Circuitos Combinacionales Aritmético-Lógicos

Electrónica Digital - Grado en Ing. Electrónica, Robótica y Mecatrónica (US) Tema 5. Apuntes propios (worked-example primero, orientado a examen).

Mapa del tema

1. **Semisumador** (half adder) y **sumador completo** (full adder).
2. **Sumador paralelo con acarreo serie** (ripple-carry).
3. **Semirrestador, restador completo** y **sumador/restador en complemento a 2**.
4. **Sumador rapido con acarreo anticipado** (carry look-ahead): generacion G y propagacion P.
5. **Comparadores** de magnitud.
6. **La ALU** (Unidad Aritmetico-Logica).

> Notacion del tema: a_i, b_i bits de entrada, c_i acarreo de entrada, S_i suma, C_{i+1} acarreo de salida, t_{pd} retardo de propagacion de una puerta. Las figuras de cada bloque se generan con `schemdraw`; las tablas de verdad y verificaciones, con `numpy`.

```
Entorno listo: numpy 1.26.4 | schemdraw 0.22
```

1. Semisumador (Half Adder)

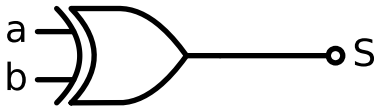
Suma **dos bits** a y b. Produce la suma S y el acarreo C, pero **no admite acarreo de entrada** (por eso "semi").

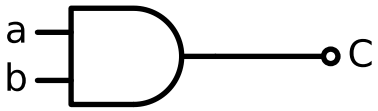
a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

De la tabla salen directamente las ecuaciones:

$$S = a \oplus b \qquad\qquad C = a \cdot b$$

La suma es un **XOR** (suma modulo 2) y el acarreo un **AND**. Es la pieza basica con la que se construye todo lo demas.






```
 a b | S C
-----+-----
 0 0 | 0 0
 0 1 | 1 0
 1 0 | 1 0
 1 1 | 0 1

OK: el par (C,S) representa siempre a+b en binario.
```

2. Sumador Completo (Full Adder)

Para sumar numeros de n bits hace falta encadenar el acarreo de un bit al siguiente:

$$\begin{array}{cccccc} & c_{n-1} & \cdots & c_2 & c_1 & c_0 \\ & a_{n-1} & \cdots & a_2 & a_1 & a_0 \\ + & b_{n-1} & \cdots & b_2 & b_1 & b_0 \\ \hline S_n & S_{n-1} & \cdots & S_2 & S_1 & S_0 \end{array}$$

El **sumador completo** suma **tres bits**: a_i, b_i y el acarreo de entrada c_i.

a_i	b_i	c_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Ecuaciones (a partir de los mapas de Karnaugh)

$$S_i = a_i \oplus b_i \oplus c_i$$

$$C_{i+1} = a_i b_i + a_i c_i + b_i c_i = a_i b_i + (a_i + b_i)\, c_i$$

La suma es un XOR de las tres entradas; el acarreo sale "por mayoria" (vale 1 cuando hay dos o mas unos).

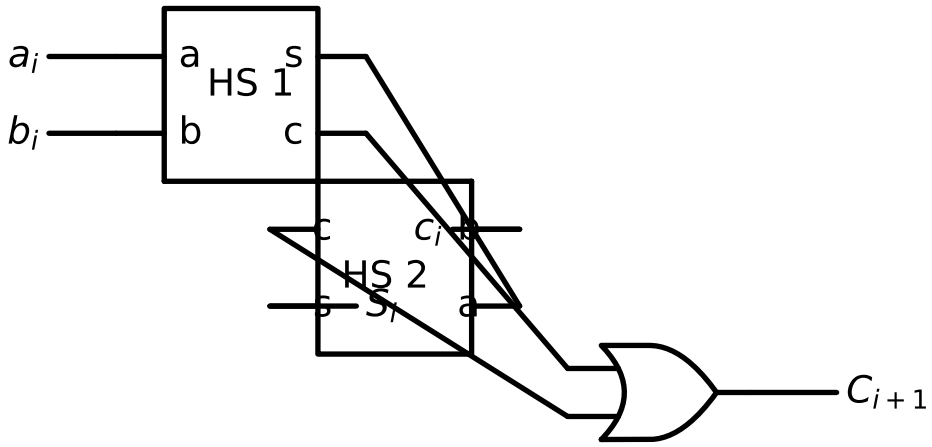
Construccion con dos semisumadores

El sumador completo clasico se arma con **dos semisumadores** y una **OR** final:

