ESTRUCTURA Y TECNOLOGÍA DE COMPUTADORES

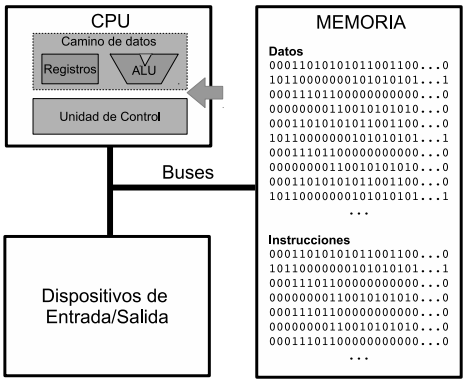
Departamento de Ingeniería y Tecnología de Computadores

Febrero 2021

ÍNDICE GENERAL

CAPÍTULO 1

COMPONENTES COMBINACIONALES DE UN PROCESADOR



El procesador (CPU, *central processing unit*) de un computador es un bloque lógico complejo encargado de la ejecución de las instrucciones de un programa escrito de acuerdo al juego de instrucciones (ISA, *instruction set architecture*) que este implementa. La construcción de un procesador puede abordarse a partir de una serie de bloques lógicos combinacionales y secuenciales más sencillos. En este tema se explica el diseño y funcionamiento de los bloques lógicos combinacionales que serán utilizados como componentes para diseñar el procesador del tema 3. Algunos de estos componentes ya se han estudiado en asignaturas anteriores, y se incluyen aquí como recordatorio y a modo de referencia. En un tema posterior se explicarán otros componentes que no son combinacionales, sino secuenciales (es decir: con memoria).

Recordemos en primer lugar que la característica principal de un circuito combinacional es que su salida en un instante determinado depende exclusivamente del valor que las entradas tienen en este instante. En otras palabras, siempre que repitamos los mismos valores para las entradas en un circuito combinacional, vamos a obtener los mismos valores en las salidas. En la construcción del procesador que estudiaremos en el tema siguiente (y en la de algunos de sus componentes que analizamos en este tema) van a estar involucrados diversos circuitos combinacionales, que mostramos a continuación.

1.1 DECODIFICADOR

Al contrario que un codificador, un *decodificador* es un circuito lógico combinacional con n líneas de entrada y 2^n líneas de salida, tal y como se muestra en la figura 1.1.

En este caso, interesa que para cada posible valor de las líneas de entrada, una y sólo una de las señales de salida tenga el valor lógico 1. Por tanto, podemos considerar el

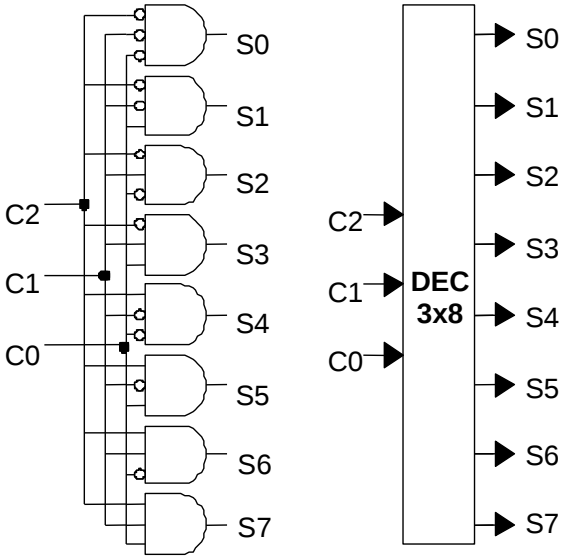


Figura 1.1: Diagrama funcional de un decodificador 3 a 8.

C2	C1	C0	S0	S1	S2	S3	S4	S5	S6	S7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Tabla 1.1: Tabla de verdad de un decodificador de 3 a 8 bits.

decodificador n a 2^n como un generador de minitérminos, donde cada salida corresponde precisamente a un minitérmino diferente. Si en la entrada, por ejemplo, colocamos la secuencia 110, entonces S6 valdrá 1, mientras que el resto de salidas valdrá 0. Al igual que ocurría con los codificadores, la extensión a mayor número de entradas es inmediata, mediante la simple adición de una puerta AND por cada salida n, a la cual se conectan todas y cada una de las entradas C_i, negadas si al escribir n en binario el bit i es 0, y sin negar en caso contrario.

Los decodificadores se usan para tareas tales como seleccionar una palabra de memoria, dada su dirección (lo veremos en el tema siguiente) o convertir códigos, por ejemplo, de binario a decimal. La tabla 1.1 resume el funcionamiento del anterior decodificador.

1.2 MULTIPLEXOR

Un multiplexor, también llamado selector de datos, es un dispositivo que selecciona una de varias líneas de entrada para que aparezca en una única línea de salida. La entrada de datos que se selecciona viene indicada por unas entradas especiales, llamadas de control. La figura 1.2 (izquierda) muestra el esquema del multiplexor más sencillo, de 2x1 (dos entradas y una salida), con dos entradas de datos y una de control, que selecciona cuál de las dos entradas de datos queremos que se transfiera a la salida. Siguiendo

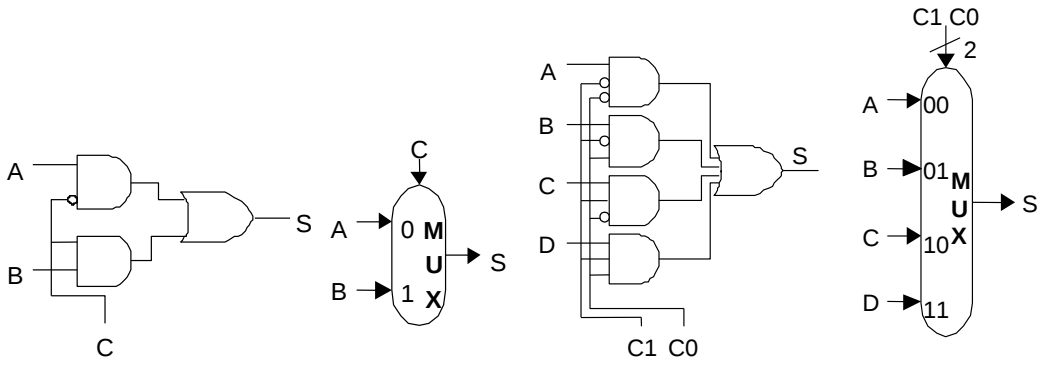


Figura 1.2: Implementaciones y bloques lógicos de dos tipos de multiplexores de un bit de ancho: (Izquierda) Multiplexor 2x1, con entradas de datos A y B y de control C. (Derecha) Multiplexor 4x1, con entradas de datos A, B, C y D y de control C1 y C0.

la misma técnica que con los decodificadores, es fácil extender los multiplexores para mayor número de entradas. La figura 1.2 (derecha) muestra un multiplexor 4x1.

En general, en un multiplexor con n líneas de entrada y una única salida, se determina cuál de las líneas de entrada $(D_{n-1}D_{n-2}D_{n-3}...D_1D_0)$ se conecta a la única línea de salida (Y) mediante un código de selección $(S_{k-1}S_{k-2}S_{k-3}\cdots S_1S_0)$ donde necesariamente $n = 2^k$. Así, por ejemplo, para un multiplexor con $n = 4$ (y por tanto $k = 2$), la ecuación lógica de la salida Y queda:

$$Y = (S1' \cdot S0') \cdot D0 + (S1' \cdot S0) \cdot D1 + (S1 \cdot S0') \cdot D2 + (S1 \cdot S0) \cdot D3$$

Además del número de entradas, otra variación importante de los multiplexores es en la anchura de los datos seleccionados. Así, en general hablaremos de un multiplexor de $2^n x1$ de m bits de anchura, si es capaz de seleccionar entre 2^n palabras de m bits cada una en función de una señal de control de n bits. En la figura 1.3 se muestra cómo crear un multiplexor 2x1 de palabras de 32 bits, a partir de multiplexores elementales de 1 bit. Usando esta misma técnica pueden conseguirse multiplexores con la anchura y el número de entradas deseadas.

