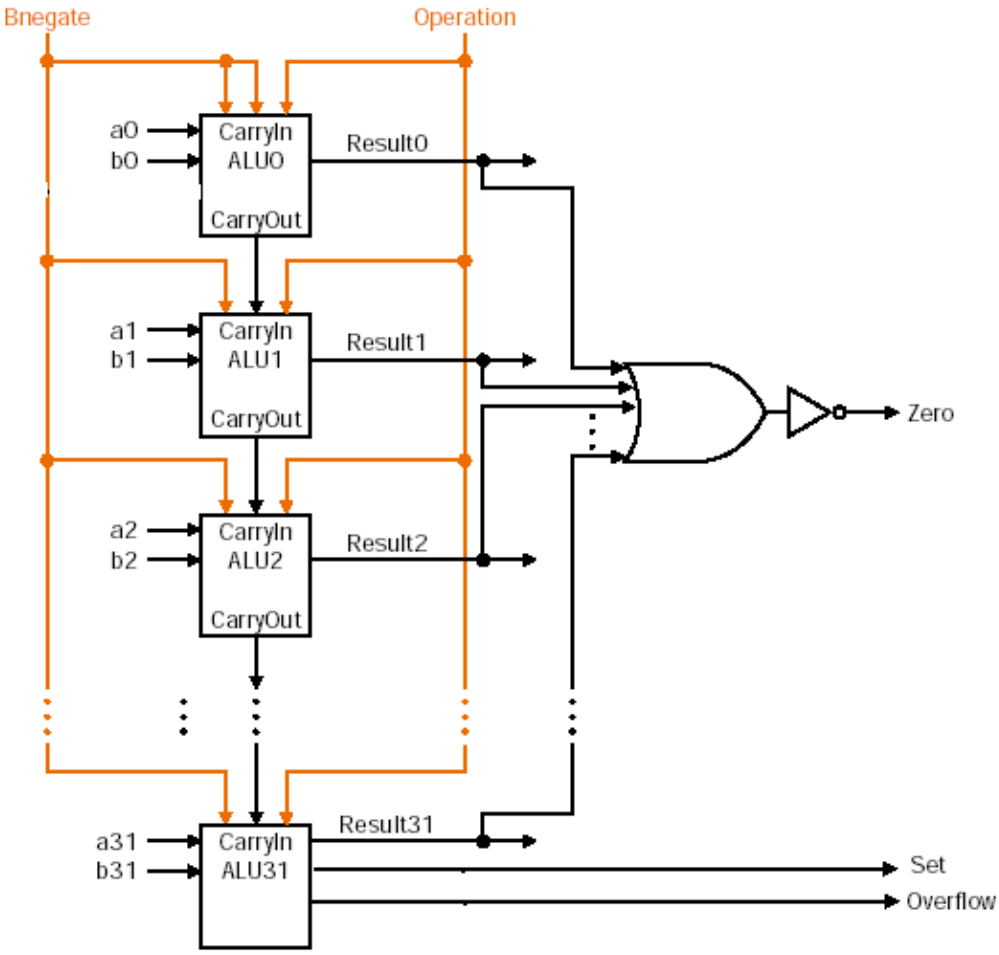


Figura c.a.11: *ALU de 32 bit para implementación de suma-resta complemento dos, operaciones lógicas y operaciones de comparación.*

Observaciones:

- Importante destacar que en ambas operaciones de comparación la *Unidad de Control*, para efectos de determinar el resultado de estas, debe sondear los bits ***Set*** y ***Zero***, que típicamente son almacenados en el registro de estado de la ALU (**STATUS register**), independiente de los bits de resultado.

- Estos bits de estado de la ALU siempre están activos independientemente de la operación aritmética lógica que lleve a cabo la ALU. Por lo tanto si el flag **zero** se activa, es un indicador que la operación resultante tiene la codificación 00000...0.

- Existen arquitecturas como la MIPS que tienen las instrucciones de comparación definidas en su conjunto de instrucciones de máquina, por lo tanto son los diseñadores del nivel de

microprogramación los que determina cuales son las operaciones internas que debe ejecutar la CPU para llevarlas a cabo. Sin embargo existen lenguajes de máquina que no las tienen definidas en su conjunto de instrucciones y por lo tanto los programadores de nivel dos deben combinar instrucciones para poder implementarlas, restar los argumentos y sondear los bits del registro de estado de la ALU.

Finalmente la ALU con las operaciones aritméticas, lógicas y de condición se puede ver del punto de vista de entrada salida como muestra la ***Figura c.a. 12***. Es importante observar que las líneas de control corresponden lógicamente a tres bits:

- 1 bit (***Bnegate***) que determina si la operación aritmética es suma o resta
- 2 bits de operación (***Operation***) que direccionan la salida global de la ALU, ver ***Figura c.a.10***.

La ***Tabla c.a. 2***, muestra los estados de las líneas de control para implementar el conjunto de instrucciones de la ALU estudiadas.

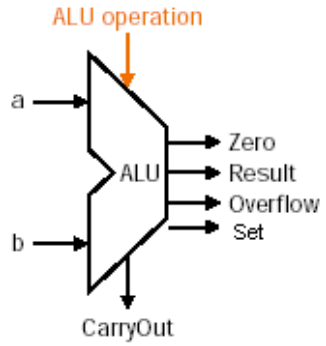


Figura c.a.12: *Diagrama de entrada salida de ALU que implementa suma-resta complemento dos, operaciones lógicas y operaciones de comparación.*

ALU control line (Bnegate,Operation)	***Instrucciones de lenguaje de máquina***
x 00	*And*
x 01	*Or*
0 10	*Add (complemento dos)*
1 10	*Subtract (complemento dos)*
1 xx	*<= (complemento dos, bit Set)*
1 xx	*== (bit Zero)*

Tabla c.a. 2: *señales de control para gobernar operaciones de ALU*

3.8.5 Optimizaciones de diseño

Vimos del diseño básico para implementar la suma ***Figura c.a.6,*** que el tiempo requerido en el peor de los casos es proporción lineal (orden lineal) al tamaño de la palabra que maneja el diseño de la ALU. Esto se debe a la dependencia de las señales de **carry**, entre las distintas ALU de 1 bit, que explicita una dependencia causal entre todas estas unidades.

La opción alternativa es utilizar técnicas de diseño para tratar de eliminar esta dependencia a costa de implementar funcionales lógicas adicionales, es decir con menos compuertas en serie para determinar las señales de **carry**. De esta manera se obtienen mejores prestaciones, pero a un mayor costo de implementación.

Fast Carry con hardware infinito

Una manera básica es determinar la función lógica de cada señal de **CarryIn** en función de las entradas a_0.. a_N , b_0.. b_N y c_o (**CarryIn** del bit menos significativo). Establezcamos por c_i la señal de **CarryIn** del bit i-ésimo, luego se tiene que:

$c_1 = (a_0 \cdot b_0) + (a_0 \cdot c_0) + (c_0 \cdot b_0)$ nivel 1

$c_2 = (a1 \cdot b_1) + (a_1 \cdot c_1) + (c_1 \cdot b_1)$ nivel 2

$c_2 = (a1 \cdot b_1) + (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) + (a_1 \cdot c_0 \cdot b_0) + (b_1 \cdot a_0 \cdot b_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot c_0 \cdot b_0)$ nivel 2

.

.

De esta forma se puede llegar a una expresión en suma de productos, que implica solo dos niveles de compuertas lógicas, pero donde el número de compuertas en cada etapa crece de manera exponencial, por lo tanto es impracticable en organizaciones de ALU de largos de palabras considerables (32 bit , 64 bit, ..)

(Propuesto 3) Determine una expresión para el número de compuertas **and** requeridas en cada etapa. Demuestre que es de la forma $O(2^{n+1})$

De esta forma se utilizan otras técnicas de diseño que permitan predecir las señales de **carry** pero sin el costo de hardware de la técnica recién mostrada.

Carry Look ahead (*propagación y generación*)

Esta técnica define los conceptos de propagación y generación en el proceso de determinar el valor lógico de las señales de ***carry*** basado en la siguiente expresión:

$$c_{i+1} = (a_i \cdot b_i) + (a_i \cdot c_i) + (c_i \cdot b_i)$$

$$c_{i+1} = (a_i \cdot b_i) + (a_i + b_i) \cdot c_i$$

Donde se define el término de propagación $p_i = (a_i + b_i)$ y generación $g_i = (a_i \cdot b_i)$ de la etapa i-esima
De esta forma la señal CarryIn de la etapa i+1 en función de los términos de propagación, generación y CarryIn de la etapa i queda como:

$$c_{i+1} = g_i + p_i \cdot c_i \qquad \textit{nivel i-ésimo}$$

Definiciones:

- Cuando el término de generación $\boldsymbol{g_i}$ se activa, expresión la señal de $\mathbf{c_{i+1}}$ queda determinada independiente de los restantes términos de la expresión. De esta forma se elimina la dependencia de los términos pasados. **(Concepto de generación**).

- Cuando no existe generación, pero la señal de propagación esta activa $\boldsymbol{p_i}$ en este escenario se manifiesta la dependencia entre $\boldsymbol{c_{i+1}}$ con $\boldsymbol{c_i}$. (**Concepto de propagación**)

De esta forma la dependencia entre los bloques $\boldsymbol{c_{i+1}}$ con $\boldsymbol{c_i}$ cuando el término de propagación está activo y los de generación está inactivo. De forma más precisa, con estas definiciones la idea es expresar las señales de **CarryIn** de cada nivel en términos de nuestras definiciones intermedias de generación y propagación.

$c_1 = g_0 + p_0 \cdot c_0$ nivel 1

$c_2 = g_1 + p_1 \cdot c_1$ nivel 2

$c_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$ nivel 2

$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$ nivel 3

.

.

(Propuesto 3) Demuestre que la generalización de la función lógica de la etapa i-ésima viene dada por la siguiente expresión:

$$c_{i+1} = g_i + \sum_{j=0}^{i-1} \left(\left(\prod_{k=0}^{j} p_{i-k} \right) \cdot g_{i-(j+1)} \right) + c_0 \cdot \prod_{k=0}^{i} p_{i-k}$$

De la expresión de la señal $\boldsymbol{c_{i+1}}$ en suma de productos de los términos de propagación y generación, se observa que esta se implementa con (i+1) términos multiplicativos (compuertas ***and)*** y se elimina la dependencia de crecimiento exponencial observada en el esquema **Fast Carry.**

(Propuesto 4) Determine el tiempo en el cual este esquema puede implementar una suma, compárelo con el escenario **Fast Carry**.

Ind: Considere un tiempo fijo T necesario para que las compuertas lógicas generen una salida estable

---

La siguiente figura muestra una representación gráfica de la implementación de estas funciones lógicas en función de los términos de propagación y generación

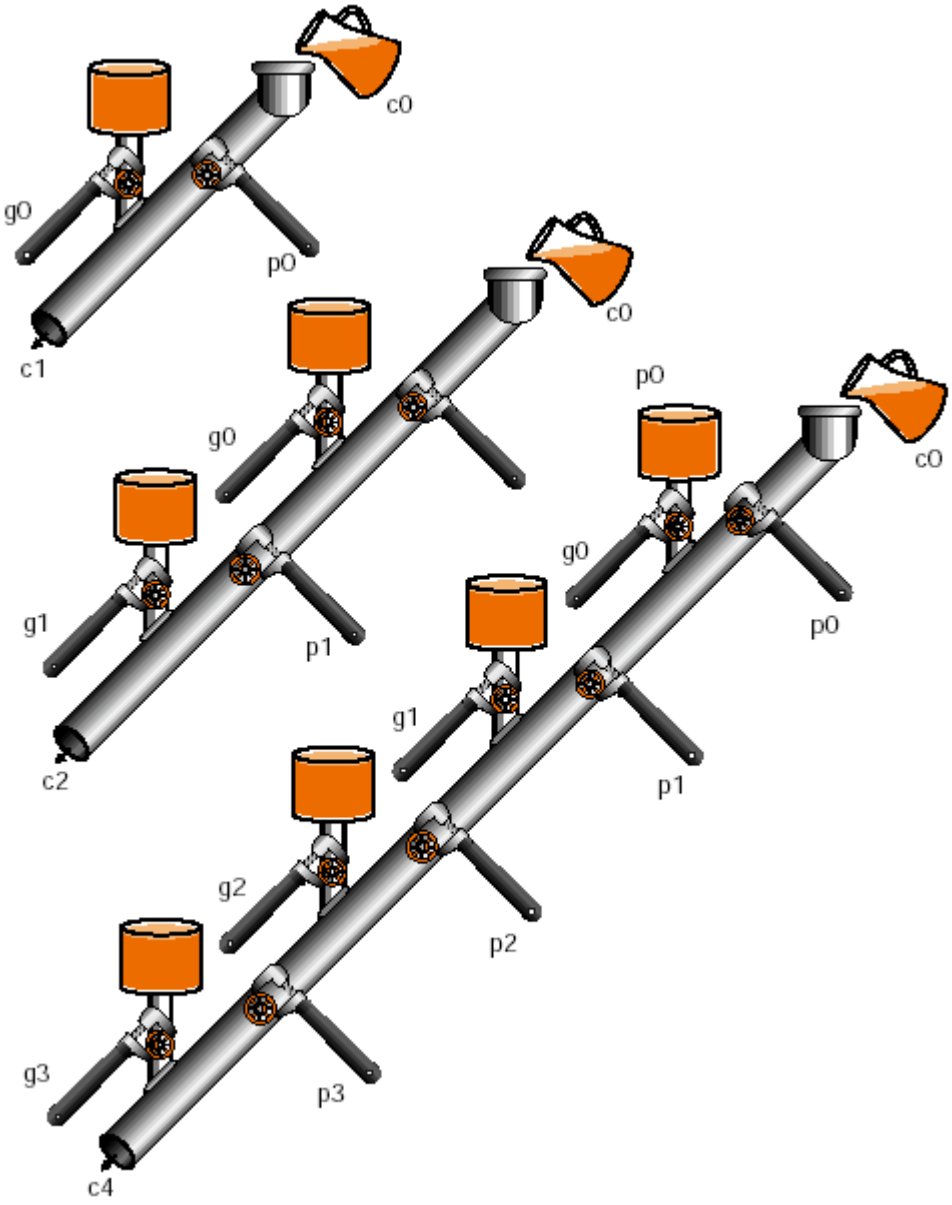


Figura c.a.13: *Representación de los conceptos de propagación y generación*