- El primer semisumador suma $a_i \oplus b_i$ y $a_i b_i$.
- El segundo suma $(a_i \oplus b_i)$ con c_i -> da S_i y $(a_i \oplus b_i)\, c_i$.
- La OR de los dos acarreos parciales da C_{i+1}:

$$C_{i+1} = a_i b_i + (a_i \oplus b_i)\, c_i$$

Retardo: cada semisumador aporta $2\, t_{pd}$. El camino critico al acarreo es $\approx 2\, t_{pd}$ por etapa. Este dato es clave para el sumador serie del apartado 3.




```
 a b cin | S Cout
---------+-------
 0 0  0  | 0   0
 0 0  1  | 1   0
 0 1  0  | 1   0
 0 1  1  | 0   1
 1 0  0  | 1   0
 1 0  1  | 0   1
 1 1  0  | 0   1
 1 1  1  | 1   1

Verificacion (C,S) == a+b+cin para los 8 casos: True
```

3. Sumador paralelo con acarreo serie (Ripple-Carry)

Encadenamos n sumadores completos: el acarreo de salida de cada etapa es el de entrada de la siguiente. La primera etapa recibe c_0 (normalmente 0 para suma pura).

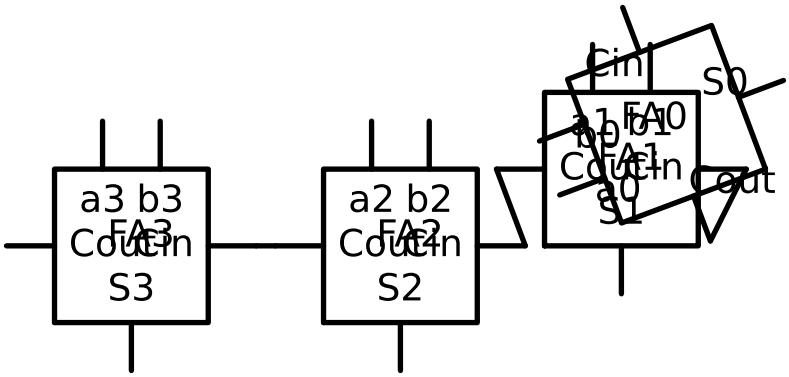
```
   a3 b3      a2 b2      a1 b1      a0 b0
    | |        | |        | |        | |
   [FA] <-c3- [FA] <-c2- [FA] <-c1- [FA] <- c0
    |          |          |          |
   S3         S2         S1         S0      (C4 = acarreo final /
overflow)
```

Retardo (lo que cae en examen)

El acarreo tiene que "rizar" (ripple) por toda la cadena. Si cada sumador completo tarda $2\,t_{pd}$ en propagar el acarreo:

$$t_{\text{total}} \approx 2n\,t_{pd}$$

Es **lento**: crece linealmente con n. Esto motiva el acarreo anticipado del apartado 4.




```
n=4 bits: combinaciones probadas = 256, errores = 0
Ejemplo: 13 + 11 = (8, 1) -> (suma 4 bits, Cout). 13+11=24=0b11000
```

4. Restadores y suma/resta en complemento a 2

4.1 Semirrestador

Calcula $R = a - b$ produciendo la resta R y el **acarreo de resta** (borrow) C.

a	b	R	C
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$R = a \oplus b \qquad C = \bar{a}\, b$$

4.2 Restador completo

Resta tres bits $R_i = a_i - b_i - c_i$ (con borrow de entrada):

$$R_i = a_i \oplus b_i \oplus c_i \qquad C_{i+1} = \bar{a_i}\, b_i + \bar{a_i}\, c_i + b_i\, c_i$$

Fijate: la **resta** R_i tiene la misma forma que la suma del sumador (triple XOR); solo cambia el acarreo.