Los multiplexores van a tener un papel destacado en la construcción del procesador. Mediante la utilización de estos elementos conseguiremos que una misma unidad funcional pueda manejar a través de sus entradas diferentes elementos de datos, en función de la instrucción a ejecutar. Por ejemplo, a través de un multiplexor podremos seleccionar si la dirección de memoria a leer se corresponde con la de una instrucción, o por el contrario, es la de un dato.

1.3 DESPLAZADORES

Un desplazador de bits, como su propio nombre indica, es un circuito combinacional que recibe como entrada una palabra de n bits y devuelve a la salida otra palabra de n bits que es el resultado de desplazar a la izquierda o a la derecha (dependiendo de la implementación concreta) un determinado número de bits (que suele ser fijo) la palabra de entrada. Entre otras cosas, este tipo de circuitos suelen ser empleados para la realización de multiplicaciones (desplazador a la izquierda) y divisiones (desplazador a la derecha) rápidas cuando el multiplicando y el divisor respectivamente es un número potencia de 2.

Aunque en la mayoría de los casos el tamaño del desplazamiento es fijo, también es posible construir desplazadores variables utilizando multiplexores. En ese caso, además

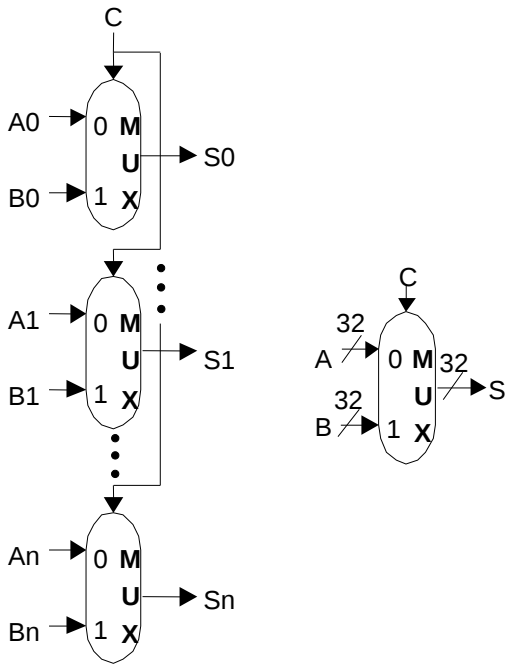


Figura 1.3: Implementación de un MUX 2x1 para palabras de 32 bits y bloque lógico correspondiente. Obsérvese que es necesario sólo un bit de control para el multiplexor.

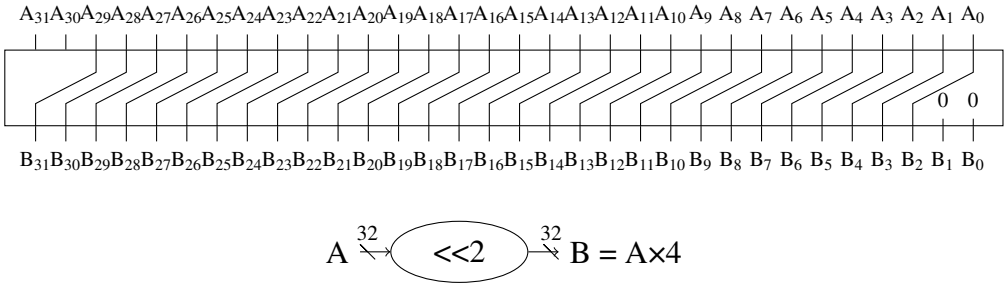


Figura 1.4: Esquema de implementación del desplazador izquierda de 2 bits y bloque lógico.

de la entrada de datos de n bits, también habría una entrada de control para especificar el desplazmiento que se desea en cada momento.

En el caso del procesador que diseñaremos en el tema siguiente vamos a emplear desplazadores a la izquierda fijos de 2 bits y 16 bits. El primero se usará para realizar multiplicaciones por 4 de forma rápida y su implementación y el bloque lógico correspondiente se muestran en la figura 1.4. El desplazador a la izquierda de 16 bits se empleará para poder cargar una constante de 16 bits en la mitad izquierda (los bits más significativos) de un registro, y su implementación sería análoga a la del desplazador de 2 bits, tal y como se muestra en la figura 1.5.

1.4 EXTENSORES

En algunas ocasiones nos podemos encontrar con que uno de los operandos con los que queremos que trabaje una unidad funcional tiene menos bits que los que maneja la entrada de la unidad funcional a través de la cual se lo hemos de proporcionar. En estos casos, si la operación a realizar por parte de la unidad funcional es una operación lógica (por ejemplo, AND) o una operación aritmética con enteros sin signo, los bits

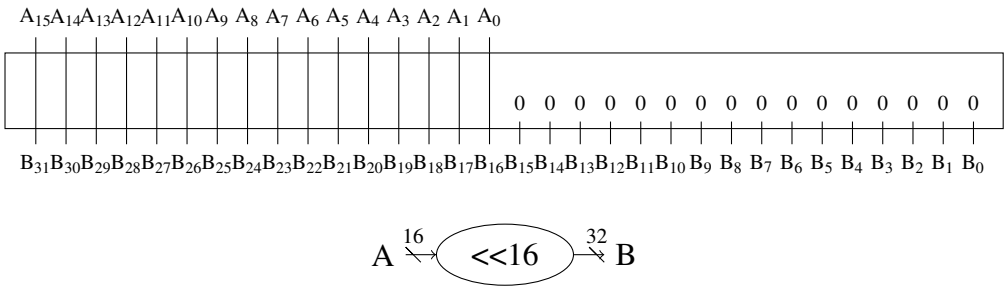


Figura 1.5: Esquema de implementación del desplazador izquierda de 16 bits y bloque lógico.

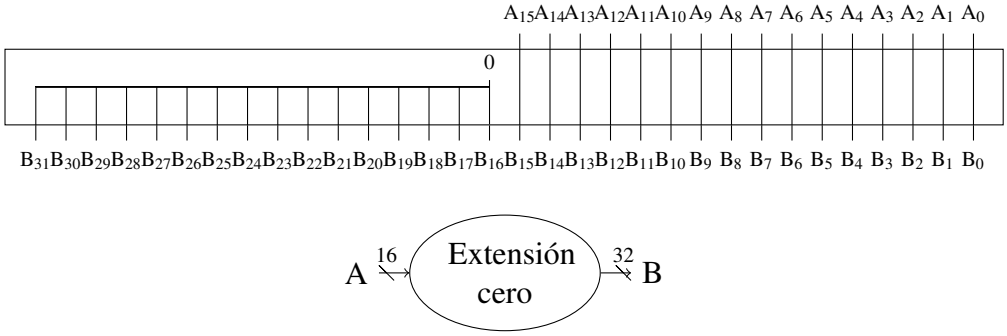


Figura 1.6: Esquema de implementación del extensor de cero y bloque lógico.

más significativos que faltan hasta completar el número total de bits que tiene la entrada se fijan a 0. A los circuitos combinacionales que llevan a cabo esta operación se les denomina extensores de cero. Por el contrario, cuando la operación a realizar es una operación aritmética de enteros con signo (que como ya se ha visto en la asignatura de *Fundamentos de Computadores*, se representan en complemento a 2) los bits más significativos restantes se rellenan replicando el bit de signo del operando en cuestión. Para la realización de esta última operación emplearemos otros circuitos combinacionales denominados extensores de signo.

Un *extensor de cero* recibe como entrada un patrón de m bits y genera como salida el patrón de entrada representado en n bits, donde $n > m$. El extensor de cero asignará a los $n - m$ bits más significativos de la salida el valor «0». La figura 1.6 muestra el bloque lógico correspondiente a una unidad de extensión de cero y su implementación.

