Lógica digital

Arquitectura de Computadores – IIC2343

Yadran Eterovic S. (yadran@uc.cl)

2025-2

La **lógica digital** permite representar y manejar números binarios en un computador

Los números pueden ser representados en *base 2*: **números binarios**

Usamos sólo dos símbolos diferentes:

- 0 y 1 —*dígitos binarios* o ***bits***

... y la misma notación posicional

P.ej., **421** en base 2 se representa así

1 1 0 1 0 0 1 0 1

... ya que

$\mathbf{1} \times 2^8 + \mathbf{1} \times 2^7 + \mathbf{0} \times 2^6 + \mathbf{1} \times 2^5 + \mathbf{0} \times 2^4$

$+ \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 = \mathbf{421}$

bit 8 bit 7 ... bit 1 bit 0

$$\mathbf{4\ 2\ 1}_{10} \equiv \mathbf{1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1}_2$$

1×2^8 1×2^7 0×2^6 1×2^5 0×2^4 0×2^3 1×2^2 0×2^1 1×2^0

Un computador ejecuta aplicaciones complejas (ejecutando instrucciones muy simples)

Una aplicación típica, p.ej.,

- un sistema de mensajes
- un *browser* de la Web
- un procesador de texto

… tiene millones de líneas de código

… + llamadas a librerías de software que implementan funciones complejas

Un computador ejecuta aplicaciones complejas (ejecutando instrucciones muy simples)

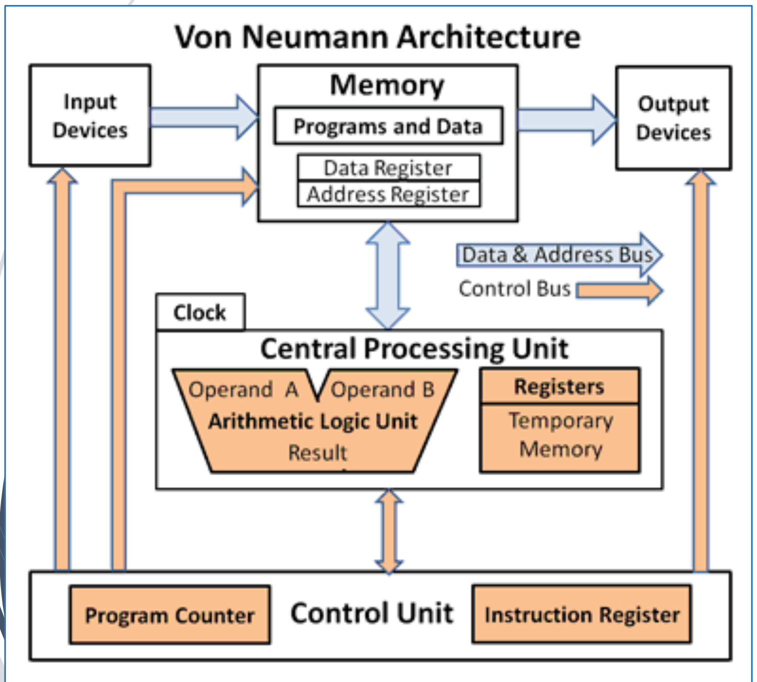
Una aplicación típica, p.ej.,

- un sistema de mensajes
- un browser de la Web
- un procesador de texto

... tiene millones de líneas de código

... + llamadas a librerías de software que implementan funciones complejas

¿?



Pero el hardware de un computador—la **lógica digital**—sólo puede ejecutar instrucciones de "bajo nivel" muy simples:

- sumar dos números
- revisar un número para ver si es cero
- copiar datos desde una parte de la memoria a otra

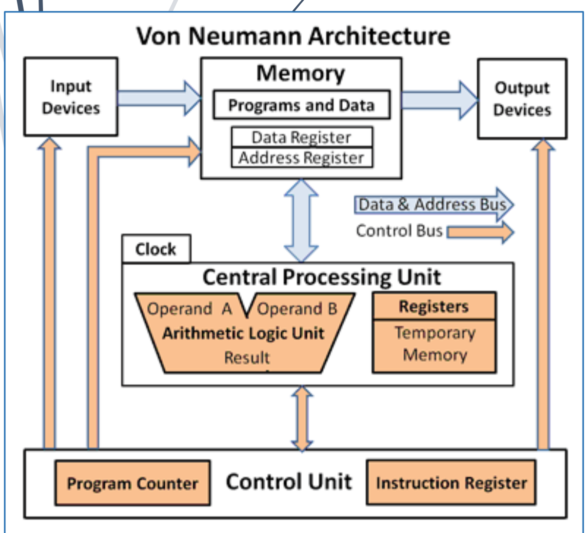
Un computador ejecuta aplicaciones complejas (ejecutando instrucciones muy simples)