4.3 Sumador/Restador en complemento a 2 - el truco practico

En complemento a 2, restar es **sumar el complemento a 2** del sustraendo:

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

donde $\overline{B}$ es el complemento a 1 (negacion bit a bit) y el $+1$ se mete como **acarreo de entrada** $c_0 = 1$.

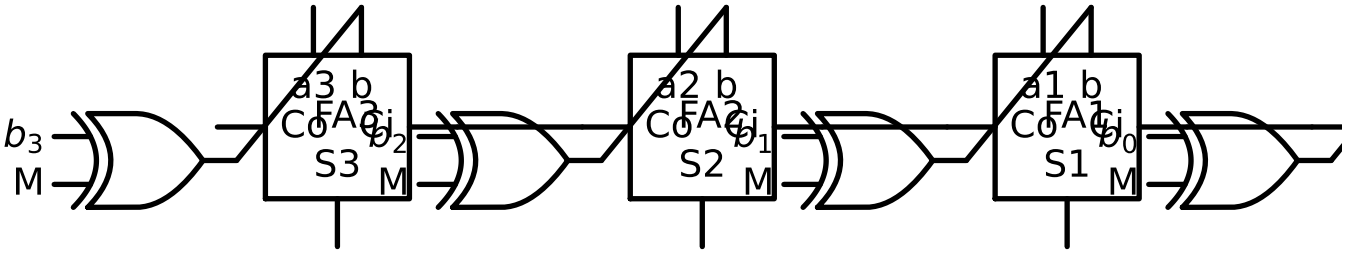
Circuito reconfigurable con una senal de control M (mode):

- Se intercala una **XOR** en cada bit b_i: la puerta calcula $b_i \oplus M$.
- Se conecta $c_0 = M$.

M	$b_i \oplus M$	c_0	**Operacion**
0	b_i	0	$A + B$ (suma)
1	$\overline{b_i}$	1	$A + \overline{B} + 1 = A - B$ (resta)

Con $M = 1$ obtenemos el complemento a 2 de B "gratis": las XOR niegan B y $c_0 = 1$ suma el 1. **Un solo bloque hace suma y resta.**

Overflow (signo): en complemento a 2 hay desbordamiento si el acarreo que entra al bit de signo y el que sale de el difieren: $V = C_n \oplus C_{n-1}$.




```
5+3               : resultado= -8  overflow=1
5-3               : resultado=  2  overflow=0
3-5               : resultado= -2  overflow=0
7+1 -> overflow   : resultado= -8  overflow=1
```

5. Sumador rapido: acarreo anticipado (Carry Look-Ahead)

El problema del ripple-carry es que el acarreo se propaga en serie ($2n\,t_{pd}$). La idea del **acarreo anticipado** es **calcular todos los acarreos en paralelo** a partir de las entradas, sin esperar a las etapas previas.

Generacion y propagacion

Partimos de $C_{i+1} = a_i b_i + (a_i + b_i)\, c_i$ y definimos:

- **Generacion**: $G_i = a_i\, b_i$ - la etapa genera acarreo "por si misma".
- **Propagacion**: $P_i = a_i + b_i$ - la etapa deja pasar el acarreo de entrada.

$$\boxed{C_{i+1} = G_i + P_i\, c_i} \qquad\qquad S_i = a_i \oplus b_i \oplus c_i$$

Todos los G_i y P_i se calculan en $1\,t_{pd}$ (dependen solo de a_i, b_i).