Por su parte, un *extensor de signo* recibe también como entrada un patrón de m bits, que esta vez se interpretará como un número entero con signo. La salida del extensor de signo será el operando de entrada representado en n bits, donde $n > m$. El extensor de signo replicará en los $n - m$ bits más significativos de la salida el bit de signo del operando de entrada (bit $m - 1$, numerando los bits a partir del 0). La figura 1.7 muestra el bloque lógico correspondiente a una unidad de extensión de signo y su implementación.

1.5 UNIDAD ARITMÉTICO LÓGICA

Una *ALU* (de *Arithmetic Logic Unit*) es un circuito capaz de realizar operaciones aritméticas y lógicas sobre dos operandos de entrada, para producir como resultado el correspondiente operando de salida. La operación concreta a realizar vendrá dada por unos bits de control, con los que indicaremos a la ALU si se desea realizar una suma, una resta, un AND lógico, etc. Hay que distinguir entre ALUs enteras y ALUs de punto

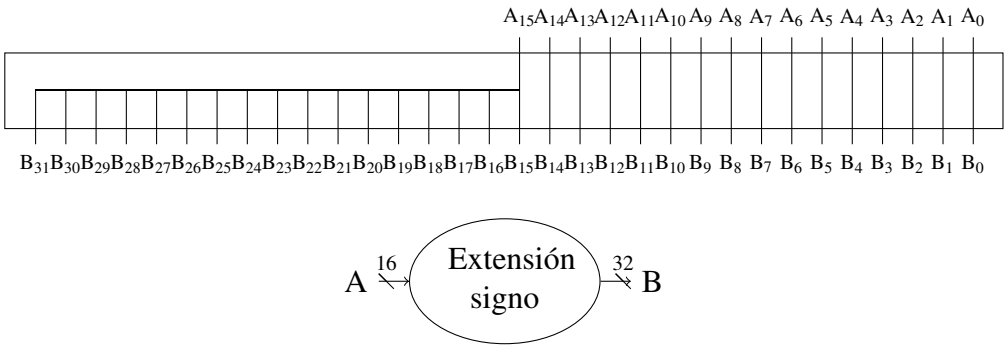


Figura 1.7: Esquema de implementación del extensor de signo y bloque lógico.

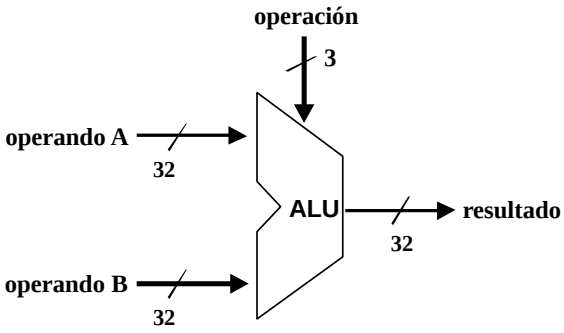


Figura 1.8: Bloque lógico de una ALU de 32 bits.

flotante, dependiendo del tipo de datos con los que trabaje el circuito. En este apartado estudiaremos únicamente el diseño de una sencilla ALU combinacional para trabajar con números enteros.

Cada uno de los operandos será una ristra de bits de una determinada longitud. En nuestro caso, trabajaremos con palabras de 32 bits, tamaño de palabra que utiliza el procesador MIPS que construiremos en el tema siguiente. El bloque lógico de esta unidad es el mostrado en la figura 1.8.

En lo que sigue diseñaremos esta unidad hasta llegar al nivel de puertas lógicas elementales.

1.5.1 Operaciones aritméticas

La ALU sencilla que vamos a diseñar ha ser capaz de realizar operaciones de suma y resta sobre enteros con signo. Operaciones más complejas como la multiplicación, la división y las operaciones de punto flotante quedan fuera de lo que cubriremos en la asignatura.

De entre las diferentes representaciones posibles para los enteros con signo ya estudiadas en la asignatura de *Fundamentos de Computadores*, la representación en complemento a 2 (C2) posee ciertas características que la convierten en la más adecuada a la hora de utilizarla internamente para la realización de las operaciones aritméticas en un computador. Por un lado, la representación en complemento a 2 evita el problema de la doble representación del cero que sufren algunas alternativas. Sin embargo, lo más importante es que permite el cálculo del opuesto de un número (cambio de signo) de manera muy sencilla en hardware, lo que permite realizar las restas como si fueran sumas después de cambiarle el signo al sustraendo. Esta última propiedad hace que con la misma circuitería capaz de realizar sumas se puedan hacer también las restas, simplificando el diseño de la ALU. Además, la detección de las condiciones de desbordamiento

es muy sencilla utilizando esta representación, ya que sólo es necesario comprobar si el signo del resultado es el esperado.

Como no podía ser de otro modo, nuestra ALU va a trabajar con enteros con signo representados en complemento a 2. En este formato de representación, el bit más significativo de un número indica su signo (bit de signo). Cuando el número es positivo este bit vale 0, mientras que cuando el número es negativo el bit de signo toma valor 1 y el resto de bits almacenan el valor absoluto del número complementado. Con los 32 bits que usará nuestra ALU para cada uno de sus operandos, el rango de números representables es desde -2^{31} hasta $2^{31}-1$. Es importante recordar que este intervalo de representación es asimétrico (existe un número negativo sin su correspondiente positivo). Considerando x_i el bit iésimo de un registro de 32 bits, el valor decimal de una representación binaria en complementado a dos es:

$$-x_{31} \times 2^{31} + x_{30} \times 2^{30} + x_{29} \times 2^{29} + x_{28} \times 2^{28} + ... + x_1 \times 2^1 + x_0 \times 2^0$$

Algunos ejemplos de representación binaria en complemento a 2 son:

Representación (en binario)		**Valor** (en decimal)		
0000 0000 0000 0000 0000 0000 0000 0000	=	0		
0000 0000 0000 0000 0000 0000 0000 0001	=	$+1$		
0000 0000 0000 0000 0000 0000 0000 0010	=	$+2$		
0111 1111 1111 1111 1111 1111 1111 1110	=	$+2^{31}-2$	=	$+2147483646$
0111 1111 1111 1111 1111 1111 1111 1111	=	$+2^{31}-1$	=	$+2147483647$
1000 0000 0000 0000 0000 0000 0000 0000	=	-2^{31}	=	-2147483648
1000 0000 0000 0000 0000 0000 0000 0001	=	$-2^{31}+1$	=	-2147483647
1111 1111 1111 1111 1111 1111 1111 1110	=	-2		
1111 1111 1111 1111 1111 1111 1111 1111	=	-1		

En complemento a 2 es muy sencillo realizar las siguientes operaciones:

- **Negar un número**: para pasar un número de positivo a negativo o viceversa, basta con cambiar todos los unos por ceros y los ceros por unos, y sumar uno al resultado. Por ejemplo, el $+24)_{10}$ representado con 8 bits sería: 0001 1000. Para negarlo invertimos todos los bits y sumamos 1: $11100111 + 1 = 11101000 = -24)_{10}$.

- **Extensión de signo**: como ya hemos visto, para aumentar el número de bits con los que se representa un número binario en complemento a 2, basta con replicar el bit de signo tantas veces como bits adicionales sean requeridos. De esta forma, el valor del nuevo número es igual al del número con menos bits. Por ejemplo, el número $+24)_{10}$ que antes representábamos con 8 bits, podemos representarlo ahora con 16 simplemente replicando el bit de signo en los 8 bits más significativos: $0000\,0000\,0001\,1000)_2$. De igual manera se haría para el $-24)_{10}$: $1111\,1111\,1110\,1000)_2$.

