Universidad Rey Juan Carlos

Arquitectura e Ingeniería de Computadores

Grado de Ingeniería del Software

TEMA 2. UNIDAD ARITMÉTICO-LÓGICA (ALU)

Ángel Serrano Sánchez de León
Óscar David Robles Sánchez
Luis Rincón Córcoles

Universidad Rey Juan Carlos

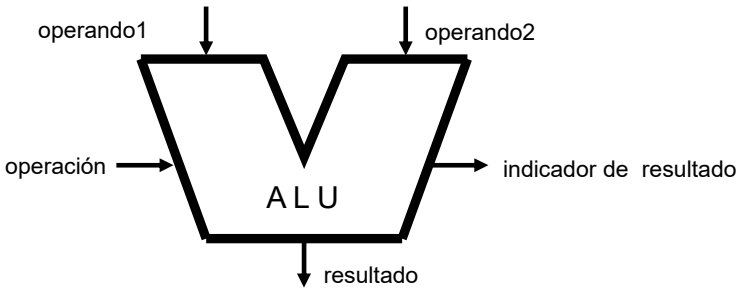
ÍNDICE

- Introducción
- Conceptos previos
- Circuitos aritméticos-lógicos básicos
- Construcción incremental de una ALU sencilla
- Otros circuitos aritméticos-lógicos

1. INTRODUCCIÓN

Una **unidad aritmético-lógica** es un circuito combinacional que realiza las operaciones aritméticas y lógicas básicas en el computador.

- Operaciones aritméticas básicas: `add`, `sub`, `mult`, `div`
- Operaciones lógicas básicas: `not`, `and`, `or`, `nand`, `nor`
- Operaciones de desplazamiento de bits: `srl`, `sll`
- En inglés: ALU (*arithmetic-logic unit*)



3

- La unidad aritmético-lógica:
  - Es el circuito encargado de realizar todas las operaciones matemáticas.
    - Aritméticas.
    - Lógicas.
    - De desplazamiento.
  - Es de tipo combinacional (no tiene memoria).
    - El valor del resultado se calcula en función del valor de las entradas en dicho instante.
  - Recibe 2 operandos, ya que la mayoría de las operaciones que realiza son binarias (de 2 operandos), si bien en ocasiones solo acepta 1 operando (por ejemplo, si la operación que vamos a realizar es la función NOT o la de cambio de signo NEG).
  - Además de estas entradas, se reciben unas señales de control que identifican la operación que se va a realizar.
  - La ALU proporciona como salida, por un lado, el resultado de la operación. Por otro lado, genera una serie de indicadores del resultado obtenido, como si salió cero, negativo, desbordamiento, etc.
  - Tanto los operandos como el resultado están expresados en binario de tamaño *n* bits. Este tamaño es el de PALABRA del computador.
  - El símbolo estándar de una ALU (a modo de caja negra) es una especie de V, donde los operandos entran por cada uno de los extremos y el resultado sale por el lado picudo.

Universidad Rey Juan Carlos

CONTENIDOS

- Introducción
- Conceptos previos
- Circuitos aritméticos-lógicos básicos
- Construcción incremental de una ALU sencilla
- Otros circuitos aritméticos-lógicos

VÍDEO

https://youtu.be/G_RPm_bkToI

5

- En este vídeo se repasan los principales sistemas de representación numérica en los computadores:
  - Binario puro.
  - Hexadecimal.
  - Complemento a 2.
- También se recuerdan las potencias de 2.

[Repaso de Introducción a la Informática]

VÍDEO

https://youtu.be/WXeQMZdSpHg

6

- En este vídeo se repasan las principales reglas aritméticas para datos en binario puro:
  - Sumar.
  - Restar.
  - Multiplicar (suma-desplazamiento).
  - Dividir (división por restauración).

[Repaso de Introducción a la Informática]

VÍDEO

https://youtu.be/7yvVqe05dEc

7

- En este vídeo se repasan las principales reglas aritméticas para datos en complemento a 2:
  - Sumar.
  - Restar.
- Solo se da una pincelada de cómo se gestionaría el signo para:
  - Multiplicar.
  - Dividir.

[Repaso de Introducción a la Informática]

VÍDEO

https://youtu.be/jlW-I-MypcQ

8

- En este vídeo se repasan las puertas lógicas:
  - NOT.
  - AND.
  - OR.
  - XOR.
  - NAND.
  - NOR.
  - XNOR.

[Repaso de Fundamentos Físicos de la Informática]

Universidad Rey Juan Carlos

VÍDEO

https://youtu.be/jHF-fxPZvJk

9

- En este vídeo se repasan los multiplexores (MUX), un tipo de circuito combinacional que actúa como un selector.
  - Entradas:
    - Entradas de datos, un total de 2^n, cada una de 1 bit. Las entradas se numeran según su peso de 0 a 2^n-1.
    - Entradas de selección (o de control), *n*, cada una de 1 bit. Las entradas se numeran según su peso de 0 a *n*-1.
  - Salida, solo 1 bit (el seleccionado).
  - Funcionamiento:
    - Solo una de las 2^n entradas pasa a la salida y las otras se ignoran.
    - Las entradas de selección se interpretan como un número codificado en binario (*n* bits).
    - La entrada que corresponde a dicho número expresado en decimal es la que se conecta a la salida.
  - Ejemplo: si hay 2 bits de selección, habrá 4 entradas como máximo (numeradas 3, 2, 1 y 0). Si los bits de selección valen 10, entonces la salida será igual a la entrada 2.
  - A veces se incluye otra entrada más llamada *enable*, que actúa como un interruptor.
    - Si *enable* vale 0, la salida vale 0 independientemente de las entradas de datos y de selección.
    - Por el contrario, si *enable* vale 1, entonces el funcionamiento del multiplexor es como hemos indicado antes.

[Repaso de Fundamentos Físicos de la Informática]

```
(bit result) mux2a1(bit entradas[2], bit control, bit enable)
{
    bit result = 0
    if (enable == 1)
    {
        switch (control)
        {
            case 0:
                result = entradas[0]
                break
            default:
                result = entradas[1]
        }
    }
    return (result)
}

(bit result) mux4a1(bit entradas[4], bit control[2], bit enable)
{
    bit result = 0
    if (enable == 1)
    {
        switch (control)
        {
            case 00:
                result = entradas[0]
                break
            case 01:
                result = entradas[1]
                break
            case 10:
                result = entradas[2]
                break
            default:
                result = entradas[3]
        }
    }
    return (result)
}

(bit result) mux8a1(bit entradas[8], bit control[3], bit enable)
{
    bit result = 0
    if (enable == 1)
    {
        switch (control)
        {
            case 000:
                result = entradas[0]
                break
            case 001:
                result = entradas[1]
                break
            case 010:
                result = entradas[2]
                break
            case 011:
                result = entradas[3]
                break
            case 100:
                result = entradas[4]
                break
            case 101:
                result = entradas[5]
                break
            case 110:
                result = entradas[6]
                break
            default:
                result = entradas[7]
        }
    }
    return (result)
}
```

VÍDEO

https://youtu.be/6uCwgxxjxvE

10

- En este vídeo se repasan los decodificadores, circuitos combinacionales que reciben un valor en binario puro y activan la salida correspondiente al mismo valor en decimal.

[Repaso de Fundamentos Físicos de la Informática]

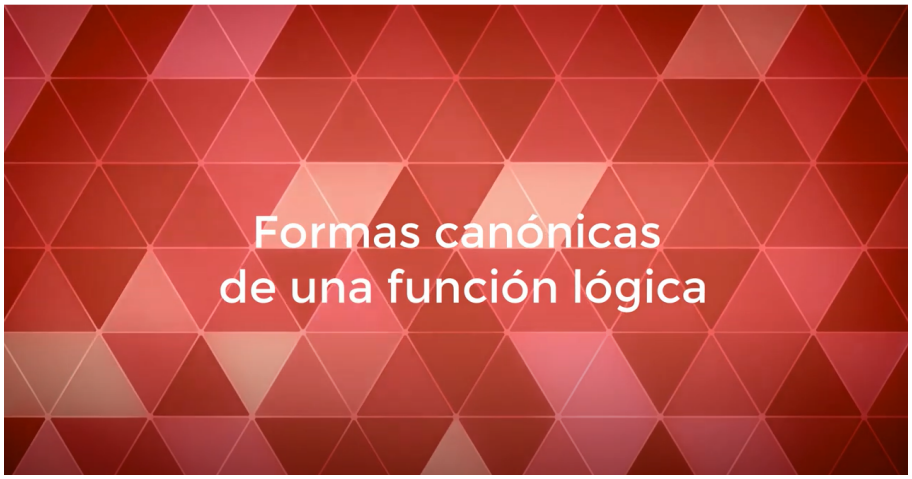
```
(bit salida[2]) decod1a2(bit entrada, bit enable)
{
    bit salida[2] = 00
    if (enable == 1)
    {
        switch (entrada)
        {
            case 0:
                salida[0] = 1
                break
            default:
                salida[1] = 1
        }
    }
    return (salida)
}

(bit salida[4]) decod2a4(bit entrada[2], bit enable)
{
    bit salida[4] = 0000
    if (enable == 1)
    {
        switch (entrada)
        {
            case 00:
                salida[0] = 1
```

```
                break
            case 01:
                salida[1] = 1
                break
            case 10:
                salida[2] = 1
                break
            default:
                salida[3] = 1
        }
    }
    return (salida)
}

(bit salida[8]) decod3a8(bit entrada[3], bit enable)
{
    bit salida[8] = 00000000
    if (enable == 1)
    {
        switch (entrada)
        {
            case 000:
                salida[0] = 1
                break
            case 001:
                salida[1] = 1
                break
            case 010:
                salida[2] = 1
                break
            case 011:
                salida[3] = 1
                break
            case 100:
                salida[4] = 1
                break
            case 101:
                salida[5] = 1
                break
            case 110:
                salida[6] = 1
                break
            default:
                salida[7] = 1
        }
    }
    return (salida)
}
```

VÍDEO



https://youtu.be/Ey06G0pq3g4

11

- En este vídeo se repasan las funciones lógicas o de conmutación, con los conceptos de tabla de verdad y las formas canónicas, la basada en la suma de minitérminos y la del producto de maxitérminos.

[Repaso de Fundamentos Físicos de la Informática]

VÍDEO

https://youtu.be/MpldrPSA30g

12

- En este vídeo se repasa el método de los mapas de Karnaugh, que permite simplificar de manera sencilla una función lógica de hasta 4 variables.

[Repaso de Fundamentos Físicos de la Informática]

CONTENIDOS

- Introducción
- Conceptos previos
- Circuitos aritméticos-lógicos básicos
- Construcción incremental de una ALU sencilla
- Otros circuitos aritméticos-lógicos

Universidad Rey Juan Carlos

CIRCUITOS ARITMÉTICOS BÁSICOS

- Semisumador de 1 bit
- Sumador completo de 1 bit
- Sumador completo de n bits
- Restador
- Sumador-restador

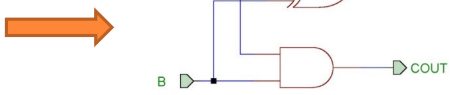
14

- Vamos a estudiar algunos circuitos aritméticos para datos en binario.
- Iremos de menor a mayor complejidad.

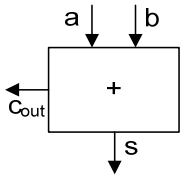
SEMISUMADOR DE 1 BIT

- El semisumador (*half adder*) es un circuito que suma dos bits de entrada a y b, y devuelve un bit de resultado s y un bit de acarreo c_{out}.

a	b	c_{out}	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



- Se suele representar con este símbolo:



$$c_{\text{out}} = a \cdot b$$

$$s = a \oplus b$$

15

```
(bit c_out, bit s) semisumador_1_bit (bit a, bit b)
{
    bit c_out = a AND b
    bit s = a XOR b
    return (c_out, s)
}
```

- Como suma dos números de 1 bit, el resultado de la suma puede ser un número de 0 a 2 (en binario, 2 bits):
  - c_{out} es el bit más significativo del resultado.
  - s es el bit menos significativo del resultado.
- La implementación del semisumador de 1 bit utiliza una puerta AND y una puerta XOR, ambas de 2 entradas.
- En la transparencia se muestra la tabla de verdad del circuito, que es la representación en formato tabular de los valores de las salidas en función de todas las combinaciones de las entradas. Como hay 2 entradas, hay un total de 2^2=4 filas.
- En la representación de un circuito como una caja negra:
  - Se dibujan las entradas como flechas que apuntan hacia la caja.
  - Se dibujan las salidas como flechas que salen de la caja.
  - No se muestra la estructura interna del circuito.

SUMADOR COMPLETO DE 1 BIT

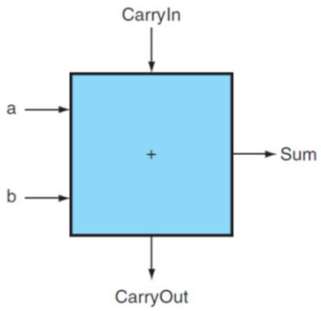
- El sumador completo (*full adder*) es un circuito que suma dos bits de entrada a y b más un acarreo de entrada c_{in} y devuelve un bit de resultado s y un bit de acarreo c_{out}.

Entradas			Salidas		Operación
a	b	c_{in}	c_{out}	s	
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+1=10
1	1	1	1	1	1+1+1=11

16

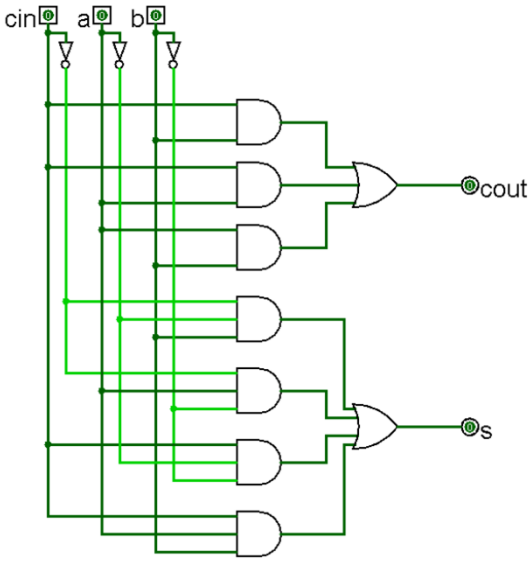
- Como suma tres números de 1 bit, el resultado de la suma puede ser un número de 0 a 3 (en binario, 2 bits):
  - c_{out} es el bit más significativo del resultado.
  - s es el bit menos significativo del resultado.
- En la transparencia se muestra la tabla de verdad del circuito. Como hay 3 entradas, hay un total de 2^3=8 filas.

SUMADOR COMPLETO DE 1 BIT



$$c_{\text{out}} = c_{\text{in}} \cdot b + c_{\text{in}} \cdot a + a \cdot b$$

$$s = \overline{c_{\text{in}}} \cdot \bar{a} \cdot b + \overline{c_{\text{in}}} \cdot a \cdot \bar{b} + c_{\text{in}} \cdot \bar{a} \cdot \bar{b} + c_{\text{in}} \cdot a \cdot b = (a \oplus b) \oplus c_{\text{in}}$$



17

```
(bit c_out, bit s) sumadorCompleto_1_bit(bit a, bit b, bit c_in)
{
    bit c_out = (c_in AND b) OR (c_in AND a) OR (a AND b)
    bit s = (a XOR b) XOR c_in
    return (c_out, s)
}
```

- En la parte inferior izquierda se muestran las expresiones lógicas para el acarreo de salida c_{out} y para la suma.
  - Recordemos que el apóstrofo (o una barra superior) indica negación NOT.
  - El punto (producto lógico) corresponde a una puerta AND.
  - La suma lógica corresponde a una puerta OR.
  - El signo + rodeado por un círculo corresponde a una puerta XOR.
- A la derecha se muestra una posible implementación del circuito con puertas lógicas.

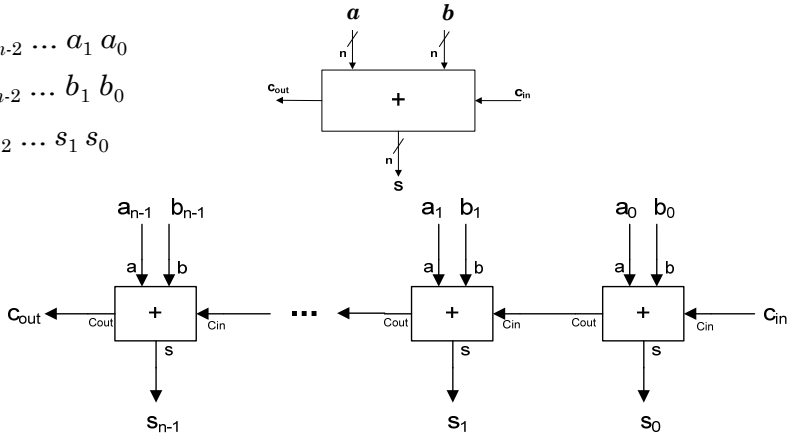
SUMADOR COMPLETO DE n BITS

- Sumador de n bits con propagación de acarreo:
  - Se construye conectando en cascada varios sumadores completos de 1 bit.

$a = a_{n-1}\ a_{n-2} \ldots a_1\ a_0$

$b = b_{n-1}\ b_{n-2} \ldots b_1\ b_0$

$s = s_{n-1}\ s_{n-2} \ldots s_1\ s_0$



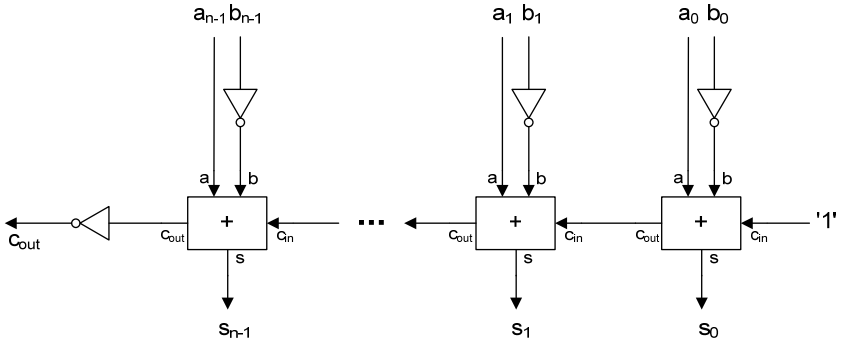

```
(bit c_out, bit s[n]) sumadorCompleto_n_bits(bit a[n], bit b[n], bit c_in) // n es
un valor entero mayor o igual que 2
{
    bit c[n], s[n]
    (c[0], s[0]) = sumadorCompleto_1_bit(a[0], b[0], c_in)
    for (int i = 1; i<n; i++)
    {
        (c[i], s[i]) = sumadorCompleto_1_bit(a[i], b[i], c[i-1])
    }
    return (c[n-1], s)
}
```

- Trabajaremos ahora con sumadores completos de 1 bit como caja negra.
- Si conectamos en serie *n* sumadores completos de 1 bit, el circuito resultante es un sumador completo de *n* bits.
- La conexión del acarreo de salida de un sumador con el acarreo de entrada del siguiente permite llevar la cuenta correctamente de "cuándo te llevas una", es decir, de la propagación del acarreo.
- **Este circuito solo permite sumar números en binario puro**.

RESTADOR

- Para restar dos números en binario se hace la suma del minuendo con el complemento a 2 del sustraendo. El complemento a 2 es el complemento a 1 más '1'. En la resta binaria hay que invertir el acarreo final.

$$a - b = a + C_2(b) = a + C_1(b) + 1 = a + \bar{b} + 1$$



19

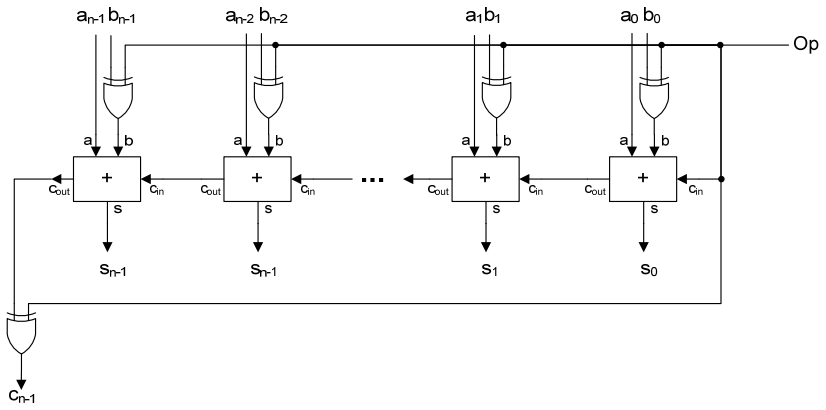
```
(bit c_out, bit s[n]) restadorCompleto_n_bits(bit a[n], bit b[n])
{
    bit c[n], s[n]
    (c[0], s[0]) = sumadorCompleto_1_bit(a[0], NOT b[0], 1)
    for (int i = 1; i<n; i++)
    {
        (c[i], s[i]) = sumadorCompleto_1_bit(a[i], NOT b[i], c[i-1])
    }
    c[n-1] = NOT c[n-1]
    return (c[n-1], s)
}
```

- Como puede verse, se ha implementado este restador mediante un sumador completo modificado.
  - Cada bit de *b* es invertido por una puerta NOT antes de entrar en el sumador correspondiente.
  - El acarreo de entrada del bit 0 siempre vale 1.
  - El acarreo de salida del bit *n*-1 es invertido con una puerta NOT para que tenga el valor correcto.

SUMADOR-RESTADOR

$$X \oplus 0 = X$$
$$X \oplus 1 = \overline{X}$$

- Se pueden combinar el sumador y el restador en un **único** circuito que realice la operación de suma o de resta en función de una señal de control.



20

```
(bit c_out, bit s[n]) sumadorRestador_n_bits(bit a[n], bit b[n], bit Op)
{
    bit c[n], s[n]
    (c[0], s[0]) = sumadorCompleto_1_bit(a[0], b[0] XOR Op, Op)
    for (int i = 1; i<n; i++)
    {
        (c[i], s[i]) = sumadorCompleto_1_bit(a[i], b[i] XOR Op, c[i-1])
    }
    c[n-1] = c[n-1] XOR Op
    return (c[n-1], s)
}
```

- Si integramos en un único circuito el sumador y el restador, obtenemos un sumador-restador.
  - Está basado también en un sumador completo modificado.
  - Cada bit de *b* pasa por una puerta XOR, donde se hace la suma exclusiva con un bit de operación (*Op*).
    - *Op* = 0 → Suma.
    - *Op* = 1 → Resta.
  - El acarreo de entrada del bit menos significativo es ahora *Op*, que no es un valor fijo, sino que es una entrada más del circuito.
    - Esta señal *Op* es generada desde fuera del circuito sumador-restador, ya sea manualmente o por otro circuito (como la unidad de control).
  - El acarreo de salida del bit más significativo también necesita una XOR para tener el resultado correcto.
  - Este circuito permite sumar y restar sin necesidad de tener dos circuitos, uno para cada operación.
  - Para permitir la resta, los operandos *a* y *b* están en complemento a 2.

CONTENIDOS

- Introducción
- Conceptos previos
- Circuitos aritméticos-lógicos básicos
- Construcción incremental de una ALU sencilla
- Otros circuitos aritméticos-lógicos

Universidad Rey Juan Carlos

CONSTRUCCIÓN DE UNA ALU SENCILLA

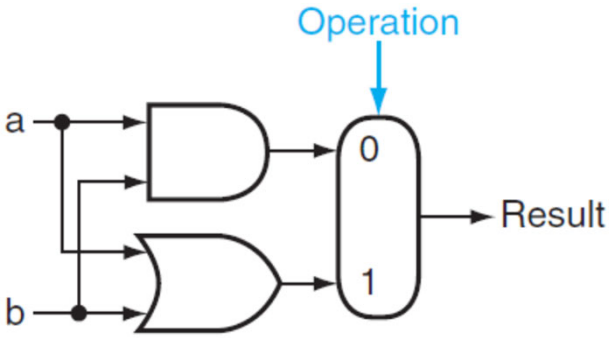
- Diseño modular de abajo arriba:
  - MiniALU para datos de 1 bit
  - ALU para datos de 32 bits por repetición y encadenamiento de 32 miniALU de 1 bit
- Operaciones aritmético-lógicas implementadas:
  - `and`
  - `or`
  - `add`
  - `sub`
  - `slt`
  - `nor`

22

- Basándonos en los circuitos anteriores y en otros que iremos explicando a continuación, vamos a construir desde cero una ALU sencilla que implementará un repertorio seleccionado del procesador MIPS:
  - Entradas:
    - Operando 1: *a* (32 bits, complemento a 2).
    - Operando 2: *b* (32 bits, complemento a 2).
    - Palabra de operación: señal que le indica la ALU qué operación debe realizar.
  - Salidas:
    - Resultado: *r* (32 bits, complemento a 2).
    - Señal de Cero (*Z*): valdrá 1 si $r = 0$ y 0 si $r \neq 0$.
    - Señal de Desbordamiento (*V*): valdrá 1 si se ha producido un desbordamiento (*overflow*) en una operación aritmética.
  - Operaciones implementadas: tomaremos una muestra representativa, pero reducida, de las operaciones típicas de una ALU.
    - Producto lógico: and.
    - Suma lógico: or.
    - Suma aritmética: add.
    - Resta aritmética: sub.
    - Activar si es menor: slt.
    - Suma lógica negada: nor.

OPERACIONES LÓGICAS: and/or

- La implementación de circuitos para operaciones lógicas es muy sencilla: basta simplemente con una batería de puertas lógicas y un multiplexor accionado por las correspondientes señales de selección.
- Ejemplo: unidad lógica de 1 bit con las funciones `and` y `or`.
- **Operación**: señal de selección de 1 bit.

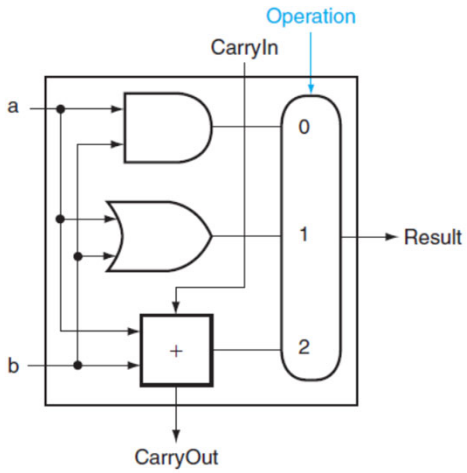



```
(bit result) ALU_1_bit(bit a, bit b, bit operation)
{
    bit result
    bit enable = 1  // No está dibujado porque se sobreentiende
    bit entradasMux2a1[2]
    entradasMux2a1[0] = a AND b
    entradasMux2a1[1] = a OR b
    result = mux2a1(entradasMux2a1, operation, enable)
    return (result)
}
```

- Comenzamos con el diseño de una "miniALU" o celda básica de la ALU completa:
  - Admite solo operandos de 1 bit.
  - Implementa todas las operaciones. Para ello:
    - Calcula todos los posibles resultados.
    - Selecciona el resultado deseado mediante un multiplexor controlado por una señal de *Operation*.
      - Por simplicidad, todos los multiplexores que aparecerán en este tema y el siguiente carecen de señal *Enable*. Supondremos que está siempre activada a 1.
- En esta primera parte tenemos:
  - Una puerta AND para el producto lógico.
  - Una puerta OR para la suma lógica.
  - Ambas puertas lógicas reciben una entrada de 1 bit llamada *a* y otra llamada *b*.
  - El resultado de la operación también tiene 1 bit y se elige mediante el multiplexor de 2 entradas. La señal *Operation* es de 1 bit.
- Operaciones realizadas:
  - *Operation* = 0 → Producto lógico (AND).
  - *Operation* = 1 → Suma lógica (OR).

OPERACIÓN SUMA: add

- Añadimos el sumador completo a nuestra ALU, haciendo más grande el multiplexor:



24

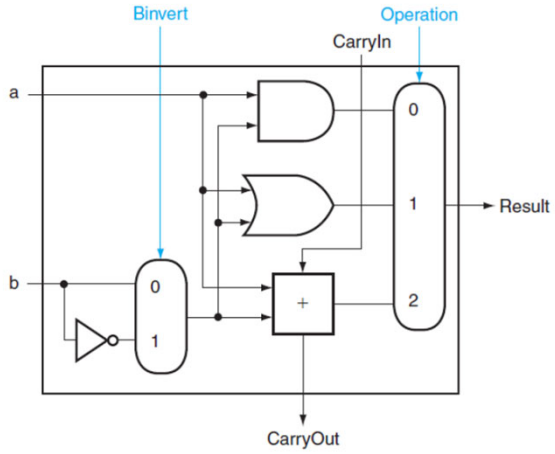
```
(bit c_out, bit result) ALU_1_bit(bit a, bit b, bit c_in, bit operation[2])
{
    bit c_out, result
    bit enable = 1  // No está dibujado porque se sobreentiende
    bit entradasMux4a1[4]
    entradasMux4a1[0] = a AND b
    entradasMux4a1[1] = a OR b
    (c_out, entradasMux4a1[2]) = sumadorCompleto_1_bit(a, b, c_in)
    entradasMux4a1[3] = undefined
    result = mux4a1(entradasMux4a1, operation, enable) // No existen los
multiplexores 3 a 1, debe tener 4 entradas de datos
    return (c_out, result)
}
```

- Añadimos también un sumador completo de 1 bit.
- Además de las entradas *a*, *b* y *Operation*, hay una nueva entrada que es el acarreo de entrada c_{in} = *CarryIn*.
- El multiplexor debe crecer, ya que esta pequeña ALU realiza 3 operaciones (producto lógico, suma lógica y suma aritmética).
    - Los bits de selección de *Operation* deben ser 2 (porque permiten hasta 4 operaciones).
- Operaciones realizadas:
    - *Operation* = 00 → Producto lógico (AND).
    - *Operation* = 01 → Suma lógica (OR).
    - *Operation* = 10 → Suma aritmética (*a* + *b*).

OPERACIÓN RESTA: sub

- Resta en complemento a 2: suma del negado más 1.

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



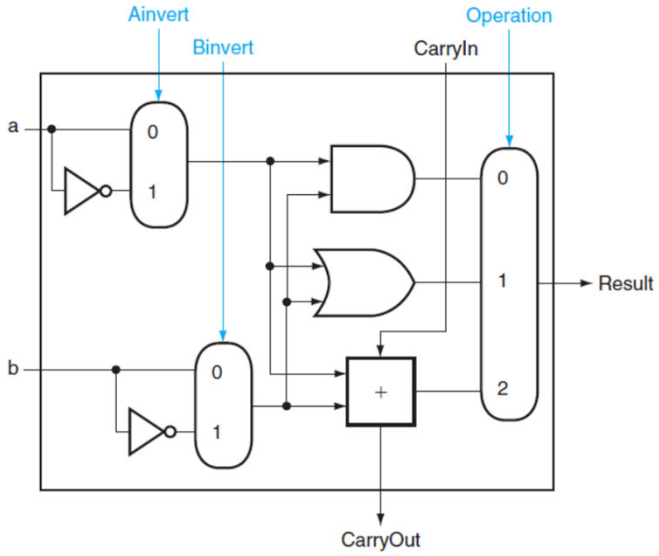
Resta: $c_{in} = B_{invert} = 1$


```
(bit c_out, bit result) ALU_1_bit(bit a, bit b, bit c_in, bit bInvert, bit operation[2])
{
    bit c_out, result
    bit enable = 1  // No está dibujado porque se sobreentiende
    bit operandoB
    bit entradasMux2a1[2]
    bit entradasMux4a1[4]
    entradasMux2a1[0] = b
    entradasMux2a1[1] = NOT b
    operandoB = mux2a1(entradasMux2a1, bInvert, enable)
    entradasMux4a1[0] = a AND operandoB
    entradasMux4a1[1] = a OR operandoB
    (c_out, entradasMux4a1[2]) = sumadorCompleto_1_bit(a, operandoB, c_in)
    entradasMux4a1[3] = undefined
    result = mux4a1(entradasMux4a1, operation, enable)
    return (c_out, result)
}
```

- Incorporamos ahora la posibilidad de restar.
- Restar es lo mismo que sumar el opuesto:
  - En el sistema de representación en complemento a 2, calcular el opuesto es hallar el valor complementario, que se calcula invirtiendo todos los bits (por eso hemos añadido una puerta NOT en el bit *b*) y al resultado sumarle 1. Este 1 adicional se introduce mediante la señal c_{in} (acarreo de entrada).
- Hemos añadido un nuevo multiplexor controlado mediante la señal B_{invert}.
  - Si vale 0, se selecciona *b* y se suma *a* + *b*. Si vale 1, se selecciona *b* negado y se resta *a* – *b*.
- Operaciones realizadas:
  - *Operation* = 00 y B_{invert} = 0 → Producto lógico (AND). Se ignora c_{in}.
  - *Operation* = 01 y B_{invert} = 0 → Suma lógica (OR). Se ignora c_{in}.
  - *Operation* = 10, B_{invert} = 0 y c_{in} = 0 → Suma aritmética (*a* + *b*).
  - *Operation* = 10, B_{invert} = 1 y c_{in} = 1 → Resta aritmética (*a* – *b*).

OPERACIONES LÓGICAS: nor

$$\overline{a+b} = \bar{a} \cdot \bar{b}$$



En el multiplexor, comparten señal de selección (*Operation*) las instrucciones:
- and y nor
- add y sub

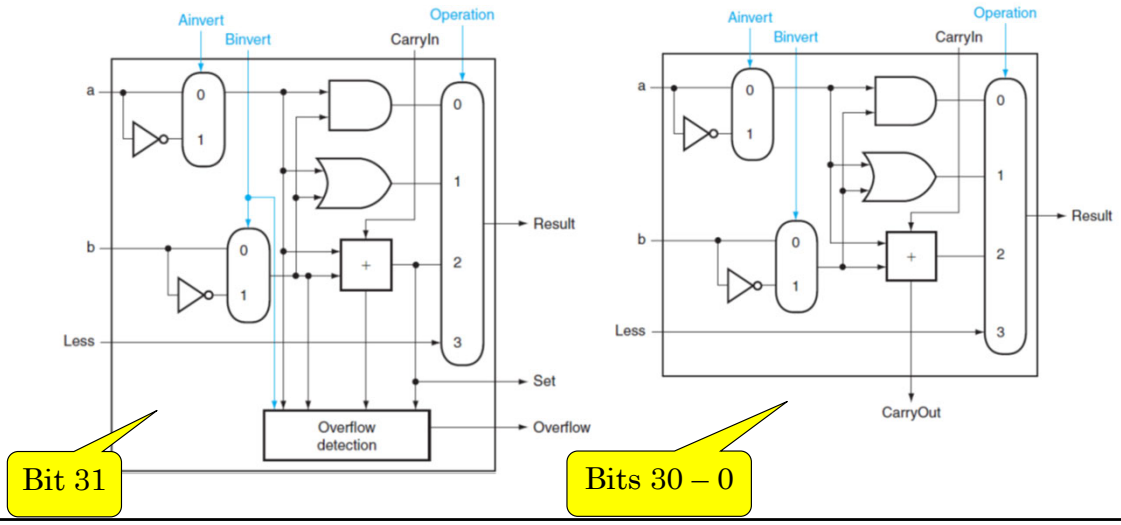
26

```
(bit c_out, bit result) ALU_1_bit(bit a, bit b, bit c_in, bit aInvert, bit bInvert, bit
operation[2])
{
    bit c_out, result
    bit enable = 1  // No está dibujado porque se sobreentiende
    bit operandoA, operandoB
    bit entradasMux2a1_a[2]
    bit entradasMux2a1_b[2]
    bit entradasMux4a1[4]
    entradasMux2a1_a[0] = a
    entradasMux2a1_a[1] = NOT a
    operandoA = mux2a1(entradasMux2a1_a, aInvert, enable)
    entradasMux2a1_b[0] = b
    entradasMux2a1_b[1] = NOT b
    operandoB = mux2a1(entradasMux2a1_b, bInvert, enable)
    entradasMux4a1[0] = operandoA AND operandoB
    entradasMux4a1[1] = operandoA OR operandoB
    (c_out, entradasMux4a1[2]) = sumadorCompleto_1_bit(operandoA, operandoB, c_in)
    entradasMux4a1[3] = undefined
    result = mux4a1(entradasMux4a1, operation, enable)
    return (c_out, result)
}
```

- La operación NOR es la negación de una suma lógica.
- Podríamos añadir una puerta NOR, pero vamos a reaprovechar la puerta AND gracias al teorema de De Morgan. Así nos ahorramos añadir al circuito una puerta lógica NOR, así como utilizar una entrada del multiplexor que selecciona el resultado.
  - Una suma negada es igual al producto de las negadas, es decir: *a* NOR *b* = (*a* + *b*)' = *a*' · *b*' = (NOT *a*) AND (NOT *b*).
  - La puerta AND y la puerta NOT de *b* ya las tenemos, pero nos falta la puerta NOT de *a*.
  - Añadimos también un multiplexor para la señal *a*. La señal de selección del nuevo multiplexor es A_{invert}.
- Operaciones realizadas:
  - *Operation* = 00, A_{invert} = 0 y B_{invert} = 0 → Producto lógico (AND). Se ignora c_{in}.
  - *Operation* = 00, A_{invert} = 1 y B_{invert} = 1 → Suma lógica negada (NOR). Se ignora c_{in}.
  - *Operation* = 01, A_{invert} = 0 y B_{invert} = 0 → Suma lógica (OR). Se ignora c_{in}.
  - *Operation* = 10, A_{invert} = 0, B_{invert} = 0 y c_{in} = 0 → Suma aritmética (*a* + *b*).
  - *Operation* = 10, A_{invert} = 0, B_{invert} = 1 y c_{in} = 1 → Resta aritmética (*a* – *b*).

OPERACIÓN DE ACTIVACIÓN CONDICIONAL: `slt`

`slt $rd, $rs, $rt` { `$rd` ← 0x00000001 si `$rs` < `$rt`; `$rd` ← 0x00000000 si `$rs` ≥ `$rt`



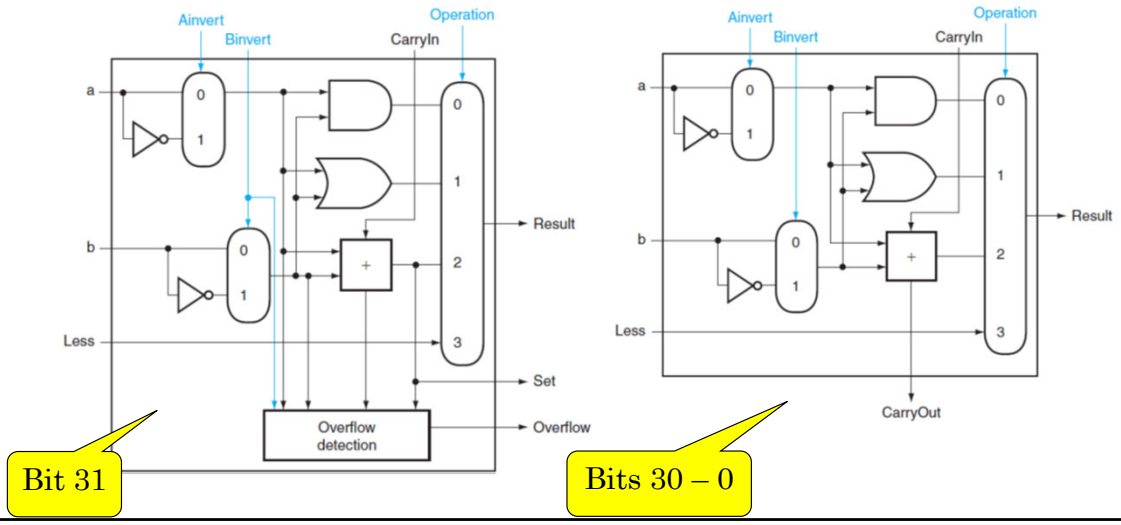
27

```
(bit c_out, bit result) ALU_1_bit_bits30a0(bit a, bit b, bit c_in, bit less, bit aInvert, bit bInvert, bit
operation[2])
{
    bit c_out, result
    bit enable = 1  // No está dibujado porque se sobreentiende
    bit operandoA, operandoB
    bit entradasMux2a1_a[2]
    bit entradasMux2a1_b[2]
    bit entradasMux4a1[4]
    entradasMux2a1_a[0] = a
    entradasMux2a1_a[1] = NOT a
    operandoA = mux2a1(entradasMux2a1_a, aInvert, enable)
    entradasMux2a1_b[0] = b
    entradasMux2a1_b[1] = NOT b
    operandoB = mux2a1(entradasMux2a1_b, bInvert, enable)
    entradasMux4a1[0] = operandoA AND operandoB
    entradasMux4a1[1] = operandoA OR operandoB
    (c_out, entradasMux4a1[2]) = sumadorCompleto_1_bit(operandoA, operandoB, c_in)
    entradasMux4a1[3] = less
    result = mux4a1(entradasMux4a1, operation, enable)
    return (c_out, result)
}
```

```
(bit result, bit set, bit overflow) ALU_1_bit_bit31(bit a, bit b, bit c_in, bit less, bit aInvert, bit bInvert,
bit operation[2])
{
    bit c_out, result
    bit enable = 1  // No está dibujado porque se sobreentiende
    bit operandoA, operandoB
    bit entradasMux2a1_a[2]
    bit entradasMux2a1_b[2]
    bit entradasMux4a1[4]
    entradasMux2a1_a[0] = a
    entradasMux2a1_a[1] = NOT a
    operandoA = mux2a1(entradasMux2a1_a, aInvert, enable)
    entradasMux2a1_b[0] = b
    entradasMux2a1_b[1] = NOT b
    operandoB = mux2a1(entradasMux2a1_b, bInvert, enable)
    entradasMux4a1[0] = operandoA AND operandoB
    entradasMux4a1[1] = operandoA OR operandoB
    (c_out, entradasMux4a1[2]) = sumadorCompleto_1_bit(operandoA, operandoB, c_in)
    set = entradasMux4a1[2]
    entradasMux4a1[3] = less
    result = mux4a1(entradasMux4a1, operation, enable)
    overflow = (a XNOR b) AND (a XOR entradasMux4a1[2])  // Equivalente: overflow = c_out XOR c_in
    return (result, set, overflow)
}
```

OPERACIÓN DE ACTIVACIÓN CONDICIONAL: `slt`

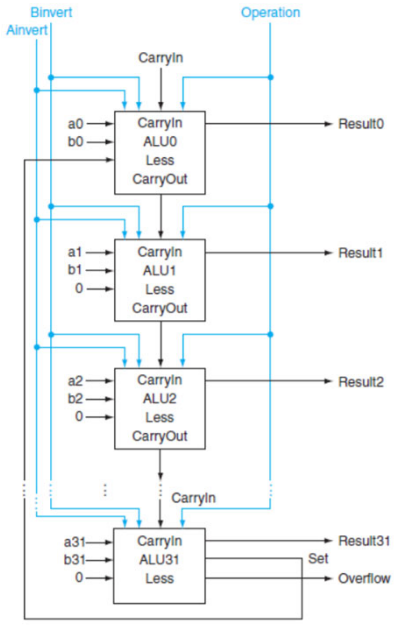
`slt $rd, $rs, $rt` { `$rd` ← `0x00000001` si `$rs` < `$rt`; `$rd` ← `0x00000000` si `$rs` ≥ `$rt`



28

- Para añadir esta nueva operación, daremos el salto en el diseño de la ALU a operandos de 32 bits en binario (complemento a 2).
- La operación "activar si es menor" slt (*set on less than*):
  - Compara el valor de dos registros ($rs y $rt).
  - Si $rs < $rt, entonces el registro de salida ($rd) vale 0000 0000 0000 0000 0000 0000 0000 0001 = 0x00000001 (31 ceros y un 1 en el bit menos significativo). En caso contrario, vale 0000 0000 0000 0000 0000 0000 0000 0000 = 0x00000000 (32 ceros).
  - Se ha usado la notación 0x para indicar hexadecimal.
- La comparación de *a* con *b* se realiza mediante la resta $a - b$, sin almacenar el resultado.
  - Si se cumple que $a - b < 0$, entonces también se cumple que $a < b$, que es lo que nos interesa.
- Para incluir esta operación en nuestra miniALU, debemos darnos cuenta de que los bits del 30 al 0 son iguales, mientras que el bit 31 (el más significativo) se comporta diferente.
  - Bits 30-0:
    - Ampliamos el multiplexor a 4 entradas, tal que la entrada 3 es directamente una señal de entrada que hemos llamado *Less*.
  - Bit 31:
    - Ampliamos el multiplexor a 4 entradas, tal que la entrada 3 es directamente una señal de entrada que hemos llamado *Less*.
    - Pero también hay dos nuevas salidas:
      - *Set*, que corresponde al bit de suma del sumador completo, es decir, se trata del bit de signo de la resta $a - b$.
      - *Overflow*, que corresponde al desbordamiento de una operación aritmética.
- Como internamente haremos una resta en esta operación, el bit 31 del resultado (y por tanto *Set*) valdrá 1 si $a<b$, porque los datos están en complemento a 2 y es el bit de signo.
- Nótese que el bit de suma del sumador 31 está conectado a la entrada 2 del multiplexor, pero también se proporciona directamente como salida *Set*.
- Operaciones realizadas:
  - Operation = 00, A_{invert}=0 y B_{invert} = 0 → Producto lógico (AND). Se ignora c_{in}.
  - Operation = 00, A_{invert}=1 y B_{invert} = 1 → Suma lógica negada (NOR). Se ignora c_{in}.
  - Operation = 01, A_{invert}=0 y B_{invert} = 0 → Suma lógica (OR). Se ignora c_{in}.
  - Operation = 10, A_{invert}=0, B_{invert} = 0 y c_{in} = 0 → Suma aritmética ($a + b$).
  - Operation = 10, A_{invert}=0, B_{invert} = 1 y c_{in} = 1 → Resta aritmética ($a - b$).
  - Operation = 11, A_{invert}=0, B_{invert} = 1 y c_{in} = 1 → Activar si es menor.
- Con esta transparencia damos por finalizado el diseño jerárquico de la miniALU de 1 bit.

ALU DE 32 BITS



Resta: $c_{\text{in}} = B_{\text{invert}} = 1$

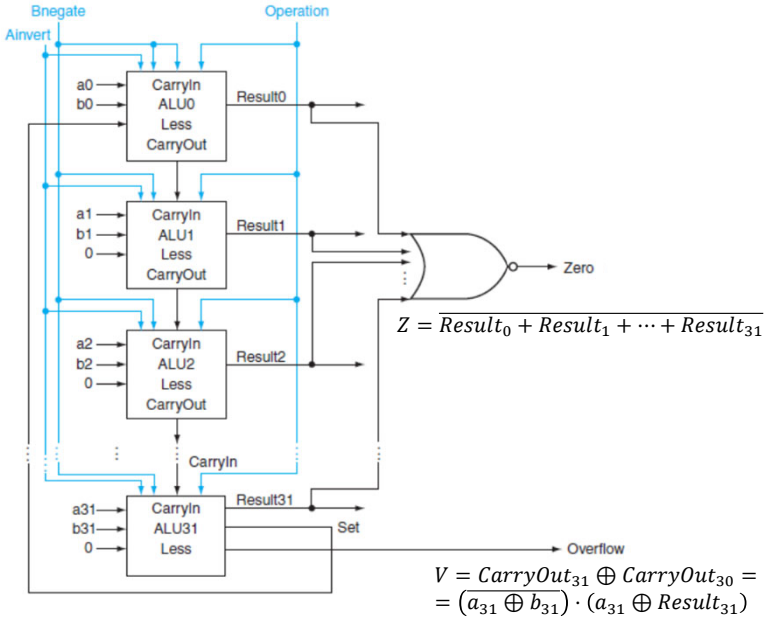
Suma: $c_{\text{in}} = B_{\text{invert}} = 0$

Ambas se pueden combinar en una única señal B_{negate}.

29

- Combinando adecuadamente las miniALU de 1 bit, se obtiene la ALU de 32 bits.
  - Cada miniALU recibe dos bits del mismo peso de *a* y de *b* y proporciona el bit del mismo peso del resultado.
  - El acarreo de salida de una miniALU se pasa al acarreo de entrada de la miniALU siguiente.
  - El acarreo de salida de la miniALU de peso 31 se ignora, ya que **nunca forma parte del resultado de una suma o de una resta para datos en complemento a 2**.
  - El acarreo de entrada de la miniALU de peso 0 es una entrada más del circuito (todavía no la hemos establecido de manera definitiva).
  - Todas las miniALU reciben la misma palabra de operación.
  - La salida *Set* de la miniALU de peso 31 es la entrada *Less* de la miniALU de peso 0.
  - El resto de las entradas *Less* de las otras miniALU están fijadas al valor 0.
- Optimización:
  - Siempre que la señal de acarreo de entrada c_{in} para la miniALU de peso 0 vale 0, la señal B_{invert} también vale 0. Y cuando la primera vale 1, la otra también.
  - Como ambas señales toman siempre los mismos valores, podemos unificarlas en una sola señal, que llamaremos a partir de ahora B_{negate} (ver siguiente transparencia).

Indicadores de resultado: Detectores de cero (Z) y de desbordamiento (V)

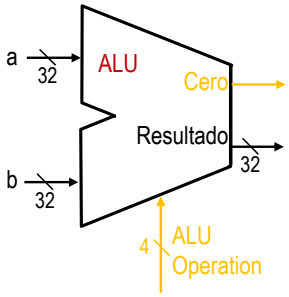


30

```
int n = 32
(bit result[n], bit overflow, bit zero) ALU_n_bits(bit a[n], bit b[n], bit aInvert, bit bNegate, bit
operation[2])
{
    bit result[n]
    bit overflow, zero
    bit set
    bit c_out[n-1]
    bit c_in0 = bNegate
    (c_out[0], result[0]) = ALU_1_bit_bits30a0(a[0], b[0], c_in0, set, aInvert, bNegate, operation)  // set lo
calcula la miniALU de peso 31 de manera concurrente a la miniALU de peso 0
    for (int i = 1; i<n-2; i++)
    {
        (c_out[i], result[i]) = ALU_1_bit_bits30a0(a[i], b[i], c_out[i-1], 0, aInvert, bNegate, operation)
    }
    (result[n-1], set, overflow) = ALU_1_bit_bit31(a[n-1], b[n-1], c_out[n-1], 0, aInvert, bNegate, operation)
    zero = result[0]
    for (int i = 1; i<n; i++)
    {
        zero = zero OR result[i]
    }
    zero = NOT zero
    return (result, overflow, zero)
}
```

- Los indicadores de resultado (*flags*) son salidas adicionales de la ALU que se activan o no en función del último resultado calculado por la ALU.
- Son útiles al ejecutar instrucciones de salto condicional (por ejemplo, beq).
- Para la ALU de MIPS vamos a añadir dos indicadores de resultado:
    - Detección de Cero (*Z*):
        - Se activa a 1 si todos los bits del resultado son 0.
        - Se calcula mediante la operación NOR de todos los bits de *R*, ya que esta función lógica solo devuelve un resultado 1 si todas sus entradas sin excepción valen 0.
        - En el dibujo hemos añadido una puerta NOR de 32 entradas (los 32 bits de resultado).
    - Detección de desbordamiento (*V*):
        - Se activa a 1 si hubo desbordamiento en una suma o una resta aritméticas para datos en complemento a 2.
        - Se calcula aplicando la operación XOR a los acarreos de salida de pesos 30 y 31, ya que, si son diferentes (XOR), significa que el resultado es erróneo.
        - También se puede calcular comprobando que los bits de signo de *a* y *b* son iguales (XNOR) y a la vez (AND) diferentes del bit de signo del resultado (XOR). Estos bits de signo son los de peso 31.

RESUMEN ALU



Las cuatro señales de control que han aparecido hasta ahora las vamos a agrupar en una única palabra de 4 bits llamada *ALU Operation*. Según su valor, la ALU sabe qué operación debe realizar.

ALU Operation

A_{invert}	B_{negate}	$op1$	$op0$	Función
0	0	0	0	and
0	0	0	1	or
0	0	1	0	add
0	1	1	0	sub
0	1	1	1	slt
1	1	0	0	nor

31

```
int n = 32
(bit resultado[n], bit cero) ALU_n_bits(bit a[n], bit b[n], bit ALU_Operation[4])
{
    bit resultado[n]
    bit cero, overflow
    bit aInvert = ALU_Operation[3]
    bit bNegate = ALU_Operation[2]
    bit op[2]
    op[1] = ALU_Operation[1]
    op[0] = ALU_Operation[0]
    (resultado, overflow, cero) = ALU_n_bits(a, b, aInvert, bNegate, op)
    return (resultado, cero)  // Ignoramos overflow
}
```

- Y aquí termina el diseño modular de una ALU de datos de 32 bits.
  - Esta ALU la utilizaremos de nuevo en el tema 3.
    - En dicho tema usaremos una versión de la ALU sin indicador de desbordamiento (V). Por eso hemos ignorado la salida overflow.
- Por simplicidad, la señal *Operation*, que controla el multiplexor interno de cada miniALU, la llamaremos a partir de ahora *op* (señal de 2 bits).
- Reordenamos los bits de control y los renombramos como *ALU Operation*, una señal de 4 bits.
  - Peso 3: A_{invert}, que controla el multiplexor que elige entre un bit de *a* y su negado.
  - Peso 2: B_{negate}, que controla el multiplexor que elige entre un bit de *b* y su negado.
  - Pesos 1 y 0: *op*1 y *op*0, las señales de selección de cada uno de los multiplexores de las miniALU.
- Nótese que para la operación AND y para la NOR se aplica la operación *op* = 00 (=producto lógico), si bien para la AND las dos señales están afirmadas ($A_{invert} = B_{negate} = 0$), mientras que, para la NOR, están negadas ($A_{invert} = B_{negate} = 1$).
- Por otro lado, como ya hemos dicho, sumar es igual que restar, luego los bits de operación son iguales (*op* = 10). Sin embargo, para la suma las dos señales están afirmadas ($A_{invert} = B_{negate} = 0$), mientras que, para la resta, las señales valen $A_{invert} = 0$, $B_{negate} = 1$.
- Operaciones realizadas por la ALU completa:
  - *ALU Operation* = 0000 → Producto lógico (AND).
  - *ALU Operation* = 0001 → Suma lógica (OR).
  - *ALU Operation* = 0010 → Suma aritmética (*a* + *b*).
  - *ALU Operation* = 0110 → Resta aritmética (*a* – *b*).
  - *ALU Operation* = 0111 → Activar si es menor.
  - *ALU Operation* = 1100 → Suma lógica negada (NOR).

Universidad Rey Juan Carlos

CONTENIDOS

- Introducción
- Conceptos previos
- Circuitos aritméticos-lógicos básicos
- Construcción incremental de una ALU sencilla
- Otros circuitos aritméticos-lógicos

OTROS CIRCUITOS ARITMÉTICOS-LÓGICOS

- Desplazadores
- Extensores de signo
- Otros circuitos

OPERACIONES DE DESPLAZAMIENTO

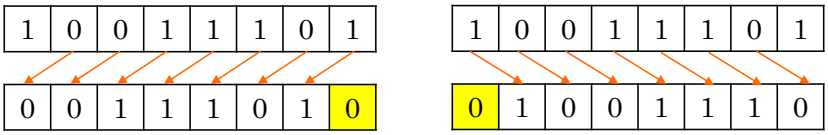
- Los **desplazamientos** son operaciones a nivel de bits en las que estos se mueven hacia la derecha o la izquierda un número determinado de posiciones.
- Tipos:
  - **Desplazamiento lógico:** Los bits que entran por los extremos son 0.
  - **Desplazamiento aritmético:** Si es hacia la derecha, se repite el bit de signo; si es hacia la izquierda, entra un 0.
  - **Rotación:** Los bits que salen por un extremo entran por el otro.
- En algunos casos, el bit que entra procede de un bit particular del registro de estado del procesador.

34

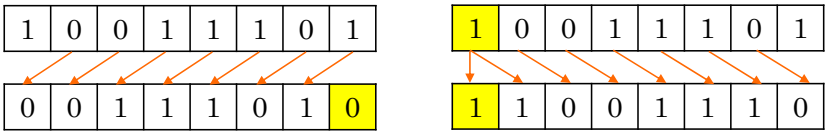
- Desplazamiento: cambiar la posición de los bits de un dato y moverlos hacia la izquierda o hacia la derecha según un cierto tamaño de desplazamiento.
- Desplazar hacia la derecha equivale a una división entera entre 2, mientras que desplazar hacia la izquierda es una multiplicación por 2 (siempre y cuando no perdamos bits por los extremos).
- La operación realizada depende del sistema de representación numérica:
  - Si los datos son en binario puro, el desplazamiento es lógico (entran ceros por los extremos).
  - Si son en complemento a 2, el desplazamiento es aritmético para conservar el signo (en el caso de desplazar hacia la derecha).
- Cuando no deseamos perder los bits que salen por un extremo, se aplica una operación de rotación.

OPERACIONES DE DESPLAZAMIENTO DE BITS

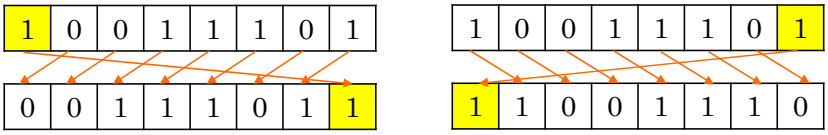
- **Lógico:** Los bits entrantes siempre son 0 (datos en binario puro).



- **Aritmético:** Se repite el bit de signo al desplazar a la derecha (datos en complemento a 2).



- **Rotación:** los bits que salen por un extremo entran por el otro.

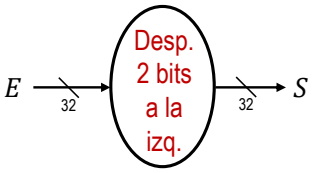


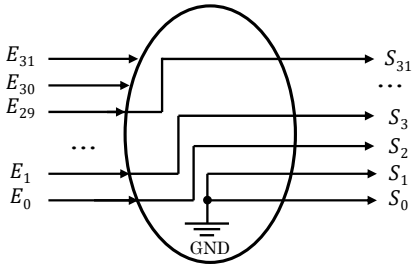
35

- En esta diapositiva vemos ejemplos de desplazamientos lógicos, aritmético y de rotación con un **desplazamiento de 1 posición** para datos de 8 bits.
- En color amarillo está remarcado el bit nuevo.
- Las flechas indican cómo se desplaza cada bit hacia su nueva posición.

EJEMPLO: DESPLAZADOR FIJO DE 2 POSICIONES HACIA LA IZQUIERDA

- Dato de entrada (E): 32 bits.
- Dato de salida (S): 32 bits.
- Desplazamiento:
  - Tamaño: 2 (fijo)
  - Dirección: hacia la izquierda.
  - Efecto producido: si los bits que se pierden por la izquierda son 0, se multiplica el dato de entrada por $2^2 = 4$.






```
bit s[32] desplazador2izquierda(bit e[32])
{
    bit s[32]
    s[0] = 0
    s[1] = 0
    for (int i = 2; i<32; i++)
    {
        s[i] = e[i-2]
    }
    return (s)
}
```

- En este circuito, se recibe un dato de 32 bits y se produce un valor a la salida también de 32 bits.
- El desplazamiento es fijo (2) y no se puede modificar.
  - Equivale a multiplicar el dato de entrada por 4.
- Los bits S_1 y S_0 son 0, ya que el valor de la salida es múltiplo de 4 (100 en binario).
- El bit S_i es igual que el bit E_{i-2}, para $2 \leq i \leq 31$.
- Este circuito lo usaremos en el tema 3.

OPERACIÓN DE EXTENSIÓN DE SIGNO

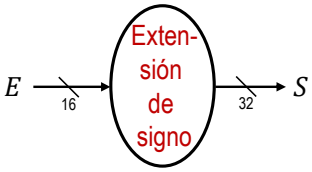
- La **extensión de signo** es una operación que permite expresar el mismo valor numérico con mayor número de bits.
  - Los bits nuevos se añaden por la izquierda.
- Depende del sistema de representación numérica utilizado:
  - **Binario puro:** se añaden ceros por la izquierda.
  - **Complemento a 2:** se repite el bit de signo del dato de entrada.

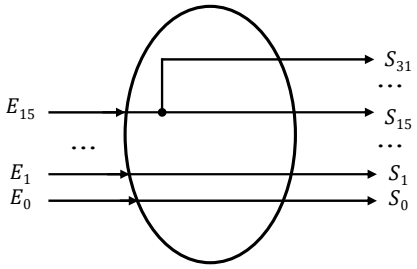
37

- Extensión de signo:
  - Operación que permite representar un valor numérico con un mayor número de bits.
  - Los bits adicionales se añaden por la izquierda.
- Para datos en binario puro, se añaden tantos ceros por la izquierda como sean necesarios.
- Para datos en complemento a 2, se repite el bit de signo del dato de entrada tantas veces como sea necesario.
  - De esta forma, se mantiene el signo del dato de entrada.

EJEMPLO: EXTENSOR DE SIGNO DE 16 A 32 BITS PARA DATOS EN COMPLEMENTO A 2

- Dato de entrada (E): 16 bits.
- Dato de salida (S): 32 bits.
- Como E está en complemento a 2, se repite el bit de signo (E_{15}).





38

```
bit s[32] extensorSigno16a32(bit e[16])
{
    bit s[32]
    for (int i = 0; i<16; i++)
    {
        s[i] = e[i]
    }
    for (int i = 16; i<32; i++)
    {
        s[i] = e[15]
    }
    return (s)
}
```

- En este circuito, se recibe un dato en complemento a 2 con 16 bits y se produce un resultado a la salida expresado con 32 bits con el mismo valor numérico que el dato original.
- Los bits S_{15}-S_0 son iguales a los bits E_{15}-E_0, respectivamente.
- Los bits adicionales (S_{31}-S_{16}) son iguales al bit E_{15} de la entrada, que es su bit de signo.
- Este circuito lo usaremos en el tema 3.

OTROS CIRCUITOS

- Sumadores paralelos con acarreo anticipado
- Multiplicadores (suma-desplazamiento)
- Divisores (división por restauración)
- Comparadores
- Circuitos aritméticos para otros sistemas de representación:
  - Complemento a 1
  - BCD (decimal codificado en binario)
  - Coma flotante (IEEE 754)
- Circuitos aritméticos en chips de escala de integración media (MSI).

39

- Estos son algunos ejemplos de otros circuitos aritmético-lógicos.
- Por falta de tiempo no los vamos a estudiar.