Expansion recursiva

Sustituyendo recursivamente cada c_i:

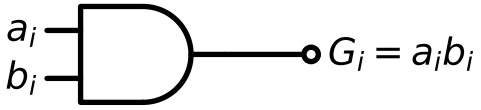
$$\begin{aligned} C_1 &= G_0 + P_0\, c_0 \\ C_2 &= G_1 + P_1 G_0 + P_1 P_0\, c_0 \\ C_3 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0\, c_0 \\ C_{i+1} &= G_i + P_i G_{i-1} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i \cdots P_0\, c_0 \end{aligned}$$

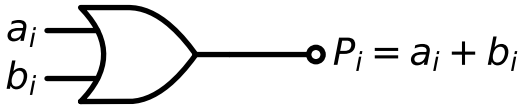
Cada acarreo es una **suma de productos** -> se calcula en $\approx 2\,t_{pd}$ (un nivel AND + un nivel OR), **independiente de** n (suponiendo puertas de muchas entradas).

Retardo total

$$t_{\text{total}} = \underbrace{1\,t_{pd}}_{G_i,P_i} + \underbrace{2\,t_{pd}}_{\text{acarreos}} + \underbrace{2\,t_{pd}}_{S_i} = 5\,t_{pd}\ (\text{constante})$$

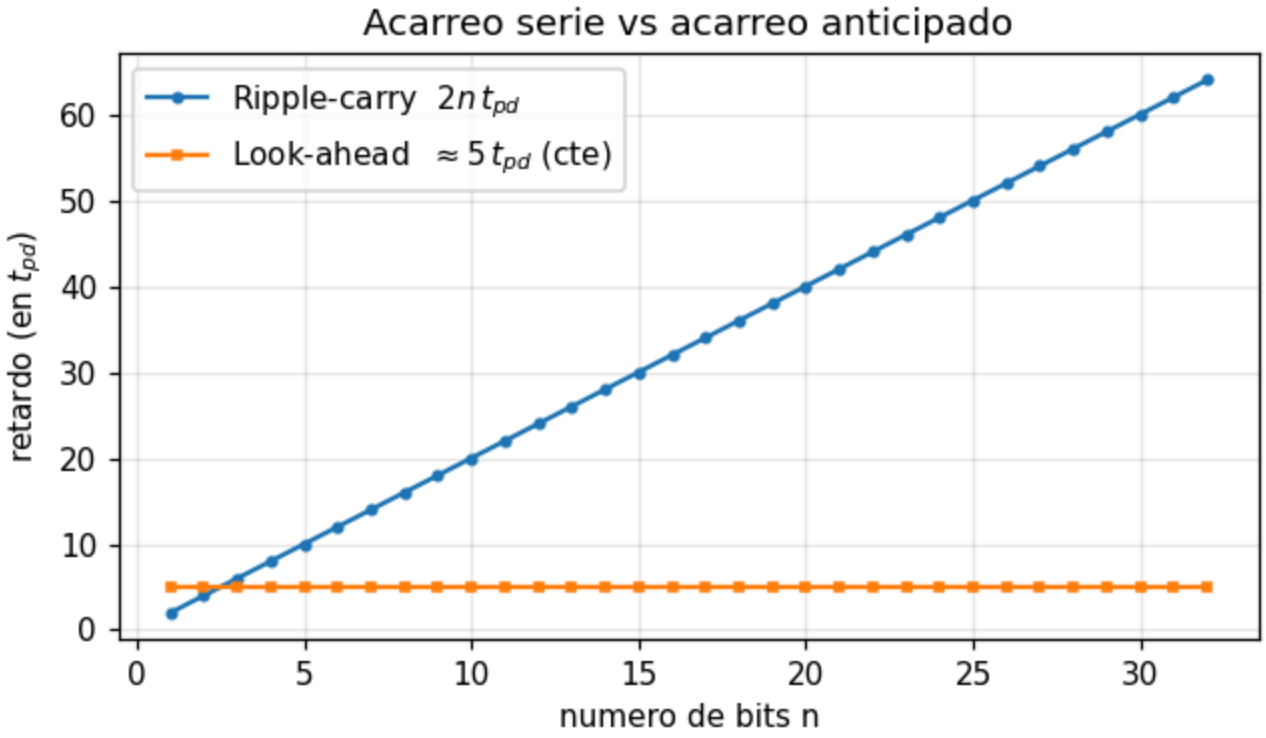
frente a los $2n\,t_{pd}$ del ripple. **Se gana velocidad a costa de mucha mas electronica** (las ecuaciones de C_i crecen con i).






```
CLA vs ripple para n=4: discrepancias en acarreos = 0

Ejemplo A=11 (1011), B=7 (0111):
  G (MSB..LSB) = [0, 0, 1, 1]
  P (MSB..LSB) = [1, 1, 1, 1]
  acarreos C0..C4 = [0, 1, 1, 1, 1]
```



6. Comparadores de magnitud

Un comparador de n bits compara A y B y activa una de tres salidas: $A > B$, $A = B$, $A < B$.

Comparador de 1 bit

a	b	$A = B$	$A > B$	$A < B$
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

$$(A{=}B) = \overline{a \oplus b} \qquad (A{>}B) = a\,\bar{b} \qquad (A{<}B) = \bar{a}\,b$$

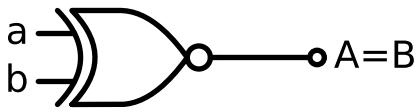
La igualdad es un **XNOR**.

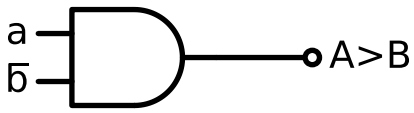
Comparador de n bits

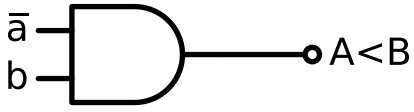
Se compara **del bit mas significativo al menos significativo**: en cuanto un bit difiere, ese bit decide. Definiendo $x_i = \overline{a_i \oplus b_i}$ (bits i iguales):

$$\begin{aligned}(A{>}B) &= a_{n-1}\bar{b}_{n-1} + x_{n-1}\,a_{n-2}\bar{b}_{n-2} + \dots \\ (A{=}B) &= x_{n-1}\,x_{n-2}\cdots x_0 \\ (A{<}B) &= \bar{a}_{n-1}b_{n-1} + x_{n-1}\,\bar{a}_{n-2}b_{n-2} + \dots\end{aligned}$$

> Ejemplo de chip real: el **74LS85** es un comparador de 4 bits con entradas en cascada para encadenar varios y comparar palabras mas anchas.







```
Comparador n=4: errores = 0 sobre 256 casos
  A= 9, B= 5 -> (A>B, A=B, A<B) = (1, 0, 0)
  A= 4, B= 4 -> (A>B, A=B, A<B) = (0, 1, 0)
  A= 2, B=13 -> (A>B, A=B, A<B) = (0, 0, 1)
```

7. La Unidad Aritmetico-Logica (ALU)

La **ALU** es el bloque combinacional que reune las operaciones aritmeticas y logicas del procesador. Recibe dos operandos A, B y unas **lineas de seleccion** que eligen la operacion; entrega un resultado F y banderas (flags) como acarreo/overflow.

Estructura tipica (dos secciones)

- **Seccion logica:** AND, OR, XOR, NOT, ... bit a bit, seleccionadas por un multiplexor.
- **Seccion aritmetica:** suma/resta en complemento a 2 (apartado 4), incremento, decremento, etc., construida sobre un sumador (a menudo con acarreo anticipado, apartado 5).