Sin embargo, como ya hemos explicado, la principal ventaja de la representación en complemento a 2 es que permite realizar las restas como sumas. En concreto, para realizar una resta basta con sumar el minuendo y el opuesto del sustraendo, despreciando el acarreo de salida. Así por ejemplo, para calcular el resultado de $24)_{10} - 8_{10}$, asumiendo que ambos números se representan con 8 bits, se calculará el opuesto de $8)_{10}$ que es $1111\,1000)_2$ y se realizará la suma: $0001\,1000 + 1111\,1000 = 0001\,0000 = 16)_{10}$

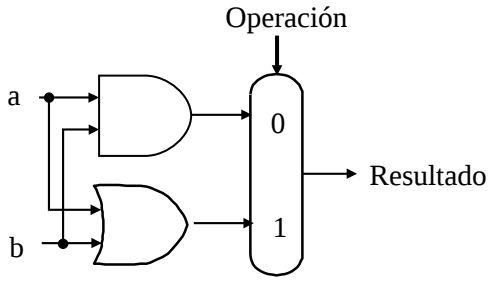


Figura 1.9: Unidad lógica de 1 bit con las operaciones lógicas AND y OR.

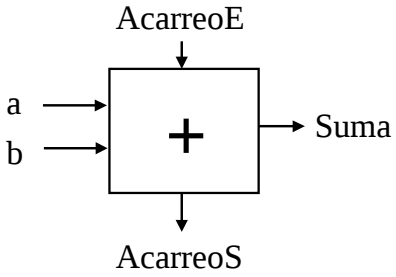


Figura 1.10: Representación de un sumador completo de 1 bit.

1.5.2 Operaciones lógicas

Las operaciones lógicas más destacadas son la disyunción o suma lógica (OR) y la conjunción o producto lógico (AND). Estas operaciones se realizan bit a bit, y son las que implementaremos en nuestra ALU. Otras operaciones lógicas habituales en los procesadores actuales son las de desplazamiento de bits, la operación XOR, la NOT, etc, si bien no serán consideradas en la sencilla ALU que vamos a construir.

Un ejemplo de operaciones lógicas bit a bit para dos palabras A y B de 32 bits sería el siguiente:

A	=	0000 1111 0011 0011 1010 0011 1111 1111
B	=	0011 0011 1111 1111 1111 1111 1111 1100
A OR B	=	0011 1111 1111 1111 1111 1111 1111 1111
A AND B	=	0000 0011 0011 0011 1010 0011 1111 1100

1.5.3 Implementación de una ALU de 1 bit

En lo que sigue, diseñaremos una Unidad Aritmético Lógica de un bit utilizando para ello puertas lógicas elementales. Empezaremos implementando para ello las funciones lógicas AND y OR que se implementarían como se indica en la figura 1.9.

La línea más gruesa corresponde a una señal de control, la cual sirve para seleccionar en el multiplexor el resultado de la operación AND u OR, en función del valor de *Operación*. En este ejemplo, si *Operación* vale 0 se seleccionará la salida de la puerta AND, y si vale 1 la salida de la puerta OR.

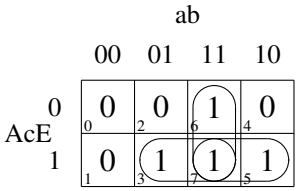
Con lo anterior, nuestra ALU sólo realizaría operaciones lógicas. Vamos a aumentarla con la operación aritmética para la suma (como veremos, la misma circuitería servirá también para restar, usando el convenio de complemento a dos). El diagrama de bloques de un sumador de 1 bit tendría el aspecto indicado en la figura 1.10.

Este sumador, también conocido como *full adder* (sumador completo) tiene tres entradas, las dos correspondientes a los sumandos y una correspondiente al acarreo de salida del sumador anterior. Las dos salidas corresponden al resultado de la suma y al

posible acarreo de salida que pudiera tener. Un sumador con sólo dos entradas y una salida se conocería como *half adder* (semisumador). La relación entre las entradas y las salidas de un sumador total está especificado en siguiente tabla de verdad:

Entradas			**Salidas**		**Función**
A	**B**	**AcE**	**AcS**	**Suma**	
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

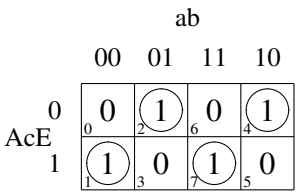
La función de salida AcarreoS (AcS) puede simplificarse con un sencillo mapa de Karnaugh:



Y la función lógica, ya simplificada, quedaría pues así:

$$AcS = (b \cdot AcE) + (a \cdot AcE) + (a \cdot b)$$

En cuanto al bit de Suma, se activa cuando una y sólo una de las entradas valen 1 o cuando las tres valen 1, y su mapa de Karnaugh quedaría así:



Puesto que los grupos de unos no pueden hacerse más grandes, la ecuación booleana correspondiente sería:

$$Suma = (a' \cdot b' \cdot AcE) + (a' \cdot b \cdot AcE') + (a \cdot b' \cdot AcE') + (a \cdot b \cdot AcE)$$

El circuito lógico que implementa ambas funciones quedaría como se indica en la figura 1.11.

Otra posible solución para el bit de Suma sería el empleo de una simple puerta XOR de tres entradas (puesto que, como se observa de la tabla de verdad, la salida es uno cuando es impar el número de entradas iguales a uno):

$$Suma = (a \oplus b \oplus AcE)$$

Nos quedamos, sin embargo, con la primera implementación, puesto que no siempre es fácil la implementación física de una puerta XOR.

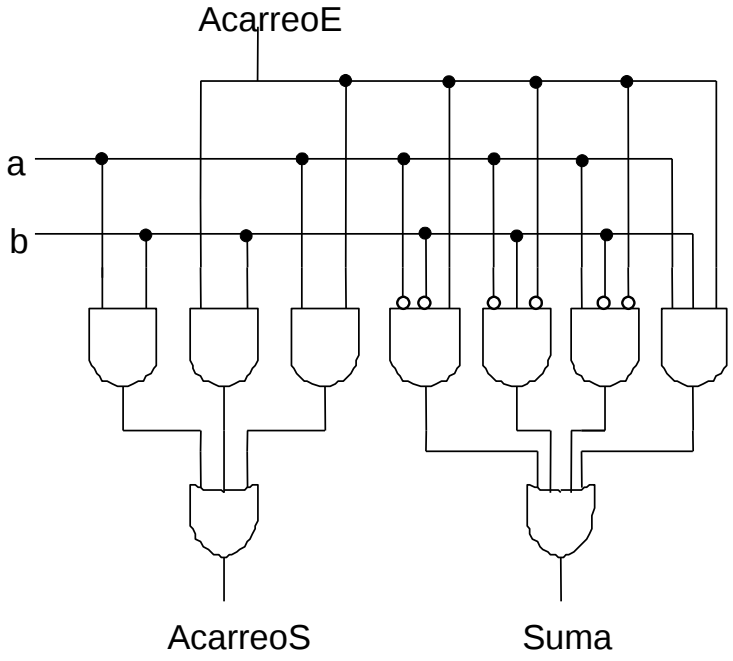


Figura 1.11: Sumador completo de 1 bit.

Una vez analizados cada uno de los componentes individuales de una ALU de 1 bit, sólo queda juntarlos en un sólo circuito, incorporando la lógica de control necesaria para seleccionar la operación lógica (OR o AND) o aritmética (SUMA) que se quiera utilizar. En este caso, la señal de entrada *Operación* necesita 2 bits para poder seleccionar una de las tres posibles operaciones, AND, OR o SUMA. En la figura 1.12 se muestra el resultado.

