

Construcción de una Unidad Aritmético Lógica (ALU)

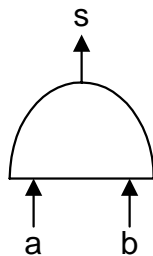
Introducción

La unidad aritmético lógica o ALU es el músculo del computador, es el dispositivo que se encarga de realizar las operaciones aritméticas (suma y resta) y las operaciones lógicas (and, or y xor).

A lo largo de este capítulo iremos construyendo la ALU paso a paso y veremos cómo es capaz de realizar todas las operaciones indicadas.

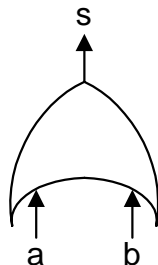
Para construir la ALU vamos a partir de bloques funcionales ya vistos, puertas AND, OR, XOR, inversores y multiplexadores, tal como aparecen en las figuras siguientes.

1. Puerta and ($s = a \cdot b$)



a	b	$s = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

2. Puerta or ($s = a + b$)



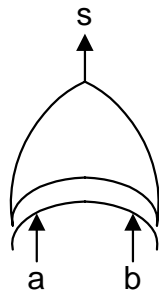
a	b	$s = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

3. Inversor ($s = \bar{a}$)



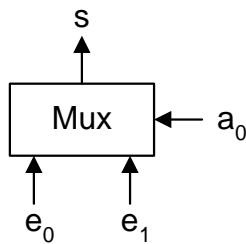
a	$s = \bar{a}$
0	1
1	0

4. Puerta xor ($s = a \oplus b$)



a	b	$s = a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

5. Multiplexor (Si $a_0 = 0$, $s := e_0$; sino, $s := e_1$)



a_0	s
0	e_0
1	e_1

Los computadores trabajan con palabras de varios bits, y por lo tanto la ALU debe ser capaz de operar con estas palabras. Por motivos de simplicidad, inicialmente vamos a diseñar una ALU de un bit para posteriormente extender el diseño al número de bits de la palabra del computador.

Construcción de una ALU de 1 bit

En primer lugar, las operaciones lógicas a realizar sobre los bits son fáciles de construir, puesto que se corresponden con puertas lógicas ya existentes. Los valores de entrada se llevan directamente a las puertas, y mediante un multiplexor elegimos la salida de una u otra puerta, es decir, elegimos el resultado de una u otra operación. En la figura 1 aparece la construcción de esta unidad de operación lógica.

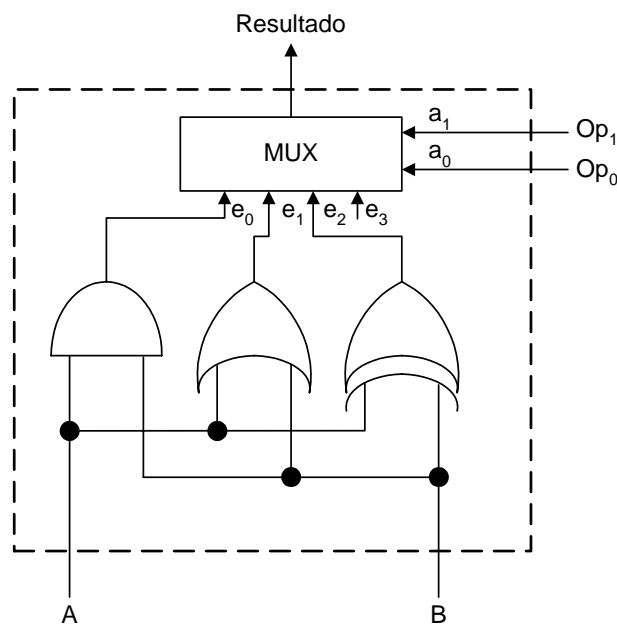


Figura 1. Implementación de las operaciones lógicas en una ALU de 1 bit.