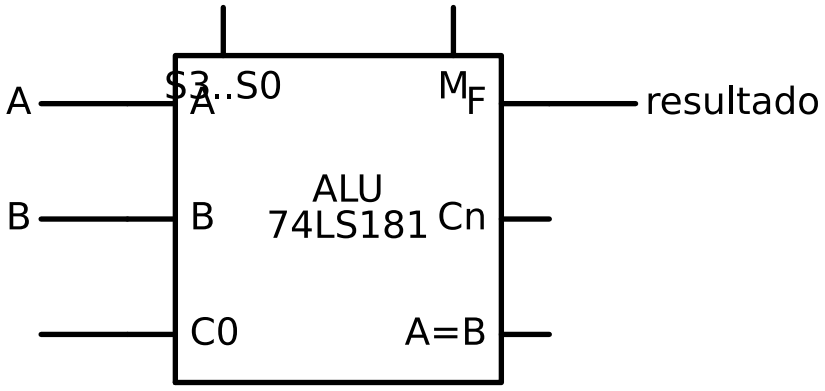
Un multiplexor final, gobernado por el bit modo M y los selectores S, elige que salida sale al bus.

Chip clasico: 74LS181

ALU de **4 bits** con **16 operaciones logicas** + **16 aritmeticas** seleccionables con $S_3S_2S_1S_0$ y el bit de modo M:

- $M = 1$: operaciones **logicas**.
- $M = 0$: operaciones **aritmeticas** (suma, resta, A, A+1, etc.), usando el acarreo C_n.

Genera ademas G y P de grupo para **acarreo anticipado en cascada** (encadenando varios 74LS181 con el 74LS182 como generador de acarreo look-ahead).



```
Seccion LOGICA (M=1):  A=1100, B=1010
   S=0 AND    -> F = 1000
   S=1 OR     -> F = 1110
   S=2 XOR    -> F = 0110
   S=3 NOT A  -> F = 0011

Seccion ARITMETICA (M=0):
   S=0 A+B  (A=9,B=5) -> F=14 (1110), Cout=0
   S=1 A-B  (A=9,B=5) -> F= 4 (0100), Cout=1
   S=2 A+1  (A=9,B=5) -> F=10 (1010), Cout=0
   S=3 A-1  (A=9,B=5) -> F= 8 (1000), Cout=1
```

8. Resumen para el examen

Bloque	Ecuaciones clave	Retardo
Semisumador	$S = a \oplus b$, $C = ab$	1-2 t_{pd}
Sumador completo	$S_i = a_i \oplus b_i \oplus c_i$, $C_{i+1} = a_i b_i + (a_i + b_i) c_i$	2 t_{pd}/etapa
Ripple-carry (n bits)	encadena $C_{i+1} \to c_i$	$\sim 2n$ t_{pd} (lento)
Restador	$R_i = a_i \oplus b_i \oplus c_i$, $C_{i+1} = \bar{a}_i b_i + \bar{a}_i c_i + b_i c_i$	-
Suma/resta C2	$A - B = A + \bar{B} + 1$; control M: $b_i \oplus M$, $c_0 = M$	-
Carry look-ahead	$G_i = a_i b_i$, $P_i = a_i + b_i$, $C_{i+1} = G_i + P_i c_i$	$\sim 5\, t_{pd}$ (cte)
Comparador	$A{=}B = \overline{a \oplus b}$ (XNOR), $A{>}B = a\bar{b}$, $A{<}B = \bar{a}b$	-
Overflow C2	$V = C_n \oplus C_{n-1}$	-
ALU	secciones logica + aritmetica + MUX (seleccion S, modo M)	-

Ideas que mas caen

- **Por que el ripple es lento** y como el **look-ahead** lo arregla calculando acarreos en paralelo (a cambio de mas puertas).
- **Restar = sumar el complemento a 2** -> el mismo hardware suma y resta con una senal M y XOR en B.

- **Distinguir** acarreo de salida C_n (suma sin signo) de **overflow** $V = C_n \oplus C_{n-1}$ (suma con signo en C2).
- El **74LS181** como ALU de referencia (16+16 operaciones, G/P de grupo para cascada).

Apuntes propios sobre las diapositivas oficiales ED5. Todas las ecuaciones se han verificado numericamente en las celdas de codigo.