1.5.4 Implementación de una ALU de 32 bit

Construir una ALU sencilla (si bien un poco ineficiente) de 32 bits a partir de ALUs de 1 bit es tan sencillo como unir las cajas negras formadas por ALUs de 1 bit en cascada, de modo que el acarreo de salida del sumador de cada una corresponda al acarreo de entrada de la siguiente. El sumador resultante se llama sumador con propagación del

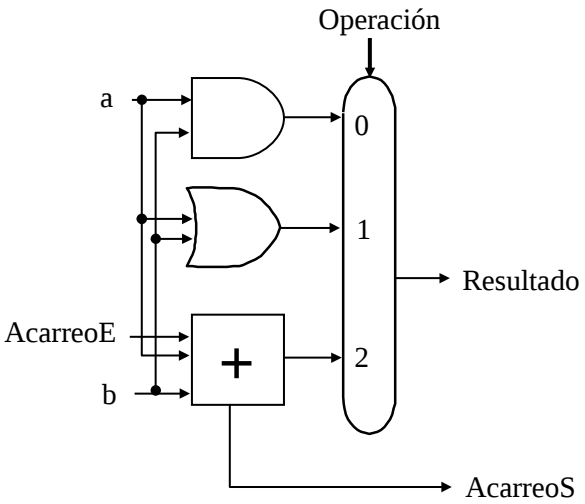


Figura 1.12: Unidad lógica de 1 bit con las operaciones AND, OR y suma.

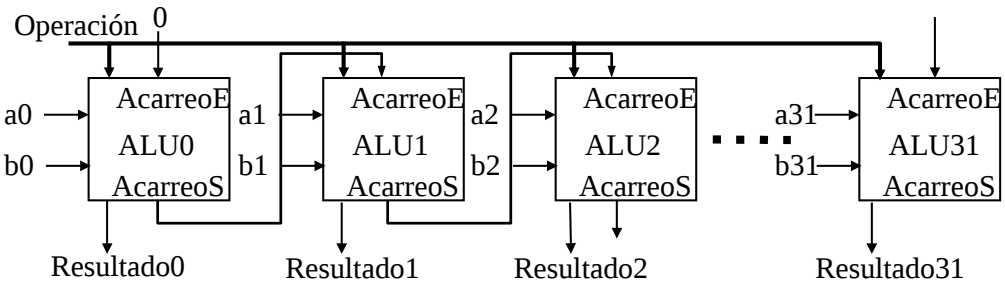


Figura 1.13: Unidad aritmético lógica de 32 bits con acarreo serie.

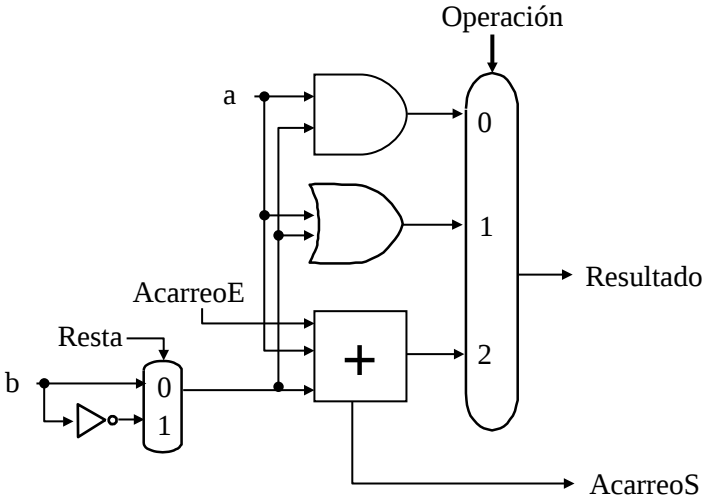


Figura 1.14: Unidad aritmético lógica de 1 bit, con posibilidad de restar (todos los bits menos el de mayor peso y el de menor peso).

acarreo (*ripple carry adder*). La ineficiencia de este sumador radica en el hecho que para calcular el bit i-ésimo de la suma es necesario haber calculado los $i-1$ bits anteriores, es decir, el bit 1 se calcula después del 0, el 2 después del 1 y así sucesivamente hasta el bit 31. El retardo del circuito completo queda, pues, como el de cada sumador individual multiplicado por el total de bits a calcular. Existen modos de solucionar este problema, mediante los llamados circuitos con acarreo anticipado (*look ahead carry*). En estos circuitos, el acarreo de salida de cada sumador no tiene que esperar a atravesar tantos niveles de puertas como ocurre en el sumador con propagación. De todos modos, el estudio detallado de estos últimos queda fuera del alcance de este curso.

Una ALU de 32 bits formada a partir del encadenamiento de ALUs de 1 bit quedaría como se indica en la figura 1.13.

La resta se implementa utilizando la aritmética en complemento a 2 y, por tanto, basta con sumar el complemento a 2 del operando B para conseguir el resultado deseado. Como ya hemos explicado, para obtener el complemento a 2 basta con negar el número bit a bit y sumarle uno. Así, para invertir el segundo operando bit a bit se añade un multiplexor que seleccione entre b_i o $\overline{b_i}$ en función de si la operación solicitada es la de sumar o la de restar. Para terminar de obtener el complemento a 2, aún falta sumarle 1. Para ello, si utilizamos la ALU de 32 bits antes explicada, basta con poner a 1 el acarreo de entrada de la ALU0, es decir la de menos peso, que para el resto de operaciones siempre estará a 0. La ALU de un bit quedaría como se indica en la figura 1.14 (en la figura 1.15 se muestra la ALU0 y en la figura 1.16 la ALU31, las cuales son ligeramente diferentes).

Nótese que, para la ALU de menor peso, si la entrada *Resta* vale 0, es decir, la ope-

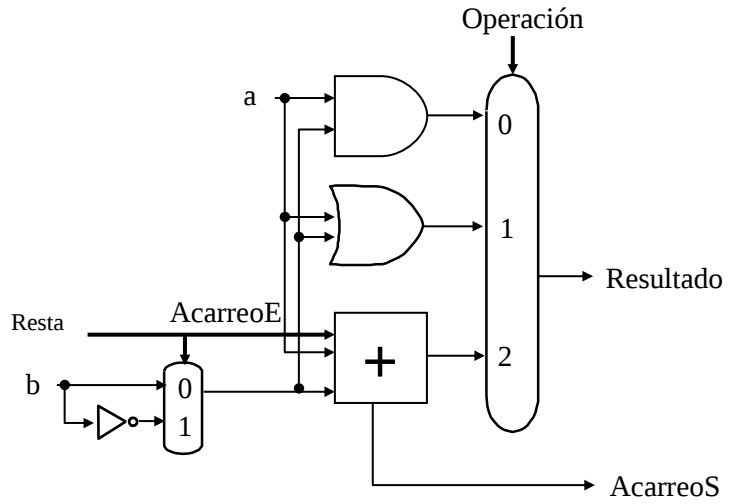


Figura 1.15: Unidad aritmético lógica de 1 bit, con posibilidad de restar (ALU0, bit de menos peso).

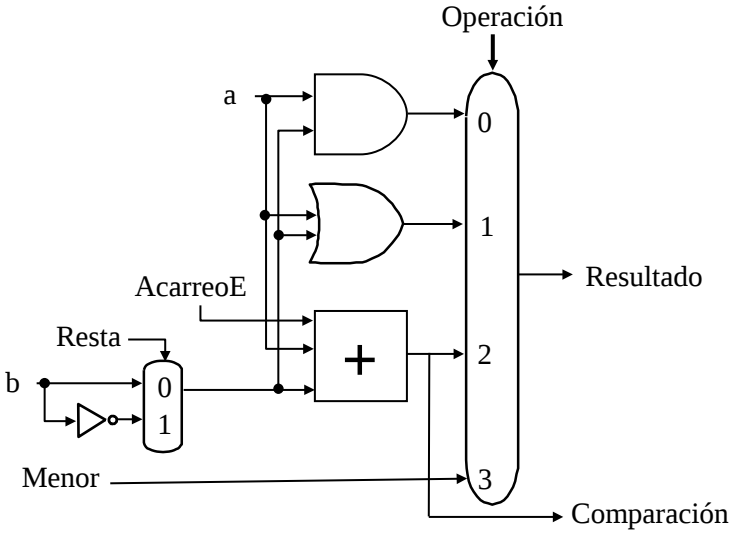


Figura 1.16: Unidad aritmético lógica de 1 bit, con posibilidad de restar y comparar (ALU31, bit de más peso).

ración no es una resta, el bit b no será invertido y el acarreo de entrada valdrá 0. Si el campo *Resta* vale 1, el bit b será invertido y el acarreo de entrada valdrá igual que *Resta*, es decir, 1.