El siguiente paso a dar en la construcción de nuestra ALU elemental de un bit será incorporar las operaciones aritméticas. Para ello utilizamos un bloque sumador, este bloque sumador ya ha sido desarrollado en el tema de circuitos combinacionales.

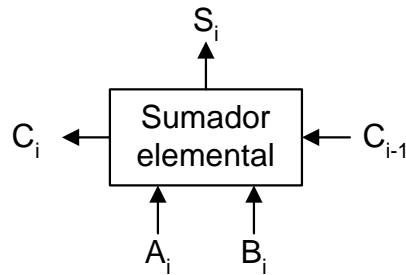


Figura 2. Esquema del bloque sumador elemental de 1 bit.

Debe recordarse que el bloque sumador tenía tres entradas, dos entradas para los bits a sumar y la tercera entrada para el carry de la suma anterior o carry previo, que aquí llamaremos **carry in**. Las salidas del bloque sumador serán el bit de resultado y el bit de carry de salida, que aquí llamaremos **carry out**, y que a su vez será *carry in* para el siguiente sumador. El bloque por el que representaremos el sumador aparece en la figura 2.

Si ahora incorporamos el bloque sumador a nuestra ALU, que sólo realizaba operaciones lógicas, tendremos una primera ALU elemental capaz de realizar operaciones lógicas y aritméticas. En esta primera ALU elemental, existen cuatro posibles operaciones, AND, OR, XOR y Suma, a partir de los mismos operandos de entrada. La elección de una u otra se realiza a través del multiplexor, la combinación binaria que aparece en las líneas de selección del multiplexor indicarán cuál de las entradas (qué operación) aparece a la salida de nuestra ALU elemental. La combinación binaria de entrada viene dada por la señal **Operación** (compuesta por dos bits) que se corresponde con la operación que queremos realizar. En la figura 3 aparece este primer esquema de ALU elemental.

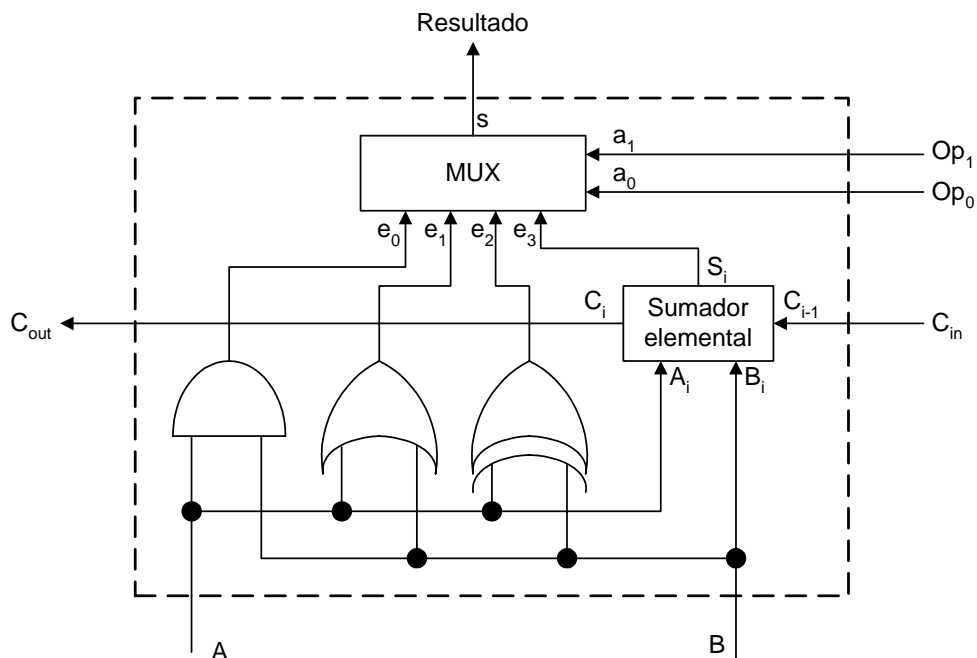


Figura 3. Esquema de una ALU elemental con operaciones lógicas y Sumador elemental.

Queda ahora por añadir a esta ALU inicial la capacidad de restar dos bits. Realizar la operación de restar es igual que sumar la versión negativa de un operando, o lo que es lo mismo: $(A - B) = A + (-B)$. Es de esta forma como los sumadores realizan la resta.

Cuando se explicó la codificación de números enteros se vieron varios formatos; uno de ellos es el denominado complemento a dos. En complemento a dos se vio que era posible representar números positivos y negativos. Los números positivos se representaban de la misma forma que en binario natural, y los números negativos se obtenían a partir de la codificación binaria del número considerado como positivo y luego complementándolo a dos, o lo que es lo mismo, invertir los bits de la codificación binaria del número (los ceros pasan a unos y los unos a ceros, esta operación a veces se denomina *complemento a uno*), y al resultado de la operación anterior sumarle 1.

Para construir la operación aritmética de resta, si en nuestro esquema de ALU elemental añadimos un nuevo multiplexor que nos permita elegir entre el operando B o su negado, tal como aparece en la figura 4, ya tendremos la mitad del camino recorrido (obtención del inverso de la combinación binaria del número a restar). La señal **Resta** controlará qué entrada del multiplexor se tomará, la normal o la invertida.

Para obtener el número negativo aún hemos de sumar 1 al número invertido. ¿Cómo se puede conseguir esto? Cada sumador elemental hemos visto que tiene tres entradas, los operandos A y B y la entrada *carry in* o carry de la suma anterior. Sin embargo, el primer sumador elemental, o el que suma los bits menos significativos, no necesitaría la señal *carry in*, puesto que no existe ninguna suma anterior. ¿Qué ocurre si la señal *carry in* se coloca a 1 en el primer sumador elemental y se selecciona la versión negada del operando B ? Tendremos:

$$A + \bar{B} + 1 = A + (\bar{B} + 1) = A + (-B) = A - B$$

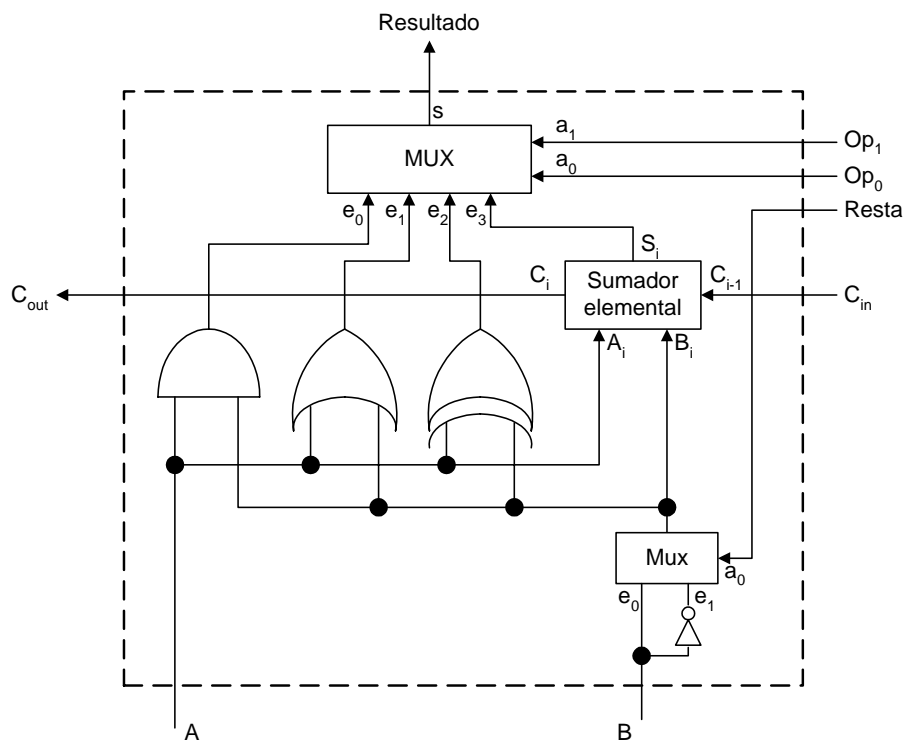


Figura 4 . Esquema de una ALU elemental capaz de realizar la resta de operandos.

Por tanto, con este esquema seremos capaces de realizar la resta de dos números. Observar que no se ha aumentado excesivamente la complejidad de la ALU elemental para realizar las operaciones de suma y resta, y es por este motivo que se utiliza el complemento a dos como estándar para la aritmética entera en los computadores.

Con esto finalizamos la construcción de la ALU elemental, que servirá de base para la construcción de una ALU completa. A partir de ahora representaremos la ALU elemental que hemos desarrollado como una "caja negra" como la mostrada en la figura 5.

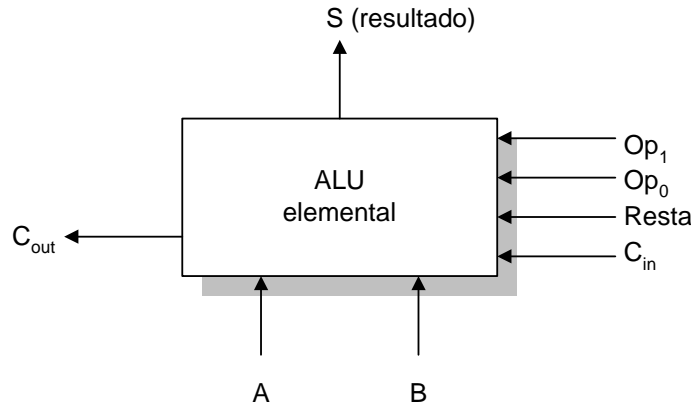


Figura 5. Esquema de una ALU elemental como "caja negra"

Confección de una ALU de 16 bits

Hasta ahora, hemos desarrollado una ALU elemental que nos permite trabajar con operandos de 1 bit. Construiremos una ALU de 16 bits conectando 16 de estas ALUs elementales, de forma que, cada una de ellas, opere con uno de los 16 bits del operando. El esquema general de conexión aparece en la figura 6.

En la figura 6 se puede apreciar el aspecto general de la ALU construida para la CPU de ejemplo. Sin embargo, éste aún no es el esquema definitivo. La ALU debe generar ciertas señales indicativas del resultado de la última operación realizada. Estas señales son los *Flags*.

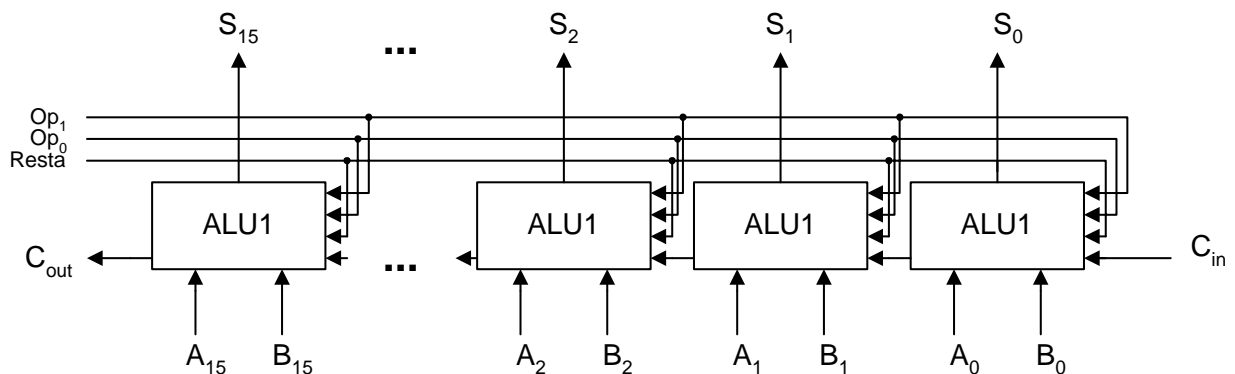


Figura 6. Esquema global de una ALU de 16 bits a partir de 16 ALUs de un bit.

Los *Flags* que debe generar son cuatro:

- *Flag de Zero*. Vale 1 cuando el resultado es igual a cero.
- *Flag de Carry*. Vale 1 cuando se produce desbordamiento interpretando los operandos y el resultado como naturales.
- *Flag de Overflow*. Vale 1 cuando se produce desbordamiento interpretando los operandos y el resultado como enteros en complemento a 2.
- *Flag de Signo*. Vale 1 cuando el resultado es negativo interpretado como un entero en complemento a 2.

Veremos cómo nuestra ALU genera cada uno de ellos.

Cuando se está realizando una suma, el *Flag de Carry* se corresponde con la señal **Carry Out** de la ALU elemental que recibe los bits de mayor peso de los operandos. Sin embargo, cuando se está realizando una resta, se debe invertir esta señal para obtener el *Flag de Carry* correcto. Esto

es debido a que se realiza una resta a través de una suma y un valor en complemento a 2. Para inventir la señal de **Carry Out** señal en función de la señal resta se utiliza una puerta XOR.

Para calcular tanto el *Flag de signo* como el *Flag de overflow*, debe recordarse que en aritmética con números enteros representados en complemento a dos, el signo del operando viene indicado por el bit más significativo de la representación binaria. Por tanto, el signo del resultado será el bit más significativo del resultado; será negativo cuando este bit valga 1 y positivo cuando valga 0. Tomando el valor de la salida de la última ALU elemental, aquella que opera con los bits más significativos, tendremos el *Flag de signo*.

Cuando se estudió la aritmética en complemento a dos, se analizaron los casos en los que se producía overflow. Los casos detectados correspondían a contradicciones entre el signo de los operandos y el signo del resultado, es decir, cuando la suma de dos números positivos generaba un número negativo, y viceversa, cuando la suma de dos números negativos generaba un número positivo. De la misma forma que se estudió el caso del problema del Overflow o desbordamiento en la suma se puede realizar su estudio en la resta. ¿Cuándo existirá overflow o contradicción de signos? Los casos en los que existe contradicción en la resta es cuando a un número positivo se le resta un número negativo y el resultado es negativo, (“+” - (“-”) = “-”). El otro caso posible que se puede producir es cuando a un número negativo se le resta un número positivo y el resultado es positivo (“-” - (“+”) = “+”). Los casos posibles aparecen en la tabla 1.

La señal **Resta** en la primera columna indica el tipo de operación a realizar (0 para la suma y 1 para la resta), las tres columnas siguientes son los signos del operando A, operando B y del resultado. La última columna corresponde a la detección del Overflow.

En nuestra ALU, aún queda por detectar el *Flag de cero*, es decir, un bit que se pone a 1 cuando el resultado de la operación es cero, y a 0 en otro caso. La forma más sencilla de realizar esta detección es comprobar el valor del resultado de cada ALU elemental, y si todos los resultados de las ALUs elementales son nulos, el *Flag de cero* debe ponerse a 1. Esto se realizará sacando una señal de cada resultado y llevándola a una puerta OR de tantas entradas como ALUs elementales existan, el resultado de la puerta OR se negará (o se usará una puerta NOR), con ello tendremos una señal que sólo será 1 cuando todas las entradas sean cero, es decir, se ha detectado el *Flag de cero*.

En la figura 7, puede verse como se generan los distintos *Flags* dentro de la ALU formada por 16 ALUs elementales. Destacar cómo se ha obtenido el *Flag de overflow*, a partir de los bits más significativos de los operandos y del resultado (signos) y con la señal **Resta**, se construiría un circuito detector de overflow que implementará la tabla 1.

Tabla 1 Casos posibles de Overflow o desbordamiento.

Resta	Signo A	Signo B	Signo Resultado	Overflow
0	0	0	0	No
0	0	0	1	Si
0	0	1	0	No
0	0	1	1	No
0	1	0	0	No
0	1	0	1	No
0	1	1	0	Si
0	1	1	1	No
1	0	0	0	No
1	0	0	1	No

1	0	1	0	No
1	0	1	1	Si
1	1	0	0	Si
1	1	0	1	No
1	1	1	0	No
1	1	1	1	No

La figura 7 corresponde a la visión de componentes general de la ALU de 16 bits. En esta figura hemos introducido además una simplificación en el montaje. La simplificación es la siguiente: Cuando deseábamos restar dos números, en el multiplexor del operando *B* elegíamos la entrada negada, la señal de **Resta** debería valer 1. El segundo paso a realizar, consistiría en colocar un 1 en la señal de *carry in* del primer sumador elemental. Pues bien se aprovecha esta circunstancia y se conecta el *carry in* del primer sumador elemental a la señal de resta. Así, cuando haya que restar se realizarán automáticamente los dos pasos de obtención del número negativo en complemento a dos: inversión de bits y suma de 1. En el caso de números positivos la señal **Resta** vale cero, y no influirá en el valor del bit *carry in*.

Colocando una puerta OR en la entrada *carry in* de la primera ALU elemental, se podrá también colocar la entrada *carry in* a 1 aunque la señal **Resta** sea cero. Por ejemplo, para realizar un incremento de una unidad en un número, habrá que sumarle una unidad al número, lo cuál puede hacerse poniéndole el *carry in* o *carry* previo de la primera ALU elemental a 1.

La ALU diseñada formará parte de una CPU teórica sobre la cual se centrará una parte importante de la asignatura. El esquema con el que se suele representar la ALU dentro de la CPU aparece en la figura 8.

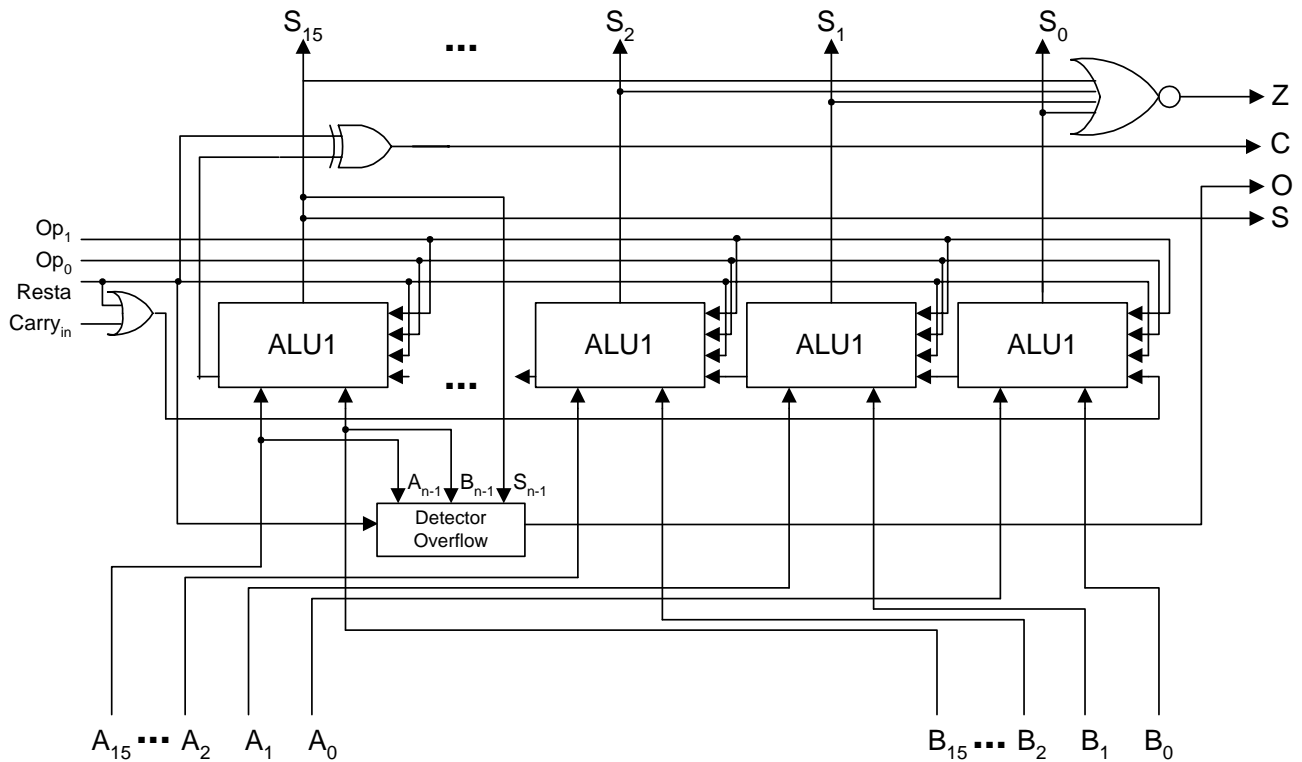


Figura 7. Esquema final de la ALU de 16 bits para la CPU teórica a partir de 16 ALUs de un bit.

Problemas de la ALU

La ALU realizada para la CPU teórica es sencilla y sirve para realizar las operaciones generales aritméticas y lógicas. Existen otro tipo de operaciones a realizar en una CPU real que complicarían su construcción.

Un problema que podemos observar es lo que se denomina *propagación del arrastre* o *propagación del carry*. En la figura 7 aparece el esquema final de la ALU, en el caso de las operaciones aritméticas, puede observarse que aunque todos los bits de las ALUs elementales entran a la vez, no todos los resultados salen a la vez. Así, saldrá en primer lugar el resultado 0; el resultado 1 dependerá de sus operandos y además del posible carry generado al obtener el resultado 0, y así sucesivamente. Se produce una dependencia de los resultados anteriores. El resultado 15, deberá esperar a que se hayan obtenido los resultados 0 a 14, por lo que se produce un retardo.

Para solucionar este problema se complica la circuitería de la ALU con un circuito denominado *anticipación del arrastre*. Este circuito realiza algunas operaciones lógicas con los operandos de forma que pueda reducir el tiempo que tarda en obtenerse el último bit del resultado. Esto no se verá aquí.

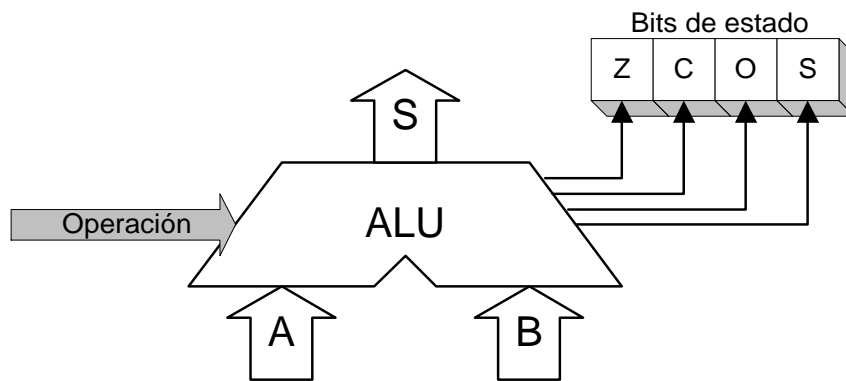


Figura 8. Esquema de representación de la ALU dentro del conjunto de la CPU.