Una aplicación típica, p.ej.,

- un sistema de mensajes
- un browser de la Web
- un procesador de texto

… tiene millones de líneas de código

… + llamadas a librerías de software que implementan funciones complejas



Para "hablarle" al hardware electrónico, hay que enviarle señales eléctricas

Las señales más simples y fáciles de distinguir son *on* y *off* :

- el alfabeto del computador tiene sólo dos letras, cuyos símbolos son los números 0 y 1
- cada letra es un **dígito binario**, o ***bit***

Los comandos —**instrucciones de máquina**— son secuencias de bits que el computador entiende y obedece:

- p.ej., 1001010100101110 le dice al computador que sume dos números

Pero el hardware de un computador—la **lógica digital**—sólo puede ejecutar instrucciones de "bajo nivel" muy simples:

- sumar dos números
- revisar un número para ver si es cero
- copiar datos desde una parte de la memoria a otra

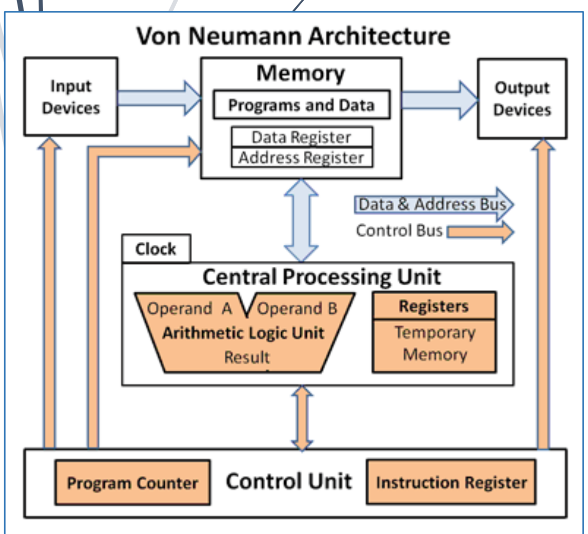
Un computador ejecuta aplicaciones complejas (ejecutando instrucciones muy simples)

Una aplicación típica, p.ej.,

- un sistema de mensajes
- un browser de la Web
- un procesador de texto

… tiene millones de líneas de código

… + llamadas a librerías de software que implementan funciones complejas



Pero el hardware de un computador—la **lógica digital**—sólo puede ejecutar instrucciones de "bajo nivel" muy simples:

- sumar dos números
- revisar un número para ver si es cero
- copiar datos desde una parte de la memoria a otra

Para "hablarle" al hardware electrónico, hay que enviarle señales eléctricas

Las señales más simples y fáciles de distinguir son *on* y *off* :

- el alfabeto del computador tiene sólo dos letras, cuyos símbolos son los números 0 y 1
- cada letra es un **dígito binario**, o ***bit***

Los comandos —**instrucciones de máquina**— son secuencias de bits que el computador entiende y obedece:

- p.ej., 1001010100101110 le dice al computador que sume dos números

Los primeros programadores se comunicaban con los computadores usando directamente estas secuencias de dígitos binarios

Los primeros programadores se comunicaban con los computadores usando directamente secuencias de dígitos binarios

Luego, los programadores inventaron notaciones más cercanas a nuestra forma de pensar; p.ej.:

- **`add A,B`** es la *representación simbólica* de la instrucción anterior

… e inventaron programas computacionales —el ensamblador o *assembler*— para traducir esta notación simbólica a la binaria:

- el *assembler* traduce **`add A,B`** a 1001010100101110

Los primeros programadores se comunicaban con los computadores usando directamente secuencias de dígitos binarios

Luego, los programadores inventaron notaciones más cercanas a nuestra forma de pensar; p.ej.:

- **add A,B** es la *representación simbólica* de la instrucción anterior

… e inventaron programas computacionales —el ensamblador o *assembler*— para traducir esta notación simbólica a la binaria:

- el *assembler* traduce **add A,B** a 1001010100101110

El lenguaje simbólico es el **lenguaje de ensamble**, o ***assembly***

El lenguaje binario es el **lenguaje de máquina**

```
swap(size_t v[], size_t k):
    size_t temp
    temp = v[k]
    v[k] = v[k+1]
    v[k+1] = temp
```

programa en lenguaje de alto nivel (C)

Y luego los programadores inventaron notaciones aún más cercanas a nuestra forma de pensar; p.ej.:

- el lenguaje C, ilustrado a través de la función **swap**

```
swap(size_t v[], size_t k):
    size_t temp
    temp = v[k]
    v[k] = v[k+1]
    v[k+1] = temp
```

programa en lenguaje de alto nivel (C)

compilador

```
swap:
    slli x6,x11,3
    add x6,x10,x6
    lw  x5,0(x6)
    lw  x7,4(x6)
    sw  x7,0(x6)
    sw  x5,4(x6)
    jalr x0,0(x1)
```

programa en **lenguaje assembly RISC-V**

Y luego los programadores inventaron notaciones aún más cercanas a nuestra forma de pensar; p.ej.:

- el lenguaje C, ilustrado a través de la función **swap**

… e inventaron programas computacionales —compiladores— para traducir esta notación al *assembly*

Y luego los programadores inventaron notaciones aún más cercanas a nuestra forma de pensar; p.ej.:

- el lenguaje C, ilustrado a través de la función **swap**

… e inventaron programas computacionales —compiladores— para traducir esta notación al *assembly*

programa en lenguaje de alto nivel (C)

```
swap(size_t v[], size_t k):
    size_t temp
    temp = v[k]
    v[k] = v[k+1]
    v[k+1] = temp
```

compilador

programa en **lenguaje assembly RISC-V**

```
swap:
    slli x6,x11,3
    add x6,x10,x6
    lw  x5,0(x6)
    lw  x7,4(x6)
    sw  x7,0(x6)
    sw  x5,4(x6)
    jalr x0,0(x1)
```

assembler

programa en **lenguaje de máquina** (binario) **RISC-V**

```
0000 … 010011
0000 … 110011
0000 … 000011
0000 … 000011
0000 … 100011
0000 … 100011
0000 … 100111
```

cada línea (instrucción) tiene 32 bits de largo programa en lenguaje de alto nivel (C)

```
swap(size_t v[], size_t k):
    size_t temp
    temp = v[k]
    v[k] = v[k+1]
    v[k+1] = temp
```

compilador

programa en **lenguaje assembly RISC-V**

```
swap:
    slli x6,x11,3
    add x6,x10,x6
    lw  x5,0(x6)
    lw  x7,4(x6)
    sw  x7,0(x6)
    sw  x5,4(x6)
    jalr x0,0(x1)
```

assembler

programa en **lenguaje de máquina** (binario) **RISC-V**

```
0000 … 010011
0000 … 110011
0000 … 000011
0000 … 000011
0000 … 100011
0000 … 100011
0000 … 100111
```

cada línea (instrucción) tiene 32 bits de largo

¡¡ Los programadores usaron el computador para que los ayudara a programar el propio computador !!

Un computador ejecuta aplicaciones complejas (ejecutando instrucciones muy simples)

Una aplicación típica, p.ej.,

- un sistema de mensajes
- un browser de la Web
- un procesador de texto

... tiene millones de líneas de código

... + llamadas a librerías de software que implementan funciones complejas

```
0000 … 010011
0000 … 110011
0000 … 000011
0000 … 000011
0000 … 100011
0000 … 100011
0000 … 100111
```

WhatsApp ≈ 355 MB

Google Chrome ≈ 1.7 GB

Microsoft Word ≈ 2.4 GB

lenguajes orientados o problemas

✓✓ lenguaje *assemly*

máquina del sistema operativo

✓✓ arquitectura del set de instrucciones (ISA)

✓✓ microarquitectura

✓ lógica digital

5) lenguajes de alto nivel, traducidos por compiladores, o interpretados

4) forma simbólica de los lenguajes de más abajo, traducida por el *assembler*

3) instrucciones adicionales, otra organización de memoria, ejecución concurrente → interpretado por el *sistema operativo*

2) instrucciones ejecutadas por el microprograma o circuitos del hardware

1) registros + *ALU* → *datapath*, controlada por hardware o microprogramada

0) compuertas (*gates*), álgebra de Boole, registros, el motor de computación; es el verdadero *hardware* del computador; está en la frontera entre la ciencia de la computación y la ingeniería electrónica

	lenguajes orientados o problemas	**5**) lenguajes de alto nivel, traducidos por compiladores, o interpretados
✓✓	lenguaje *assemly*	**4**) forma simbólica de los lenguajes de más abajo, traducida por el *assembler*
	máquina del sistema operativo	**3**) instrucciones adicionales, otra organización de memoria, ejecución concurrente → interpretado por el *sistema operativo*
✓✓	arquitectura del set de instrucciones (ISA)	**2**) instrucciones ejecutadas por el microprograma o circuitos del hardware
✓✓	microarquitectura	**1**) registros + *ALU* → *datapath*, controlada por hardware o microprogramada
✓	lógica digital	**0**) compuertas (*gates*), álgebra de Boole, registros, el motor de computación; es el verdadero *hardware* del computador; está en la frontera entre la ciencia de la computación y la ingeniería electrónica

Los (circuitos digitales de los) computadores modernos se construyen a partir de (miles de millones de copias de) *compuertas* combinándolas de innumerables formas

En un circuito digital hay sólo dos valores lógicos —**lógica binaria**:

- una señal eléctrica de entre 0 y 0.5 volts representa el valor binario 0
- una señal eléctrica de entre 1 y 1.5 volts representa el valor binario 1

Los (circuitos digitales de los) computadores modernos se construyen a partir de (miles de millones de copias de) *compuertas* combinándolas de innumerables formas

En un circuito digital hay sólo dos valores lógicos —**lógica binaria**:

- una señal eléctrica de entre 0 y 0.5 volts representa el valor binario 0
- una señal eléctrica de entre 1 y 1.5 volts representa el valor binario 1

Las **compuertas** (*gates*) —dispositivos electrónicos pequeños a su vez hechas de *transistores*— pueden calcular varias funciones a partir de estas señales

18

Los (circuitos digitales de los) computadores modernos se construyen a partir de (miles de millones de copias de) *compuertas* combinándolas de innumerables formas

En un circuito digital hay sólo dos valores lógicos —**lógica binaria**:

- una señal eléctrica de entre 0 y 0.5 volts representa el valor binario 0
- una señal eléctrica de entre 1 y 1.5 volts representa el valor binario 1

Las **compuertas** (*gates*) —dispositivos electrónicos pequeños a su vez hechas de *transistores*— pueden calcular varias funciones a partir de estas señales

Un **transistor** es un dispositivo semiconductor que puede usarse como amplificador y como interruptor de señales electrónicas:

- inventado en la práctica en 1947 (el concepto es de unos veinte años antes)
- **W. Shockley, J. Bardeen y W. Brattain ganaron el Premio Nobel de Física en 1956**

La palabra viene de la acción de transferencia-resistencia

Un **transistor** tiene tres terminales o conectores hacia el resto del circuito:

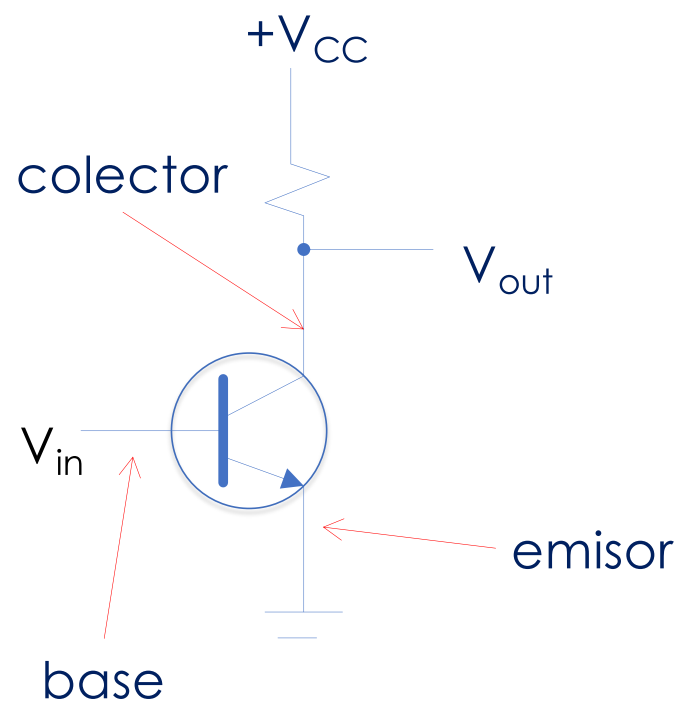
- *colector*, *base* y *emisor*

Cuando el voltaje de entrada, V_{in} , es menor que un cierto valor crítico, el transistor se apaga y actúa como una resistencia infinita:

- el voltaje de salida, V_{out} , es igual a V_{cc} , un voltaje regulado externamente (p.ej., 1.5 volts)

Cuando V_{in} excede el valor crítico, el transistor se enciende y actúa como un alambre:

- V_{out} se hace 0

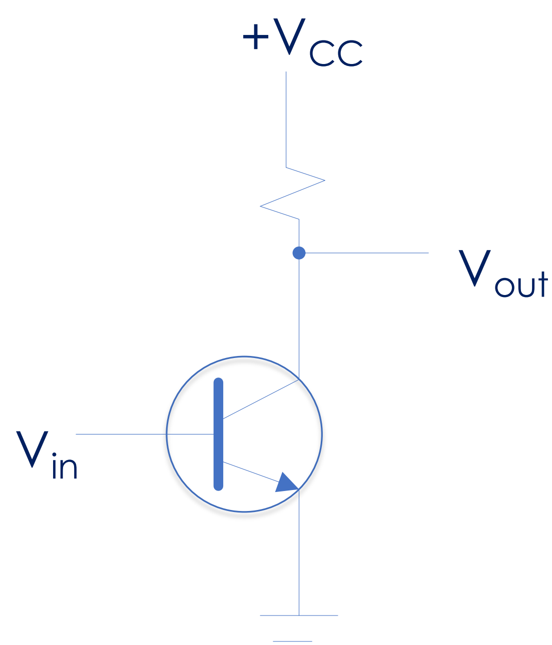


Representación esquemática relativamente estándar de un transistor

Un transistor por sí solo actúa como un ***inversor***:

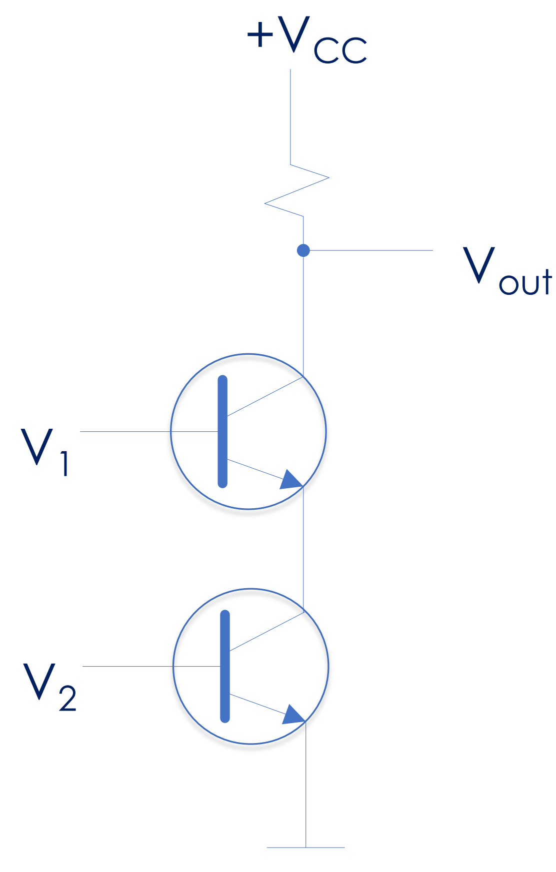
- si V_{in} es 0, entonces V_{out} es 1, y viceversa
- no es instantáneo

  … pero el tiempo que toma cambiar de un estado al otro es un nanosegundo o menos
- (corresponde, como vamos a ver, a la compuerta lógica **NOT**)



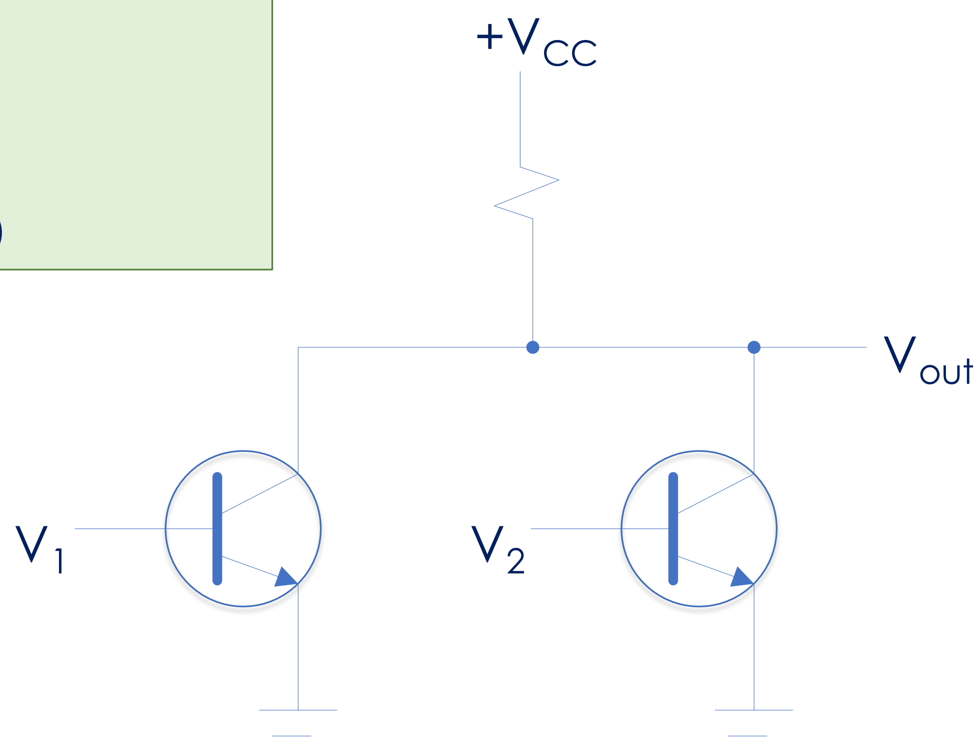
Dos transistores conectados *en serie* (el emisor del primero es conectado al colector del segundo) forman un circuito con la siguiente propiedad:

- basta que uno de los transistores actúe como una resistencia infinita para que el par de transistores sea una resistencia infinita
- V_{out} es 0 si y sólo si ambos voltajes de entrada V_1 y V_2 son 1
- en cualquier otro caso, V_{out} es 1
- (corresponde a la compuerta lógica **NAND**)



Dos transistores conectados en paralelo (los colectores están conectados entre ellos) forman un circuito con la siguiente propiedad:

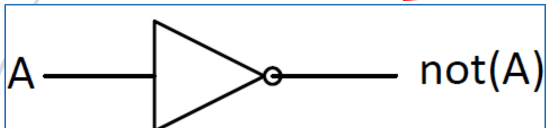
- basta que uno de los transistores actúe como un alambre para que el par de transistores sea un alambre
- V_{out} es 1 si y sólo si ambos V_1 y V_2 son 0
- en cualquier otro caso, V_{out} es 0
- (corresponde a la compuerta lógica **NOR**)



Los tres circuitos anteriores son los más simples de implementar:

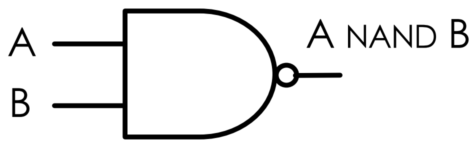
- sólo requieren de uno o dos transistores

... forman tres ***compuertas lógicas*** fundamentales: **NOT**, **NAND** y **NOR**



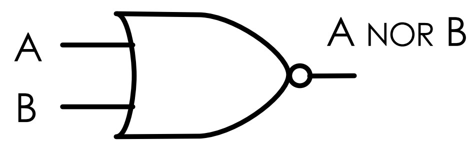
A	not(A)
0	1
1	0

Compuerta **NOT**:
output = $\bar{A} = \neg A$



A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Compuerta **NAND**:
output = $\overline{A \bullet B}$

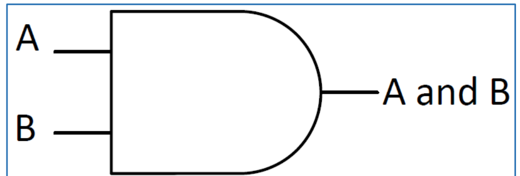


A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Compuerta **NOR**:
output = $\overline{A + B}$

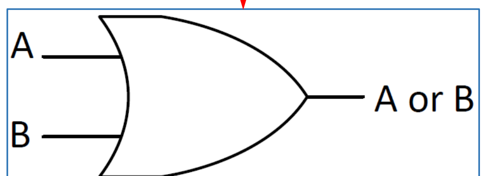
Ahora, si conectamos un inversor (una compuerta NOT) a la salida de una compuerta NAND, el nuevo circuito se llama una compuerta **AND**

... y si conectamos una compuerta NOT a la salida de una compuerta NOR, obtenemos una compuerta **OR**



A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Compuerta **AND**:
output = *A* • *B* = *A* ∧ *B*



A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Compuerta **OR**:
output = *A* + *B* = *A* ∨ *B*

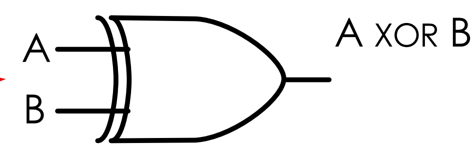
Si bien las compuertas **NAND** y **NOR** son más simples de implementar ... conceptualmente es más fácil trabajar con las compuertas **AND** y **OR**

La otra compuerta popular en la práctica es la compuerta **XOR**

Se puede construir a base de compuertas AND, OR y NOT:

$$A \oplus B = (\bar{A} \bullet B) + (A \bullet \bar{B})$$

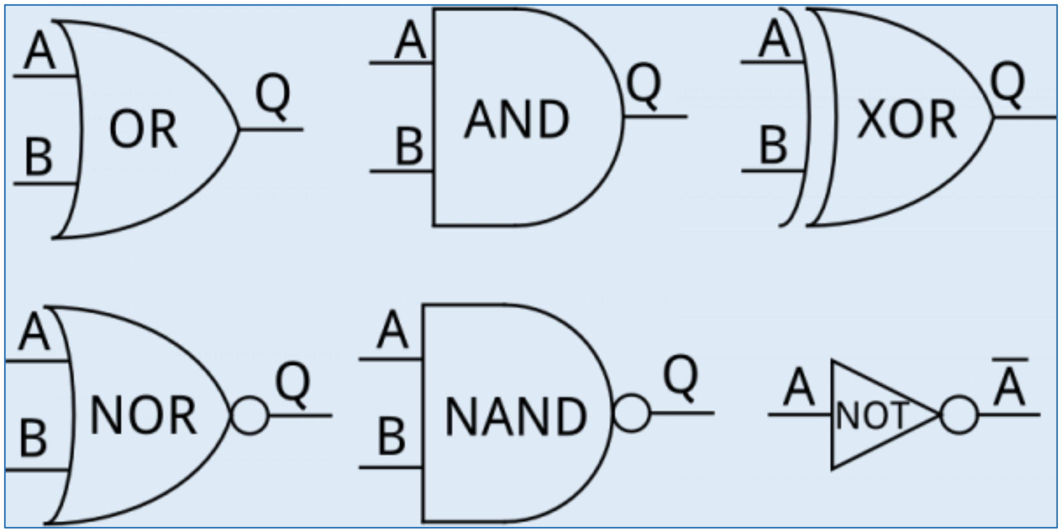
(… o bien sólo a base de compuertas NAND, o sólo a base de compuertas NOR)



A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Compuerta **XOR**:
output = $A \oplus B$

Los símbolos para representar las compuertas



Las **tablas de verdad**, de 1s y 0s, debajo de cada compuerta en las diaps. 23, 24 y 25 representan la **función** (relación entre los *inputs* y el *output*) que calcula cada compuerta

Las compuertas **NAND** y **NOR** son compuertas *completas* → cualquier función de Boole puede ser calculada usando sólo compuertas **NAND** o también sólo compuertas **NOR**

G. Boole: matemático inglés del siglo XIX

Empleamos el **álgebra** (de *switching*) **de Boole** para describir mediante ecuaciones lógicas las funciones lógicas correspondientes a los circuitos que pueden ser construidos combinando compuertas

Las variables y funciones pueden tomar **sólo los valores 0 y 1**

… y típicamente hay **tres operadores**:

OR ("o" lógico —*disjunción*): $A + B$ es 1 si cualquiera de las variables es 1

AND ("y" lógico —*conjunción*): $A \bullet B$ es 1 sólo si ambos inputs son 1

NOT (*negación*): $\bar{A}$ es 1 sólo si el input es 0

Leyes del álgebra de Boole

de identidad: $A + 0 = A$ y $A \bullet 1 = A$

del uno y del cero: $A + 1 = 1$ y $A \bullet 0 = 0$

Inversas: $A + \bar{A} = 1$ y $A \bullet \bar{A} = 0$

Conmutativas: $A + B = B + A$ y $A \bullet B = B \bullet A$

Asociativas: $A + (B + C) = (A + B) + C$ y $A \bullet (B \bullet C) = (A \bullet B) \bullet C$

Distributivas: $A \bullet (B + C) = (A \bullet B) + (A \bullet C)$ y $A + (B \bullet C) = (A + B) \bullet (A + C)$

de De Morgan: $\overline{A \bullet B} = \bar{A} + \bar{B}$ y $\overline{A + B} = \bar{A} \bullet \bar{B}$

Una función de Boole tiene una o más variables de entrada (*inputs*)

... y produce un resultado (*output*)

... que depende sólo de los valores de las variables de entrada —**lógica combinacional**

P.ej., las diaps. 23, 24 y 25 muestras seis funciones de Boole:

- las funciones correspondientes a las compuertas NOT (una variable de entrada: *A*),

  ... NAND, NOR, AND, OR y XOR (dos variables de entrada c/u: *A* y *B*)

  ... especificadas mediante sendas **tablas de verdad**

Para ver la aplicabilidad de las compuertas y funciones, diseñemos un circuito que sume números binarios

Las reglas para sumar dos sumandos de un bit c/u son las siguientes:

0 + 0 = 0 , sin reserva (*carry*) (la reserva es 0)

0 + 1 = 1 + 0 = 1 , sin reserva (la reserva es 0)

1 + 1 = 10 , es decir, el dígito correspondiente a la suma es 0 y hay una reserva (carry) de 1 (que pasa a la próxima posición a la izquierda)

P.ej., para sumar dos números binarios de cuatro dígitos cada uno, 1011 + 0110, escribo los números uno debajo del otro (alineados, tal como si estuviera sumando números decimales):

$$
\begin{array}{cr}
 & 1\ 0\ 1\ 1 \\
+ & \underline{0\ 0\ 1\ 0} \\
= & \mathbf{1\ 1\ 0\ 1}
\end{array}
$$

… y voy sumando pares de dígitos en una misma columna (de un mismo color), a partir de la columna de más a la derecha:

1 + 0 = **1** y no produce reserva

1 + 1 = **0** pero **produce una reserva de 1** que pasa a la columna de la izquierda

0 + 0 = **0** pero **hay que sumarle la reserva de 1** que viene desde la columna de la derecha → 0 + 1 = 1 y no produce reserva

1 + 0 = **1** y no produce reserva

Tabla de verdad y **circuito digital** —llamado ***half adder***— de un sumador de un un bit, con dos inputs, *A* y *B*, y dos outputs, *S* y *C*

Por lo tanto, cuando se suman dos dígitos, *A* y *B*, se producen dos resultados

$\Longrightarrow$ un circuito sumador de (sumandos de) un bit tiene dos outputs:

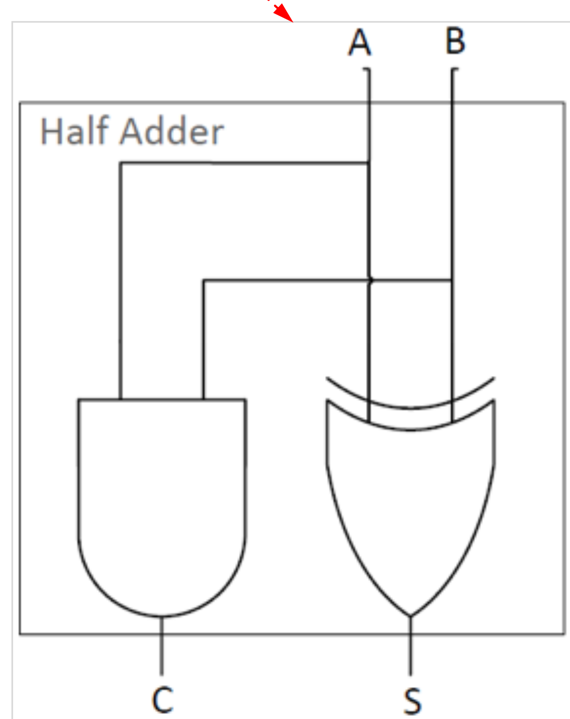
un bit, *S*, correspondiente a la suma

un bit, *C*, correspondiente a la reserva

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

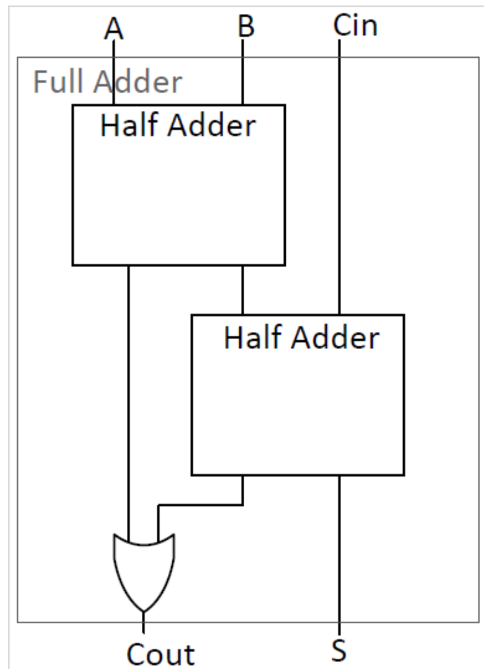
$A \bullet B$

$A \oplus B$



Un ***full adde***r —para sumandos de un bit—es construido apartir de dos *half adders*

Suma tres inputs: *A*, *B* y la reserva, C_{in} , que viene desde la derecha …

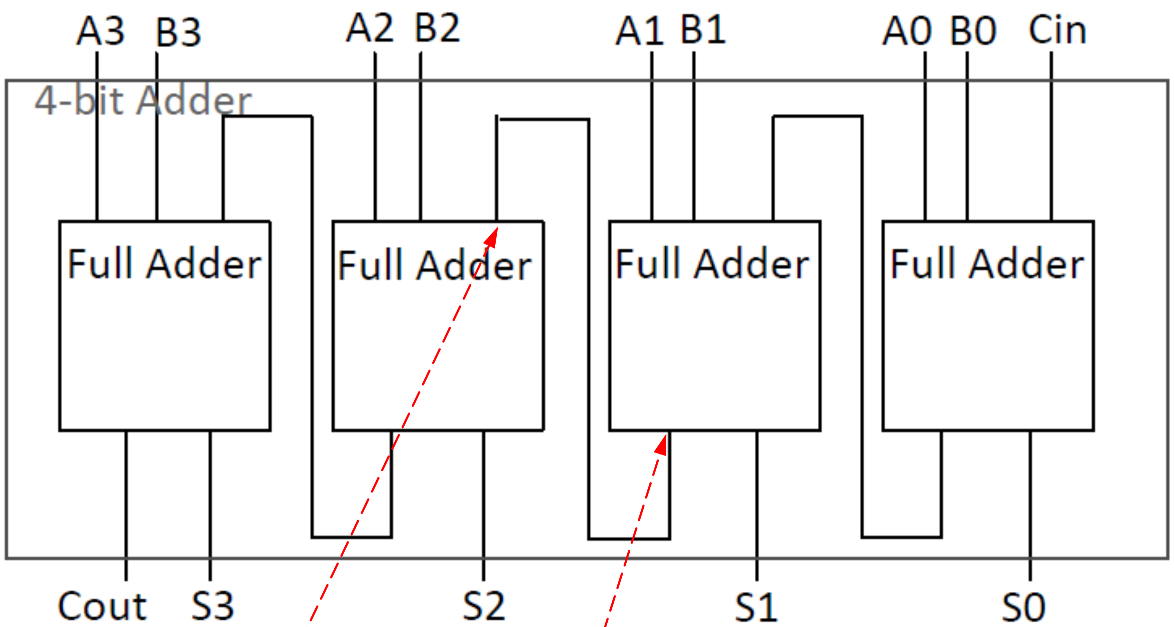


… y produce dos outputs: la suma, *S*, y la reserva hacia la izquierda, C_{out}

Un (circuito) sumador de sumandos de 4 bits

$$A_3A_2A_1A_0 + B_3B_2B_1B_0$$

… necesita 4 *full adders*, conectados como se muestra a continuación



el C_{out} del *full adder* para el bit *i*-ésimo se conecta …

… al C_{in} del *full adder* para el bit (i+1)-ésimo