Además de la suma y la resta, vamos a incluir también una operación para comparar la magnitud de los dos operandos, con el fin de saber cuál es mayor o menor. A esta operación la llamaremos *slt* (*set on less than*, activar si menor que), y devolverá un 1 si el primer operando fuente es menor que el segundo, y 0 en caso contrario (en realidad, puesto que la salida de la ALU es de 32 bits, devolverá 00..01 o 00..00, es decir, las constantes 0 y 1 en 32 bits). Para implementar la nueva operación hay que realizar unas pequeñas modificaciones en la ALU. Si se quiere saber si A $<$ B basta con saber si la diferencia es negativa, es decir, A $<$ B si A $-$ B es menor que cero. Para ello, podemos realizar la resta de A $-$ B y si el bit de signo, es decir, el bit más significativo de la ALU de 32 bits vale 1, significa que A es menor que B. La ALU31 quedaría como se indica en la figura 1.16.

La operación de comparación *slt* ha de devolver 00..01 en el bit menos significativo

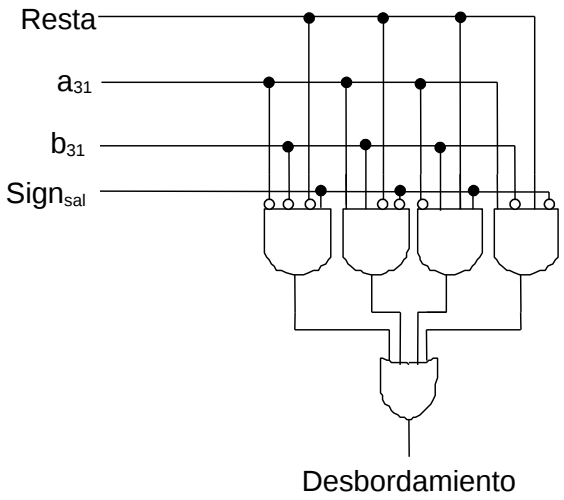


Figura 1.17: Unidad de detección de desbordamiento.

de la salida de 32 bits (Resultado0) si A=(a31..a0) es menor que B=(b31..b0), y el resto de bits (Resultado31..Resultado1) han de quedar a 0. Pero como se ha explicado antes, para saber si A es menor que B se ha de comprobar el bit de signo del resultado de la resta A-B. Por lo tanto, la ALU de 32 bits que realiza la comparación ha de conectar la salida del sumador de la ALU31 a la entrada *Menor* del multiplexor de la ALU0, dejando a 0 las entradas *Menor* del resto de las ALUs.

Finalmente, es interesante que nuestra ALU sea capaz de decir si sus dos operandos de entrada A y B son exactamente iguales. Para ello, basta con hacer la resta de los dos operandos y añadir una unidad de detección de 0 en el resultado: dos valores de 32 bits son iguales si su resta es igual a 00..00, y son diferentes en caso contrario. Una sencilla operación NOR sobre todos los bits del resultado será suficiente para esta tarea.

Antes de mostrar el esquema completo de la ALU de 32 bits, hay que tener en cuenta un posible problema que se puede presentar a la hora de realizar operaciones aritméticas. En concreto, hay que tener en cuenta que el conjunto de números enteros es infinito, mientras que no lo es la cantidad de bits destinada a representar dichos números dentro de un computador. Así, a la hora de construir la ALU, hay que tener en cuenta la posibilidad de que el resultado de una operación no se pueda representar con los bits de que se dispone para almacenar el resultado. Es decir, que pueda producirse un desbordamiento (*overflow*).

Por ejemplo, con 32 bits para representar números binarios, se sabe que se pueden representar 2^{32} números diferentes. Suponiendo que operamos con aritmética entera positiva, estos números irían del 0 al $2^{31} - 1$. Si sumamos dos números y su resultado es mayor que $2^{31} - 1$, entonces se habrá producido un desbordamiento, puesto que el resultado no cabe en 32 bits. Más en general, en una operación aritmética sobre valores enteros se produce desbordamiento cuando el resultado no se puede representar con los bits destinados, en nuestro ejemplo 32 bits. En el caso de la suma, esto sucede cuando se suman dos números positivos y el resultado es negativo (el bit más significativo vale 1). En el caso de la resta sucede cuando se resta un número negativo de uno positivo y el resultado es negativo o cuando se resta uno positivo de uno negativo y el resultado es positivo.

La siguiente tabla muestra las condiciones que han de cumplirse para que se produzca desbordamiento:

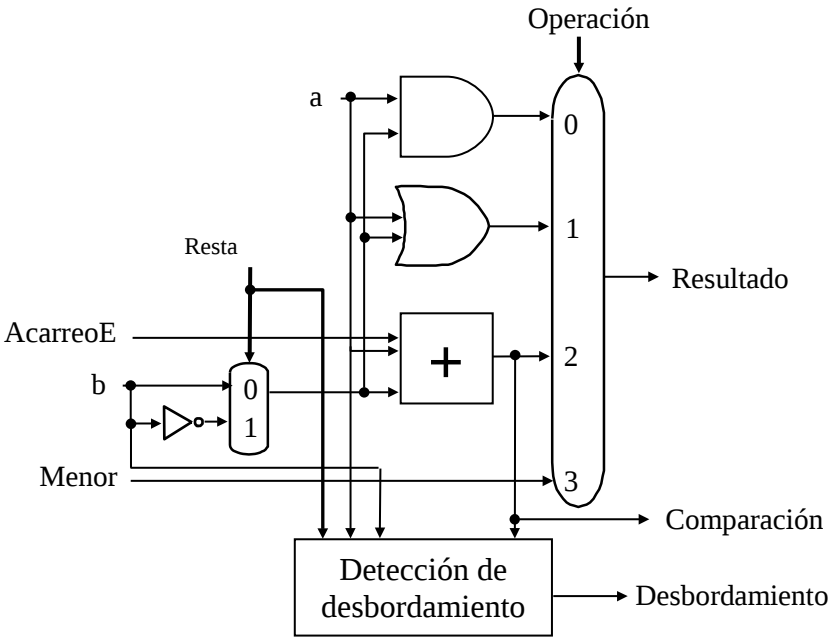
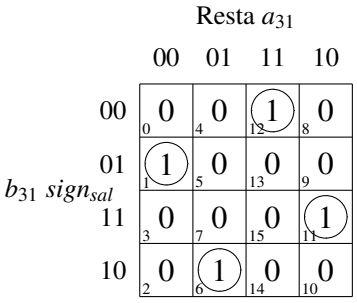


Figura 1.18: ALU31 con detección de desbordamiento.

Operación	**Operando A**	**Operando B**	**Resultado**
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A − B	≥ 0	< 0	< 0
A − B	< 0	≥ 0	≥ 0

Tomando en cuenta la tabla anterior, es fácil extender la ALU31 (la correspondiente al bit de mayor peso) para que detecte si hubo desbordamiento, simplemente mirando el tipo de operación que se desea hacer (suma o resta) y los signos de los operandos de entrada y el de la salida. Si llamamos a éste último $sign_{sal}$, y pasamos las cuatro posibles condiciones de desbordamiento a un mapa de Karnaugh, obtenemos:



La función obtenida será:

$$Desbordamiento = Resta' \cdot a'_{31} \cdot b'_{31} \cdot sign_{sal} + Resta' \cdot a_{31} \cdot b_{31} \cdot sign'_{sal} + Resta \cdot a'_{31} \cdot b_{31} \cdot sign_{sal} + Resta \cdot a_{31} \cdot b'_{31} \cdot sign'_{sal}$$

La implementación con puertas correspondiente aparece en la figura 1.17. La ALU31, con circuito detector de desbordamiento, queda como se indica en la figura 1.18. Y la

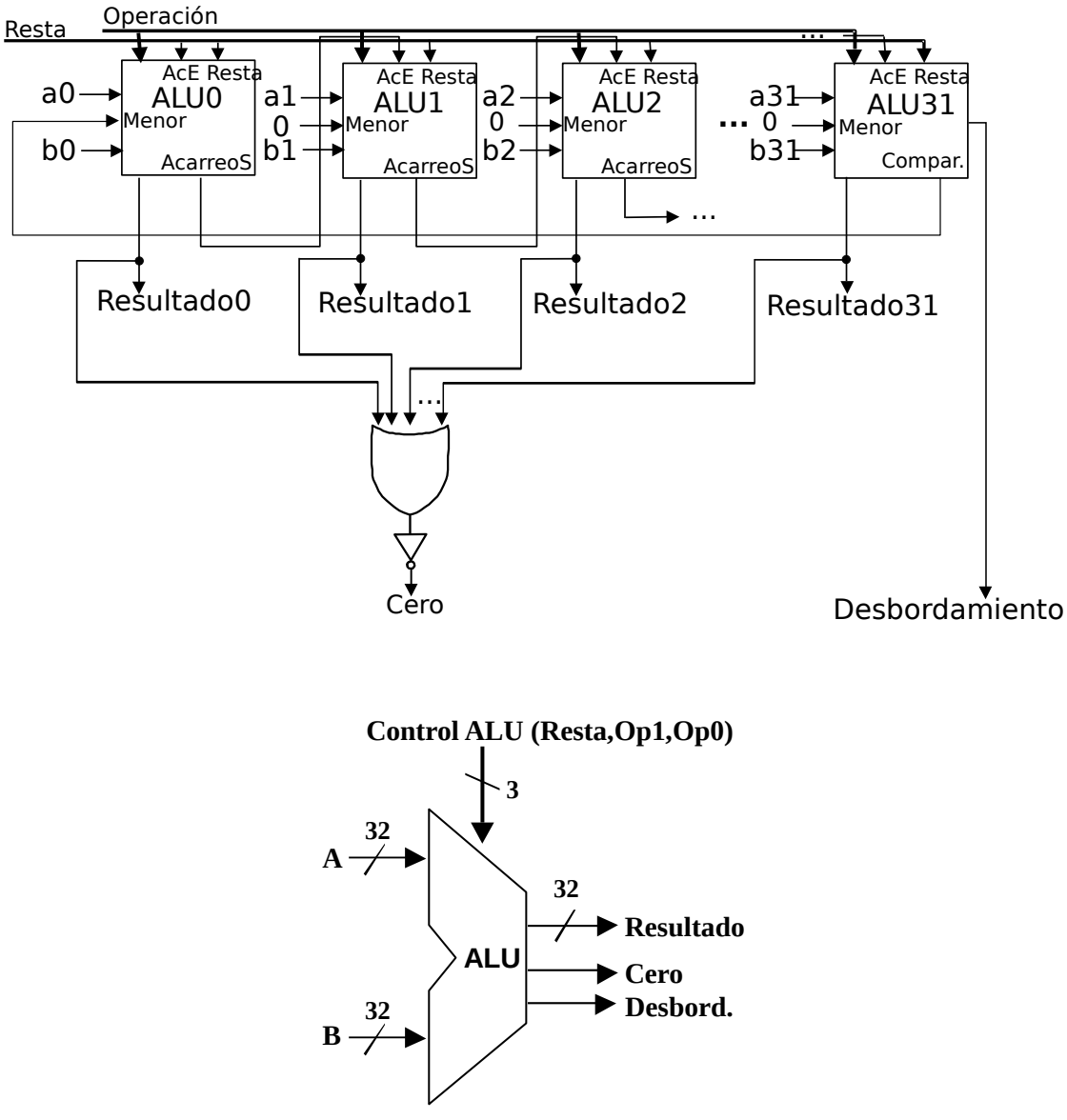


Figura 1.19: Esquema final de la ALU de 32 bits y bloque lógico correspondiente.

ALU completa de 32 bits que realiza sumas, restas, operaciones lógicas AND y OR (bit a bit), y comparaciones de magnitud e igualdad quedaría, finalmente, como aparece en la figura 1.19.

La relación entre las entradas de control y la operación que se llevaría a cabo en la ALU está especificado en la siguiente tabla:

Resta	Op_1	Op_0	Operación
0	0	0	AND
0	0	1	OR
0	1	0	SUMA
1	1	0	RESTA
1	1	1	SLT

1.6 EJERCICIOS: COMPONENTES DE UN PROCESADOR

1. Calcula el retardo de un sumador de 32 bits construido a partir de la concatenación en serie de 32 sumadores completos de 1 bit que siguen el esquema visto en la figura 1.10. Supón un retardo despreciable para las conexiones, de 5 ns para las puertas NOT, y de 10 ns para las AND y OR, independientemente del número de entradas que tengan.

2. La unidad de detección de desbordamiento de la ALU que se ha explicado en el apartado 1.5 puede simplificarse si, en lugar de trabajar con los bits b_{31} y Resta, se considerase directamente la salida del multiplexor controlado por este último bit (y cuyas entradas de datos eran b_{31} y b'_{31}). Si llamamos a esta salida b_{mux}, construye un circuito detector de desbordamiento más simple que el visto, cuyas entradas sean b_{mux}, a_{31} y $sign_{sal}$.

3. (*) Diseña un circuito combinacional que multiplique por 10 un número positivo de 32 bits. Para ello disponemos de desplazadores a la izquierda de 1 y 3 bits, y de un sumador de 32 bits. Nota: No es necesario tener en cuenta el desbordamiento. Recuerda: $10x = 2x + 8x$.

4. (*) Utilizando únicamente cuatro multiplexores del tamaño adecuado, implementa un módulo combinacional capaz de dividir o multiplicar por 2 un número entero sin signo de 4 bits. El circuito contará con una entrada de 4 bits de datos para recibir el operando de entrada y otra entrada de 1 bit de control, además de una salida de 4 bits. Cuando el bit de control valga 0, la salida debe ser igual al número de entrada multiplicado por 2 (desplazado un bit hacia la izquierda), y cuando el bit de control valga 1, debe ser igual al número de entrada dividido entre 2 (desplazado un bit a la derecha).

5. Diseña un desplazador de bits a la izquierda que permita realizar un desplazamiento de hasta 3 posiciones para la ristra de 4 bits que recibirá como entrada. Es decir, el desplazador dispondrá de 2 entradas, una con la palabra de 4 bits sobre la cual se quiere realizar la operación de desplazamiento, y otra de dos bits con la cantidad de desplazamiento (el número de bits a desplazar, de 0 a 3). Para la realización del circuito se dispone de multiplexores 4 a 1.

6. Amplía el circuito del ejercicio anterior para que el usuario pueda elegir también el sentido del desplazamiento (izquierda o derecha). Para ello se añade una tercera entrada (sentido) a través de la cual se especifica si el desplazamiento es hacia la izquierda (sentido = 0) o hacia la derecha (sentido = 1). Para la realización del circuito hay que utilizar, de nuevo, multiplexores 4 a 1.

7. (*) Se quiere diseñar un nuevo tipo de multiplexor 2 a 1, que denominaremos *multiplexor por rango*, para operandos de 8 bits. Como los multiplexores vistos en el apartado 1.2, el *multiplexor por rango* tiene dos tipos de entradas: entradas de datos (2) con los operandos de 8 bits (A y B), y 1 entrada de control (C) que en este caso será de 3 bits. El nuevo tipo de multiplexor dispone de una única salida de, evidentemente, 8 bits (S). El comportamiento del *multiplexor por rango* es sencillo: si el valor de la entrada C es estrictamente menor que 3, entonces la salida S toma el valor de la entrada A. En otro caso, la salida S toma el valor de la entrada B. Muestra cómo puede llevarse a cabo el diseño de dicho multiplexor a partir de un multiplexor 2 a 1 normal para operandos de 8 bits, un decodificador 3 a 8 y una puerta lógica básica.

8. Implementa un circuito combinacional que reciba como entrada un número entero con signo de 32 bits codificado en complemento a 2 y devuelva su valor absoluto a través de su salida de 32 bits. Para ello se dispone de la ALU de 32 bits explicada en el apartado 1.5 y cuyo diagrama lógico se muestra en la figura 1.19, y las puertas lógicas básicas que pudiesen ser necesarias.

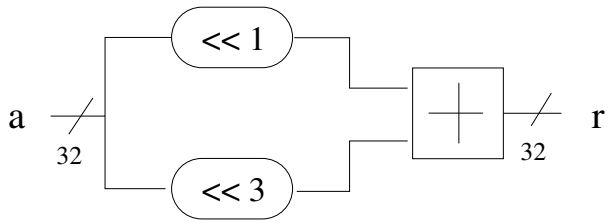
9. (*) Construye un circuito con 3 entradas de datos de 32 bits (que denominaremos `A`, `B` y `C`) y una entrada de control de 3 bits (que llamaremos `O`), y que tenga una salida de 32 bits (que llamaremos `S`). La tabla siguiente resume el comportamiento del circuito, mostrando el valor que se obtendría a la salida:

O	S	O	S
000	A+B	100	A−B
001	A+C	101	A−C
010	B+C	110	B−C
011	−1	111	0

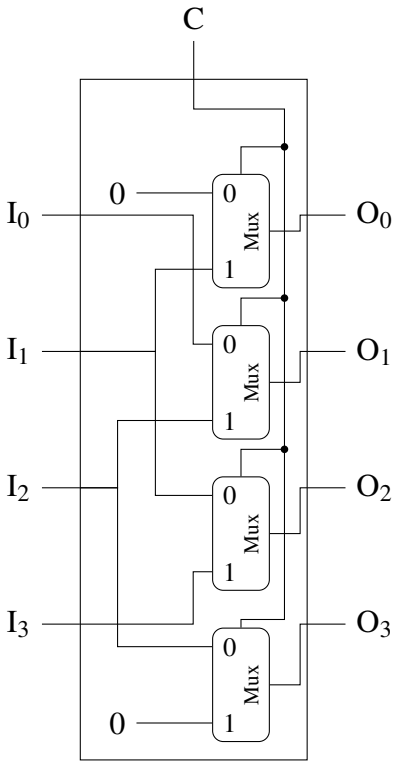
Para su realización puedes usar una ALU como la explicada en el apartado 1.5, un decodificador 3×8, y tantos multiplexores 2 a 1 de 32 bits y puertas lógicas básicas como necesites.

1.7 SOLUCIÓN A EJERCICIOS SELECCIONADOS

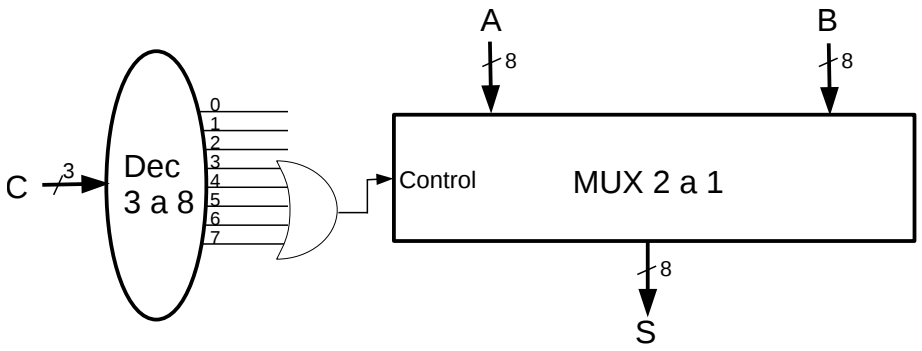
3. Utilizando los dos desplazadores proporcionados generaremos los valores $2x$ y $8x$. Después, utilizando el sumador de 32 bits, sumaremos los valores generados por los desplazamientos a la izquierda para obtener el resultado de la multiplicación por 10. El circuito quedaría de la siguiente forma:



4. Cuando el bit de control indique que hay que realizar una multiplicación, la salida O correspondiente será el valor de la entrada I desplazada un bit a la izquierda. Esto es, I_0 debe conectarse con O_1, I_1 con O_2 e I_2 con O_3. O_0 deberá mostrar el valor 0. En la caso de la división, el desplazamiento será a la derecha, poniendo el O_3 el valor 0.



7. Un decodificador 3 a 8 es un circuito que recibe a través de su entrada un número binario de 3 bits y activa la salida de 1 bit asociada a dicho número. De esta forma, a través del decodificador vamos a ser capaces de detectar los dos rangos que nos interesan. Para ello conectaremos la entrada C del multiplexor por rango con la entrada del decodificador, y las entradas A y B con las entradas de datos del multiplexor 2 a 1 normal. La salida S del multiplexor por rango será la salida del multiplexor normal. Para implementar el funcionamiento descrito en el enunciado, debemos asegurar que la entrada de control del multiplexor normal valdrá 1 cuando alguna de las salidas 3, 4, 5, 6 o 7 del decodificador vale 1. En otro caso, el valor de dicha entrada de control será 0. Esto podemos conseguirlo usando una puerta OR de 4 entradas, a la que conectaremos las salidas 3, 4, 5, 6 y 7 del decodificador. La salida de dicha puerta OR la conectaremos con la entrada de control del multiplexor normal. En definitiva, el circuito pedido quedaría como sigue:



Otra solución, que requeriría una puerta OR con sólo dos entradas, consistiría en conectar el bit más significativo de la entrada del decodificador (C_2) directamente con la puerta OR, en vez de los 4 bits (del 4 al 7) de salida del decodificador.

9. A la hora de diseñar el circuito pedido, usaremos 2 multiplexores 2 a 1, uno en cada entrada de la ALU, para poder seleccionar entre `A`, `B` y `C`. Vemos que el primer operando en todos los casos es `A` o `B`, con lo que el primer multiplexor las tendrá conectadas (a través sus entradas 0 y 1 respectivamente) y su salida se conectará a la primera entrada de la ALU. Para el segundo operando, que se conectará a la segunda entrada de la ALU, este toma los valores de `B` o `C`, con lo que el segundo multiplexor tendrá conectadas ambas entradas (a través sus entradas 1 y 0 respectivamente). Observamos que las operaciones a realizar son la suma (valores de `O` 0, 1 y 2) y la resta (valores de `O` 4, 5 y 6), con lo que los dos bits menos significativos de la señal de control de la ALU serán siempre $10_{)2}$ y el bit más significativo coincidirá con el bit 2 de la entrada `O`.

   Vemos también que la entrada `B` está conectada en ambos multiplexores, por lo que podemos obtener el valor 0 configurando ambos multiplexores para que dejen pasar el valor de su entrada 1 (el valor de `B`) y haciendo una resta con la ALU, que en este caso sería inmediato pues el bit 2 de la entrada `O` ya vale 1. Para generar el valor -1 ($1111...11_{)2}$), añadimos un multiplexor, que será el que determine el valor de la salida `S` y que nos permitirá elegir entre el resultado de la ALU y la constante $1111...11_{)2}$.

   Por último, usaremos el decodificador 3×8 para facilitar la configuración de los multiplexores en cada caso. La señal de control del multiplexor que controla el valor de `S` estará conectado con la salida 3 del decodificador. La señal de control del multiplexor que controla el valor de la primera entrada de la ALU la generará una puerta OR que tendrá conectadas las salidas 2, 6 y 7 del decodificador. Mientras que la señal de control del multiplexor que controla el valor de la segunda entrada de la ALU la generará otra puerta OR que tendrá conectadas las salidas 0, 4 y 7 del decodificador.

   El circuito resultante queda como sigue: