

DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 2 - Operaciones Aritméticas y Lógicas

Profesor: Germán Leandro Contreras Sagredo

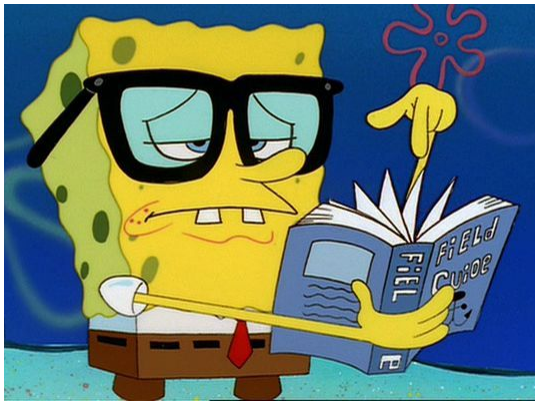
Objetivos de la clase

- Conocer un poco de la historia de la evolución de los computadores.
- Aprender sobre componentes, circuitos eléctricos y su relación con la lógica booleana.
- Aprender a realizar operaciones aritméticas y lógicas con circuitos eléctricos.
- Realizar ejercicios que consoliden los conocimientos anteriores.

Hasta ahora…

Ya tenemos una representación binaria para números enteros que nos permite realizar operaciones aritméticas simples. ¿Cómo traspasamos esto a nuestra“máquina programable” vía *hardware*?

Para ver cómo… aprendamos un *poquito* de historia antes.



Computadores en el pasado

- **Analógicos:** Procesan señales continuas manipulables.
- Realizan operaciones y/o simulaciones a través de la modificación de estas señales (representable en cantidades de voltaje, temperatura, presión, etc.).
- No muy confiables: ruido y descalibración afectan la precisión.



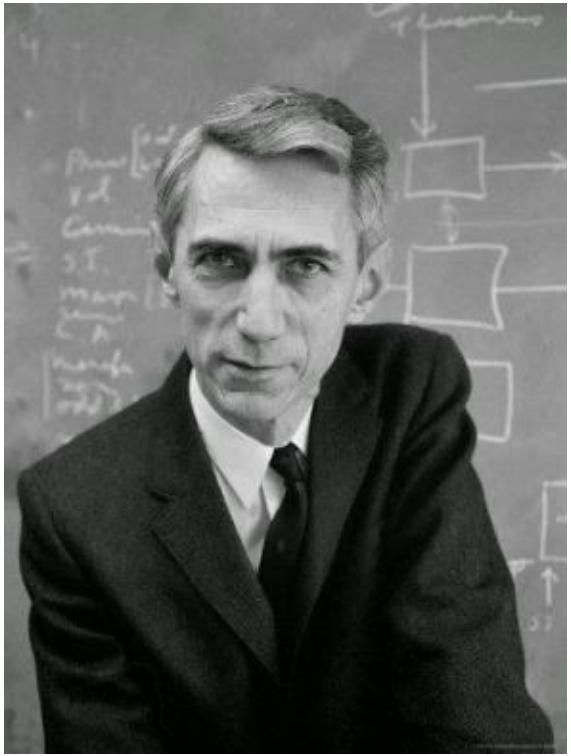


Computadores en el pasado

- **Claude Shannon:** Padre de la Teoría de la Información.

- Demuestra, a través de su teoría, que es más eficiente transformar las fuentes de datos (incluyendo análogas) en bits para una transmisión confiable.

- Una transformación digital nos llevará a tener una comunicación más confiable.



De lo analógico a lo digital

Para esta transformación, debemos seguir tres pasos:

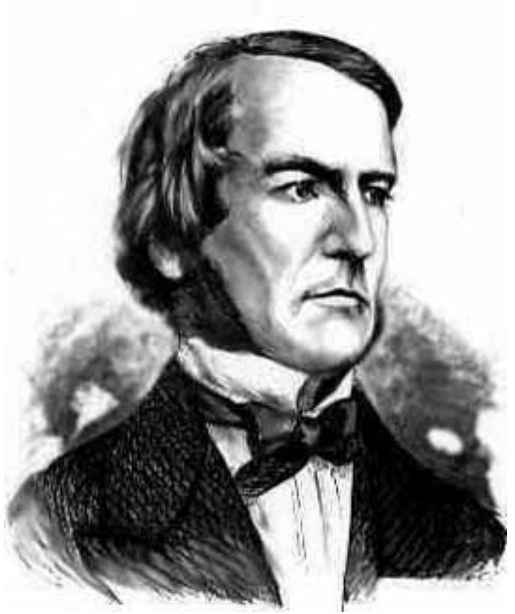
- Poseer un sistema numérico eficiente (lo tenemos).
- **Mecanismo para operar de forma eficiente sobre el sistema numérico.**
- Procedimiento para plasmar el mecanismo y el sistema numérico en elementos físicos.

De lo analógico a lo digital - Mecanismo

Lógica booleana

- Shannon publica un *paper* en 1938 que demuestra que la lógica booleana puede representarse con circuitos digitales.

- Veremos cómo realizar esta representación con las operaciones booleanas existentes.



George Boole

De lo analógico a lo digital - Mecanismo

Las fórmulas de lógica booleana poseen dos componentes importantes:

- Proposiciones lógicas o variables: valores de verdad (falso o verdadero; V o F). **Análogo a bits (0-1)**.
- Conectivos lógicos: Permiten unir variables y se definen usando tablas de verdad. **Análogo a operaciones (+, -, etc.)**.

De lo analógico a lo digital - Mecanismo

Tablas de verdad de los conectivos lógicos

A	*B*	*A* and *B*
V	V	V
V	F	F
F	V	F
F	F	F

Notación: $\wedge$

A	*B*	*A* or *B*
V	V	V
V	F	V
F	V	V
F	F	F

Notación: $\vee$

A	not(*A*)
V	F
F	V

Notación: $\neg$

De lo analógico a lo digital - Mecanismo

- Con `AND`, `OR` y `NOT` podemos representar cualquier tabla de verdad de conectivos binarios mediante la conexión de múltiples variables en conjunto con estos operadores.

- Ejemplo: `XOR` (u "`OR` exclusivo"). Notación: $\veebar$

- $A \veebar B = (\neg A \wedge B) \vee (A \wedge \neg B)$

A	*B*	*A* xor *B*
V	V	F
V	F	V
F	V	V
F	F	F

De lo analógico a lo digital - Mecanismo

De esta forma, además, podemos representar **operaciones aritméticas**. Veamos los valores resultantes de la suma de un bit.

Resultado de la suma

A	*B*	*S*
0	0	0
0	1	1
1	0	1
1	1	0

Acarreo de la suma

A	*B*	*C*
0	0	0
0	1	0
1	0	0
1	1	1

En resumen:

- $S = A \veebar B$
- $C = A \wedge B$

De lo analógico a lo digital - Mecanismo

Finalmente, a modo de recordatorio: **equivalencias lógicas**.

Algunas de estas nos permitirán facilitar la resolución de algunos problemas.

Equivalence	Name of Identity
$p \wedge T \equiv p$ $p \vee F \equiv p$	Identity Laws *
$p \wedge F \equiv F$ $p \vee T \equiv T$	Domination Laws *
$p \wedge p \equiv p$ $p \vee p \equiv p$	Idempotent Laws
$\neg(\neg p) \equiv p$	Double Negation Law
$p \wedge q \equiv q \wedge p$ $p \vee q \equiv q \vee p$	Commutative Laws
$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ $(p \vee q) \vee r \equiv p \vee (q \vee r)$	Associative Laws
$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	Ditributive Laws
$\neg(p \wedge q) \equiv \neg p \vee \neg q$ $\neg(p \vee q) \equiv \neg p \wedge \neg q$	De Morgan's Laws *
$p \wedge (p \vee q) \equiv p$ $p \vee (p \wedge q) \equiv p$	Absorption Laws
$p \wedge \neg p \equiv F$ $p \vee \neg p \equiv T$	Negation Laws

De lo analógico a lo digital

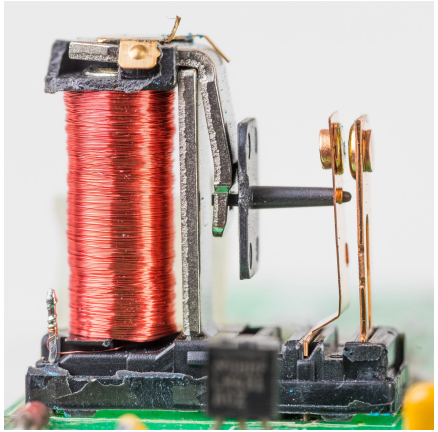
Para esta transformación, debemos seguir tres pasos:

- Poseer un sistema numérico eficiente (lo tenemos).
- Mecanismo para operar de forma eficiente sobre el sistema numérico (ahora lo tenemos).
- **Procedimiento para plasmar el mecanismo y el sistema numérico en elementos físicos.**

De lo analógico a lo digital - Implementación física

Entre los varios aportes de Claude Shannon, se encuentra la descripción de la implementación física de los conectivos `AND`, `OR` y `NOT`.

- Para cada conectivo define una compuerta lógica.
- Cada compuerta lógica se define mediante relés.



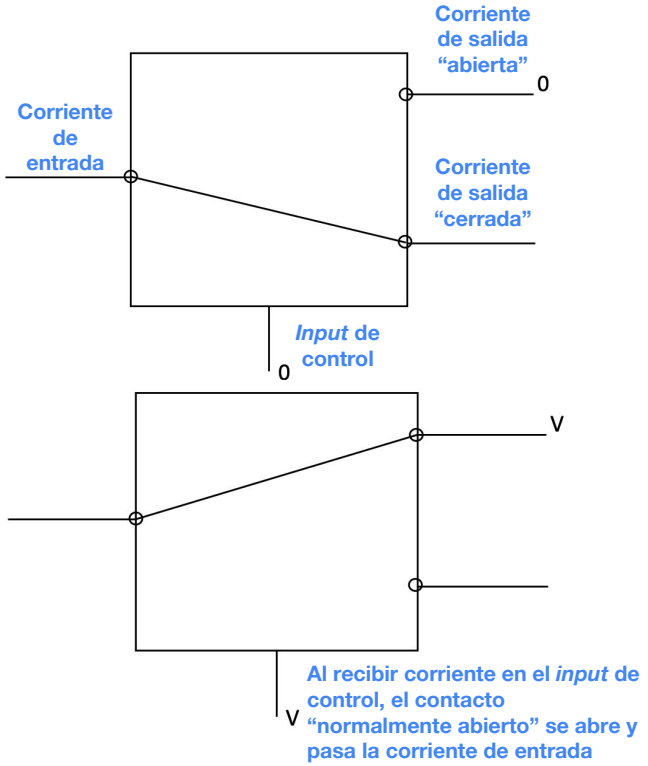


Relés/Relevadores

De lo analógico a lo digital - Implementación física

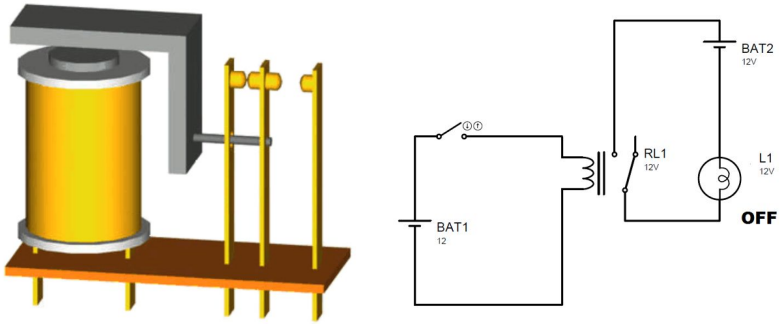
- Los relés poseen un cable de entrada, dos de salida (“normalmente abierto” y “normalmente cerrado”) y un *input* de control.

- El *input* de control determina la salida de la corriente de entrada (magnetismo de la bobina hace que cambien los contactos internos y, así, el camino que sigue la corriente).

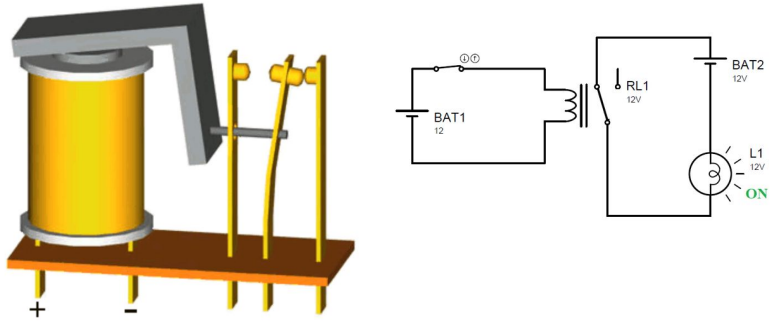


De lo analógico a lo digital - Implementación física

A modo de ilustración, supongamos que conectamos la corriente de salida “normalmente abierta” a un circuito conectado a una ampolleta.



Si no alimentamos al relé, el circuito de la ampolleta no se cierra y no logra ser alimentada por la corriente de la batería BAT2.

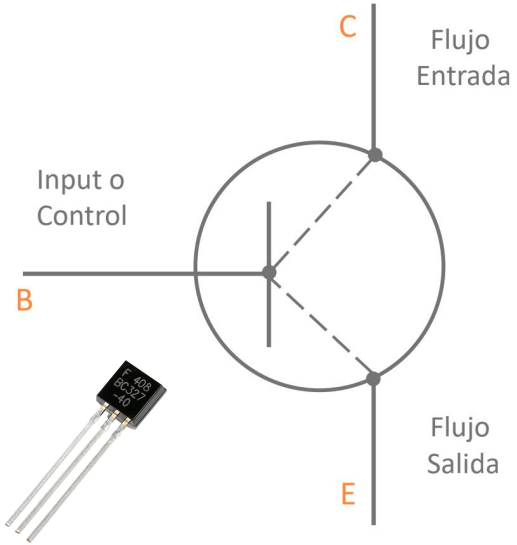


Si lo alimentamos, el campo electromagnético de la bobina hace que la salida “normalmente abierta” se cierre, permitiendo que la ampolleta se prenda con la carga de la batería.

De lo analógico a lo digital - Implementación física

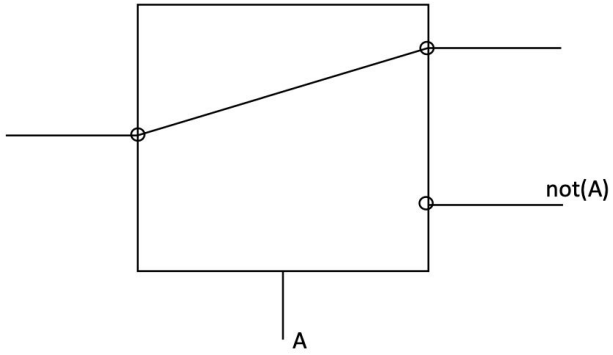
En la actualidad, son más utilizadas las compuertas lógicas creadas a partir de **transistores**.

- Poseen menores costos de mantención, mayor vida útil y rapidez de cambio de estado significativamente más alta.
- No obstante, veremos la construcción de compuertas mediante relés para seguir los pasos de Shannon. 👣

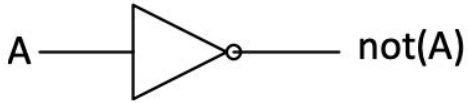


De lo analógico a lo digital - Implementación física

Construcción: Compuerta lógica NOT



La salida normalmente cerrada representará el valor negado de *A*: Si *A* = 1, entonces el contacto sube y la salida "normalmente cerrada" se quedará sin corriente (not(*A*)=0). Si *A* = 0, entonces el relé sigue cerrado y la corriente fluye por la salida normalmente cerrada (not(*A*)=1).

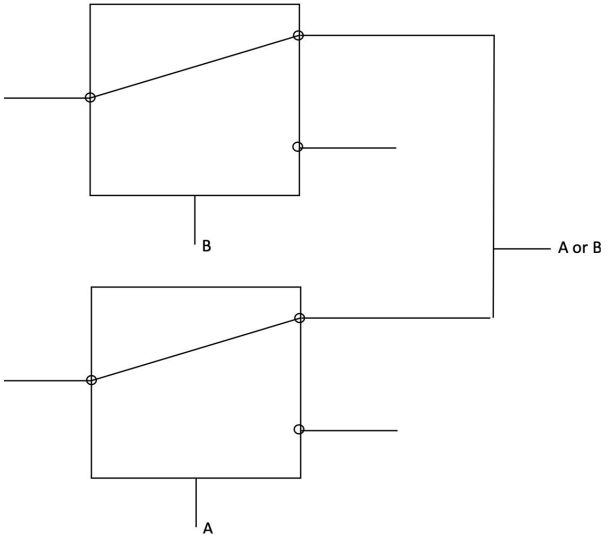


Notación digital a utilizar como abstracción.

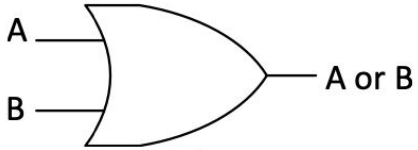
A	not(*A*)
1	0
0	1

De lo analógico a lo digital - Implementación física

Construcción: Compuerta lógica OR



La conexión de las salidas normalmente abiertas de dos relés en paralelo representarán el resultado del conectivo lógico OR: Si el *input* de control *A* o el *input* de control *B* poseen el valor 1, entonces uno de los contactos subirá y la corriente de entrada fluirá a la salida de nuestra compuerta (*A* or *B* = 1). Solo se quedará sin corriente si ambos *inputs* de control son iguales a 0 (*A* or *B* = 0).

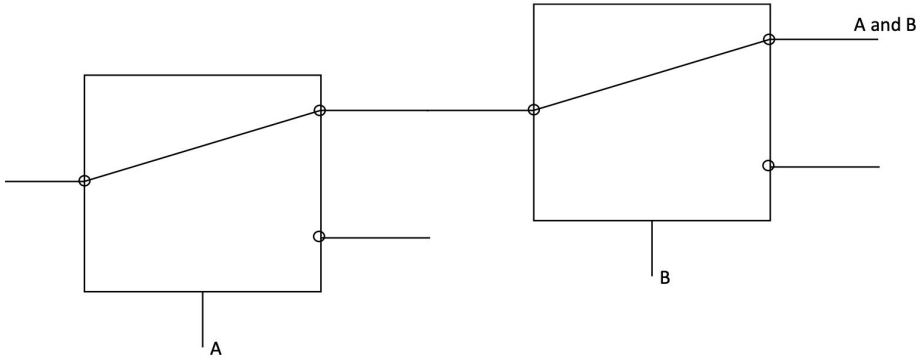


Notación digital a utilizar como abstracción.

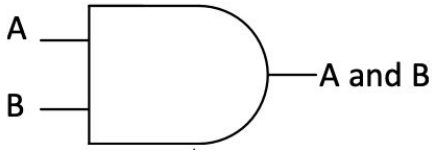
A	*B*	*A or B*
1	1	1
1	0	1
0	1	1
0	0	0

De lo analógico a lo digital - Implementación física

Construcción: Compuerta lógica AND



La conexión de las salidas normalmente abiertas de dos relés en secuencia representarán el resultado del conectivo lógico `AND`: Si el *input* de control *A* y el *input* de control *B* poseen el valor 1, entonces ambos contactos subirán y la corriente de entrada fluirá a la salida de nuestra compuerta (*A* and *B* = 1). Solo se quedará sin corriente si alguno de los *inputs* de control son iguales a 0 (*A* and *B* = 0).



Notación digital a utilizar como abstracción.

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

De lo analógico a lo digital - Implementación física

Ejercicios

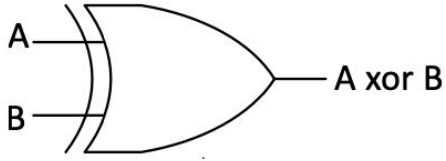
Usando relés, crea las siguientes compuertas lógicas:

- XOR.
- NAND = NOT AND.
- NOR = NOT OR.
- XNOR = NOT XOR.

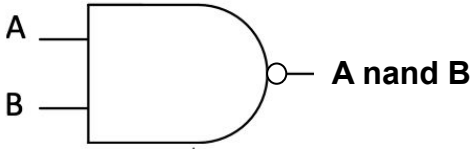
De lo analógico a lo digital - Implementación física

Compuertas lógicas XOR, NAND, NOR, XNOR

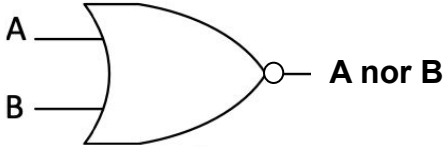
Notación digital a utilizar como abstracción.



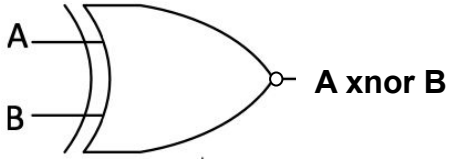
A	*B*	*A* xor *B*
1	1	0
1	0	1
0	1	1
0	0	0



A	*B*	*A* nand *B*
1	1	0
1	0	1
0	1	1
0	0	1



A	*B*	*A* nor *B*
1	1	0
1	0	0
0	1	0
0	0	1

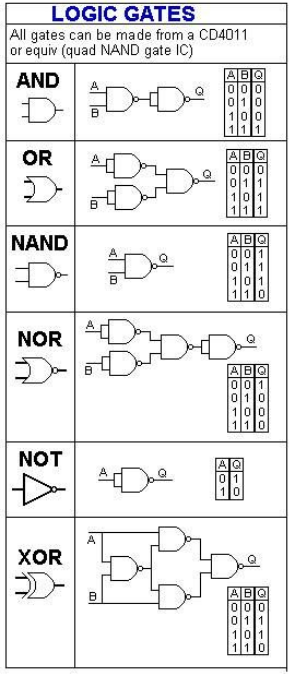


A	*B*	*A* xnor *B*
1	1	1
1	0	0
0	1	0
0	0	1

De lo analógico a lo digital - Implementación física

¿Y de qué nos sirven las compuertas negadas?

- Puntualmente, las compuertas NAND y NOR se conocen como **compuertas globales**. Esto implica que solo usando NAND o solo usando NOR podemos definir el resto de compuertas lógicas.



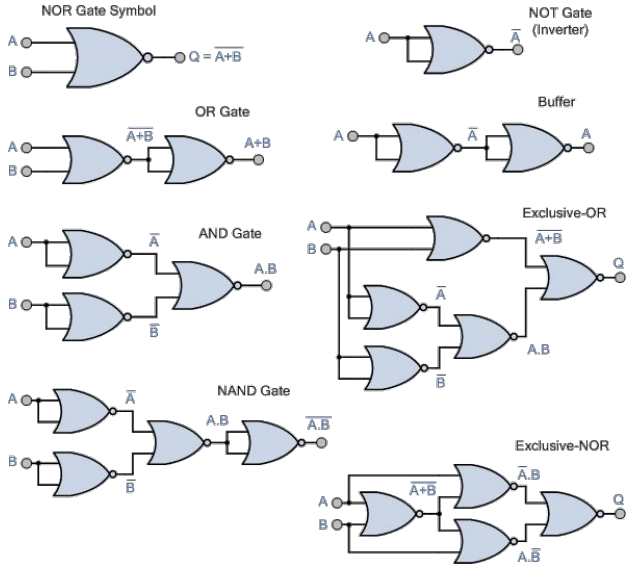
Equivalencias de NAND.

De lo analógico a lo digital - Implementación física

¿Y de qué nos sirven las compuertas negadas?

- El uso de compuertas `NAND` y `NOR` en los circuitos digitales reduce tanto espacio como tiempos de propagación de carga y, además, son más económicos.

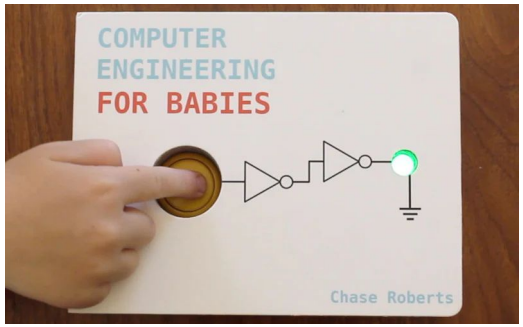
- Veremos su uso para **almacenamiento de datos** en la próxima clase.



Equivalencias de NOR.

De lo analógico a lo digital - Libro resumen

- **Recomendación:** *Computer Engineering for Babies*.
- Resume todas las compuertas lógicas, incluyendo **latches** (próxima clase).
- Aprobado por mi sobrina. 💙



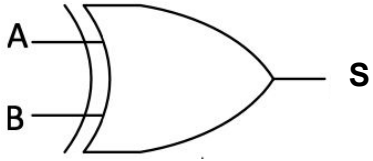


Operaciones aritméticas digitales

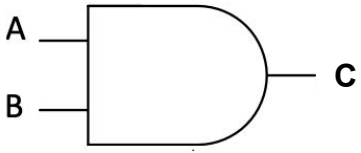
Ya logramos definir una implementación física y tenemos todas nuestras compuertas definidas… ahora, ¿cómo operamos?

Partiremos por la suma: ya conocemos las fórmulas lógicas que nos permiten obtener la suma y el acarreo entre dos bits. Las traduciremos directamente a circuitos digitales.

- $S = A \veebar B$



- $C = A \wedge B$

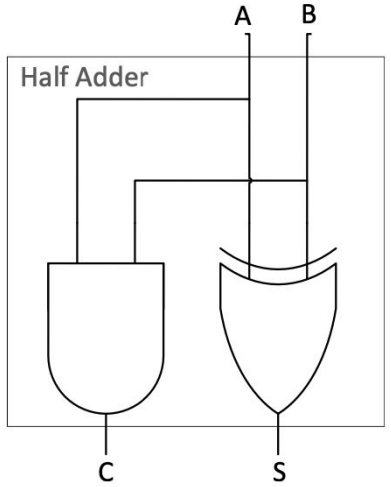


Operaciones aritméticas digitales

Si unimos las compuertas anteriores en un único componente, tenemos lo que se llama un ***Half-adder*** o medio sumador de un bit.

- ¿Por qué "medio" sumador?: Porque **no incluye el bit de acarreo de la suma anterior**. Solo sirve para la suma de los bits menos significativos entre dos números.

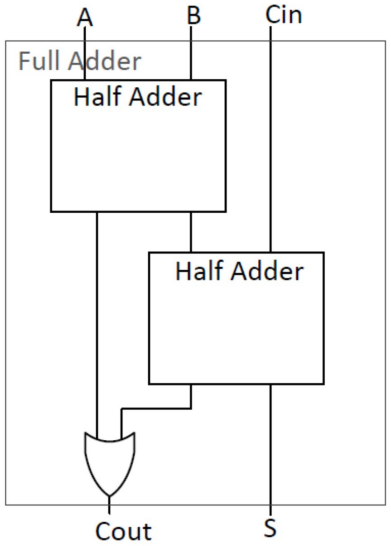
- Solución: **unir dos *half-adders***.



Operaciones aritméticas digitales

La unión de dos *half adders* se conoce como ***Full-adder*** o sumador completo de un bit.

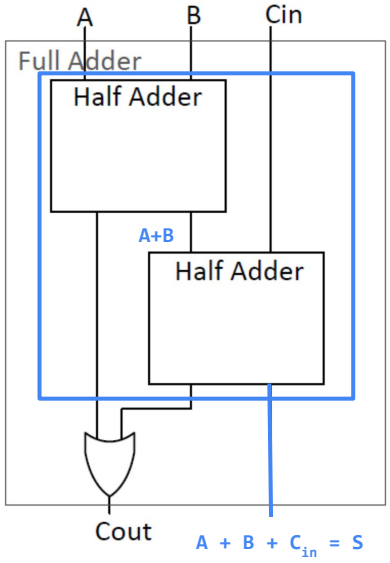
- La señal C_{in} corresponde al acarreo de entrada (***carry in***). Si A y B corresponden a los bits menos significativos en una suma, entonces $C_{in} = 0$.
- C_{out} corresponderá al acarreo de salida (***carry out***). Este vendrá a ser el *carry in* de la siguiente suma.



Operaciones aritméticas digitales

¿Cómo funciona en términos prácticos?

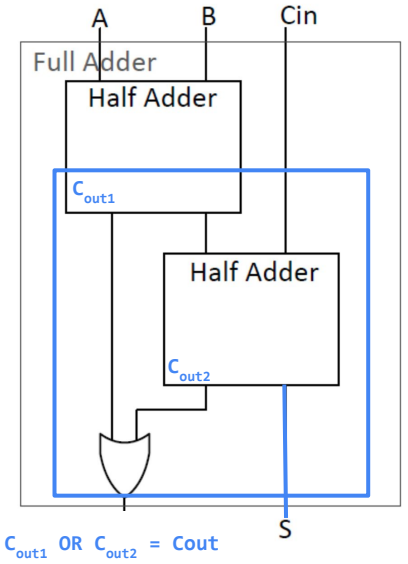
- En primer lugar, queremos obtener la suma entre *A*, *B* y el *carry* de entrada.
- Para llevar a cabo esto, primero obtenemos la suma de *A* con *B* y, posteriormente, sumamos este resultado con el *carry* de entrada con otro *half adder*.



Operaciones aritméticas digitales

¿Cómo funciona en términos prácticos?

- Ahora, queremos obtener el *carry* de salida.
- El *carry* de salida **solo** se obtiene de uno de los dos *half adders*. ¿Por qué?
- Si $C_{out1} = 1$, esto ocurre si y solo si $A = 1$, $B = 1$ y, por ende, $A + B = 0$.
  El segundo *half adder* sumará $C_{in} + 0$, por lo que independiente del valor de C_{in}, $C_{out2} = 0$.



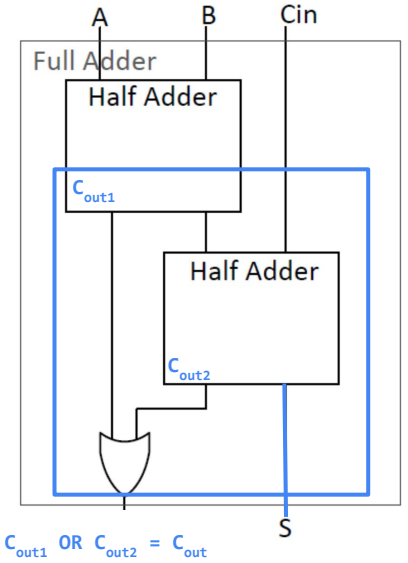
Operaciones aritméticas digitales

- Si $C_{out1} = 0$, entonces tenemos tres casos:
  $A = 0, B = 0$; $A = 1, B = 0$; $A = 0; B = 1$.

- Nos interesan los casos donde $A + B = 1$.
  En este caso, si $C_{in} = 1$, entonces tendremos que $C_{out2} = 1$.

- Como queremos determinar si hubo o no *carry* en cualquiera de los dos casos, se incorporan ambos a través de un OR.
  Podría ser XOR también, ¿por qué?

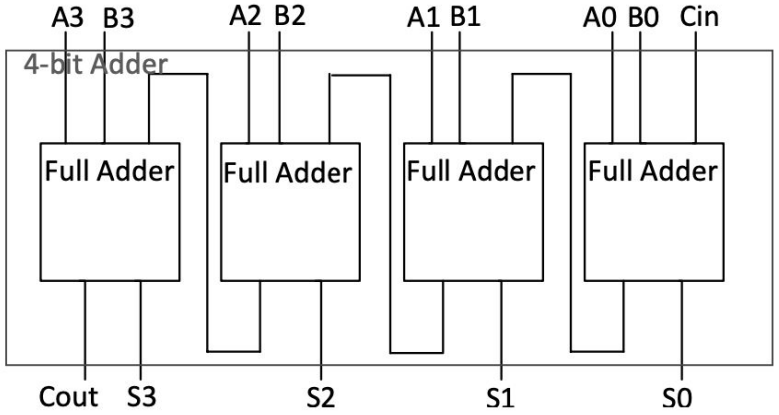
R: Porque solo uno de los dos será igual a 1, lo que no cambia el *output*.



Operaciones aritméticas digitales

Veamos entonces cómo llevar a cabo la suma entre dos números *A* y *B* de 4 bits. En este caso, el número indica la posición del bit en la secuencia (0 = menos significativo o extremo derecho, 3 = más significativo o extremo izquierdo).

- Aunque no se use directamente, C_{out} es necesario para poder conectar la componente con el C_{in} de otro sumador si es que corresponde.



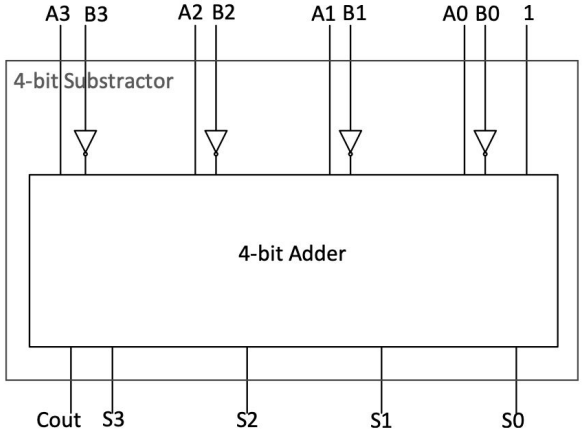
Operaciones aritméticas digitales

Con la suma lista, ¿cómo implementamos la resta?

- **Suma del complemento de 2**.

- Nos abstraemos del sumador de 4 bits (4-bit *adder*) y cambiamos el *input* de *B* para entregar su complemento.

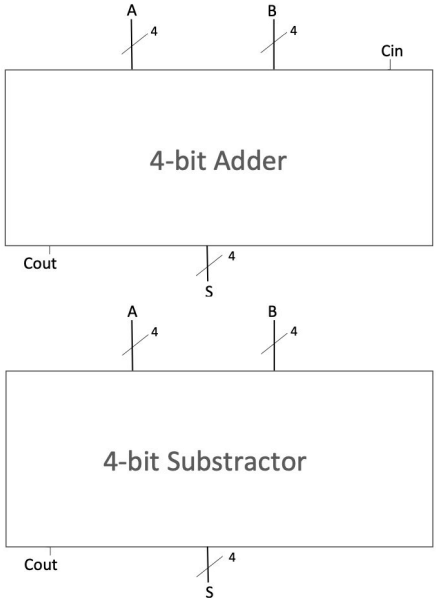
- La unidad adicional del complemento de 2 se incorpora convenientemente con el acarreo de entrada ($C_{in} = 1$).



Abstracción de componentes

Para facilitar la eventual construcción de nuestro computador básico, es necesario que podamos modularizar y abstraernos de las componentes que vamos diseñando.

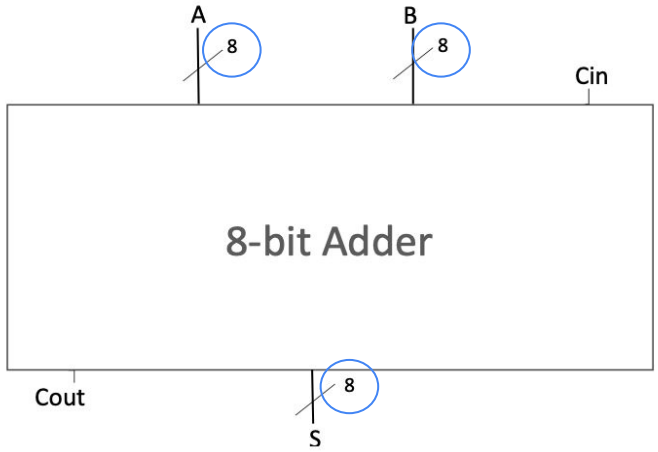
Por ejemplo, con el sumador y restador de 4 bits definidos, nos basta con conocer el nombre de los componentes y sus señales de entrada y salida para entender su funcionalidad dentro de un diagrama.



Abstracción de componentes

La **notación de buses** también es importante para facilitar la abstracción de nuestros componentes.

Por ejemplo, si queremos extender a 8 bits nuestro sumador, simplemente modificamos el número de los buses de entrada y salida del componente.



Resolución de circuitos - *Minterms* y *Maxterms*

Supongamos que queremos elaborar un circuito para la siguiente tabla de verdad.

¿Cómo lo podemos hacer sin fuerza bruta?

A partir de ***minterms*** y ***maxterms***.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Resolución de circuitos - *Minterms* y *Maxterms*

***Minterm*:** Combinación de *inputs* que entregan un *output* igual a 1 en una expresión booleana, es decir, que hacen que se evalúe como verdadera.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

En el ejemplo, estos son:

- $A = 0, B = 1, C = 0$
- $A = 1, B = 0, C = 1$
- $A = 1, B = 1, C = 1$

¿Cómo los combinamos?

Resolución de circuitos - *Minterms* y *Maxterms*

Minterm - Construcción

- Las señales dentro de una combinación de *input* **se conectan con compuertas `AND`** porque se debe verificar que sus valores son los esperados. A las señales de valor 0 **se les aplica `NOT` para verificar que tengan un valor de verdad falso**.

- **Todas las combinaciones se conectan con compuertas `OR`**, ya que basta con que una de ellas se cumpla para que el *output* sea igual a 1.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Resolución de circuitos - *Minterms* y *Maxterms*

Minterm - Construcción

Por lo tanto, la expresión booleana para la tabla con *minterms* es:

(NOT(*A*) AND *B* AND NOT(*C*))
OR (*A* AND NOT(*B*) AND *C*)
OR (*A* AND *B* AND *C*)

Lo que se puede traducir directamente a un circuito.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Resolución de circuitos - *Minterms* y *Maxterms*

Maxterm: Combinación de *inputs* que entregan un *output* igual a 0 en una expresión booleana, es decir, que hacen que se evalúe como falsa.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

En el ejemplo, estos son:

- $A = 0, B = 0, C = 0$
- $A = 0, B = 0, C = 1$
- $A = 0, B = 1, C = 1$
- $A = 1, B = 0, C = 0$
- $A = 1, B = 1, C = 0$

¿Cómo los combinamos?

Resolución de circuitos - *Minterms* y *Maxterms*

Maxterm - Construcción

- Las señales dentro de una combinación de *input* **se conectan con compuertas `OR` utilizando sus negaciones**. Por ejemplo, si un *input* es 1, se le aplica `NOT`; en otro caso se deja como está. Esto asegura que para una combinación de señales con *output* 0 la expresión **sea falsa**.

- **Todas las combinaciones se conectan con compuertas `AND`**, ya que basta con que una de ellas no se cumpla para que el *output* sea igual a 0.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Resolución de circuitos - *Minterms* y *Maxterms*

Maxterm - Construcción con *minterms*

- Se puede obtener la fórmula lógica de la misma forma que con los *minterms*.

- Una vez obtenida la fórmula, simplemente **se aplica un `NOT` sobre ella y se aplica la "Ley De Morgan" para obtener el resultado final**.

Esta forma es más directa, pero en ambos casos se llega al mismo resultado.

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Resolución de circuitos - *Minterms* y *Maxterms*

Maxterm

Por lo tanto, la expresión booleana para la tabla con *maxterms* es:

$$
\begin{aligned}
& (A \text{ OR } B \text{ OR } C) \\
\text{AND } & (A \text{ OR } B \text{ OR } \text{NOT}(C)) \\
\text{AND } & (A \text{ OR } \text{NOT}(B) \text{ OR } \text{NOT}(C)) \\
\text{AND } & (\text{NOT}(A) \text{ OR } B \text{ OR } C) \\
\text{AND } & (\text{NOT}(A) \text{ OR } \text{NOT}(B) \text{ OR } C)
\end{aligned}
$$

A	*B*	*C*	*Output*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Lo que se puede traducir directamente a un circuito.

Resolución de circuitos - *Minterms* y *Maxterms*

Ambos circuitos son equivalentes: representan la misma tabla de verdad. Se preferirán los *minterms* si la mayoría de las filas posee un valor de verdad igual a 0 ya que la fórmula final tendrá menos expresiones; en otro caso, se preferirán los *maxterms*.

Por otra parte, se puede utilizar la siguiente notación matemática:

- NOT: $\overline{x}$
- AND: $\cdot$
- OR: $+$

(NOT(A) AND B AND NOT(C)) OR (A AND NOT(B) AND C) OR (A AND B AND C)

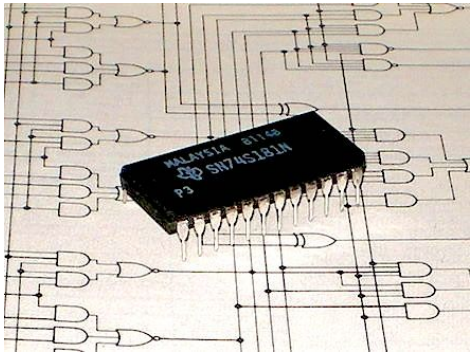
$\equiv$

$(\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) + (A \cdot B \cdot C)$ * Mucho más amigable.

Unidad aritmético-lógica

Ahora que sabemos operar con componentes digitales, hablemos de la ***Arithmetic Logic Unit*** **(ALU)**.

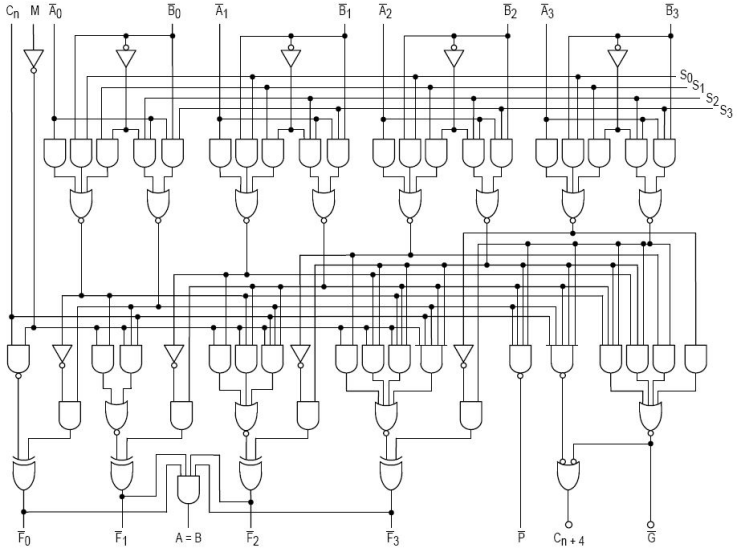
Corresponde a la unidad aritmético-lógica del computador, es decir, la **componente principal** para realizar **operaciones numéricas**.



Componente 74181, primera ALU completa de 4 bits incorporada dentro de un único chip.

Unidad aritmético-lógica

¿Cómo es el diagrama del circuito del componente 74181?

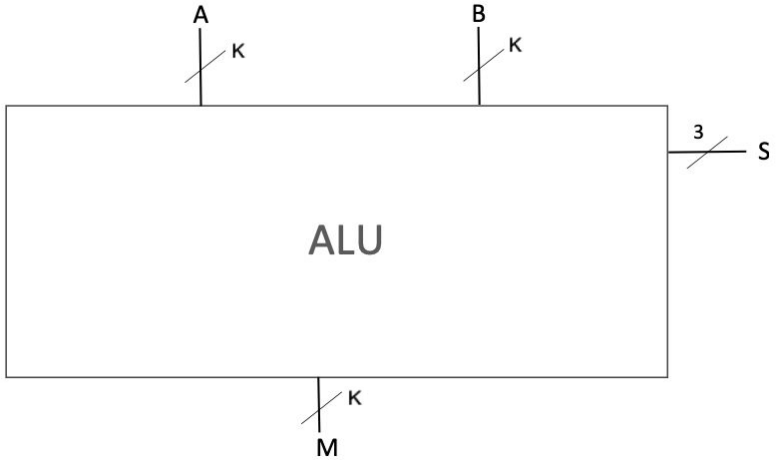




Por esto optamos por la abstracción.

Unidad aritmético-lógica

En este caso, trabajaremos con una versión simplificada de la ALU.



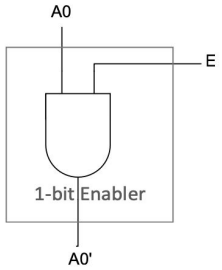
S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right

A continuación, veremos las componentes y operaciones que nos faltan para construir esta ALU.

Unidad aritmético-lógica - Circuitos de control

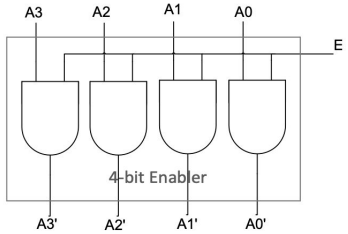
La ALU es capaz de entregar resultados distintos según la operación escogida (referirse a la tabla anterior). Para ello, necesitamos componentes que nos permitan controlar los buses de salida.

- ***Enabler* o habilitador**
  Componente que habilita o no el paso de una señal o bus de datos.



E	A_0'
0	0
1	A_0

Con el uso de la compuerta AND el resultado depende netamente de la señal *E*.

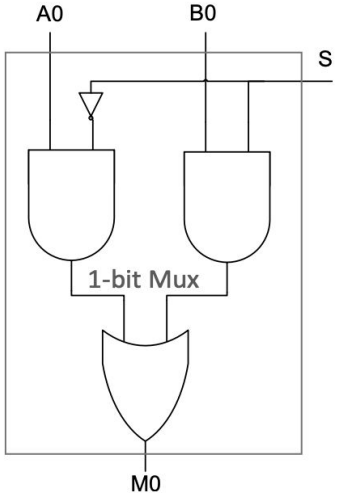


Se extiende de manera sencilla el componente, manteniendo una única señal *E*.

Unidad aritmético-lógica - Circuitos de control

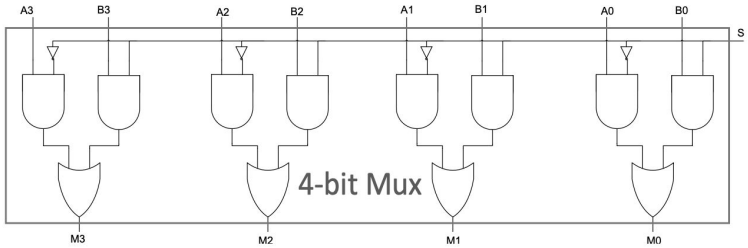
- **Multiplexor o Mux**

  Permite el paso de una señal o bus de datos entre un conjunto de opciones. Se construye como una extensión del *enabler*.



S	M_0
0	A_0
1	B_0

En la parte superior usamos *enablers* (uno con su señal de paso negada) para seleccionar la entrada deseada. El componente `OR` se encarga de transmitir la señal escogida.

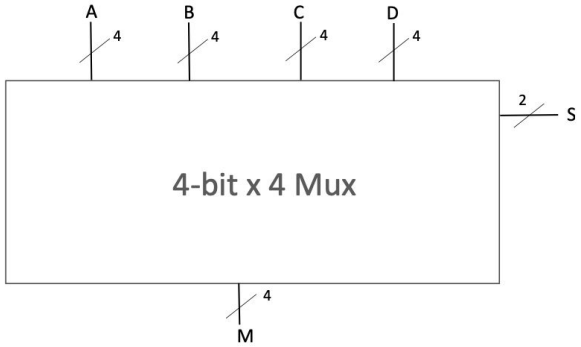


La extensión de este componente también es trivial: simplemente se replica para cada bit de los buses de entrada.

Unidad aritmético-lógica - Circuitos de control

- **Multiplexor o Mux**

  También podemos extenderlo a más buses de entrada, lo que hace que aumente la cantidad de bits necesarios de selección.



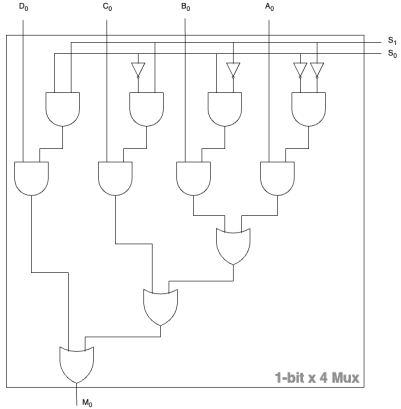
S_1	S_0	M
0	0	A
0	1	B
1	0	C
1	1	D

¿Cómo se construye? Lo veremos a continuación.

Unidad aritmético-lógica - Circuitos de control

Multiplexor o Mux

Primero se define el componente 1-bit x 4 Mux, que selecciona una de cuatro señales de entrada de 1 bit según la tabla. El valor de S_1S_0 asociado a una señal es arbitrario, podría ser modificado.



S_1	S_0	M_0
0	0	A_0
0	1	B_0
1	0	C_0
1	1	D_0

Cabe destacar que la cantidad de bits de la señal de selección *S* define cuántas señales de entrada se pueden escoger:

- #Bits(*S*) = 1 → # Entradas = 2^1 = 2, ya que tenemos 2 valores posibles con 1 bit (0-1).
- #Bits(*S*) = 2 → # Entradas = 2^2 = 4, ya que tenemos 4 valores posibles con 2 bits (00-01-10-11).
- #Bits(*S*) = 3 → # Entradas = 2^3 = 8, ya que tenemos 8 valores posibles con 3 bits (000-001-010-011-100-101-110-111).

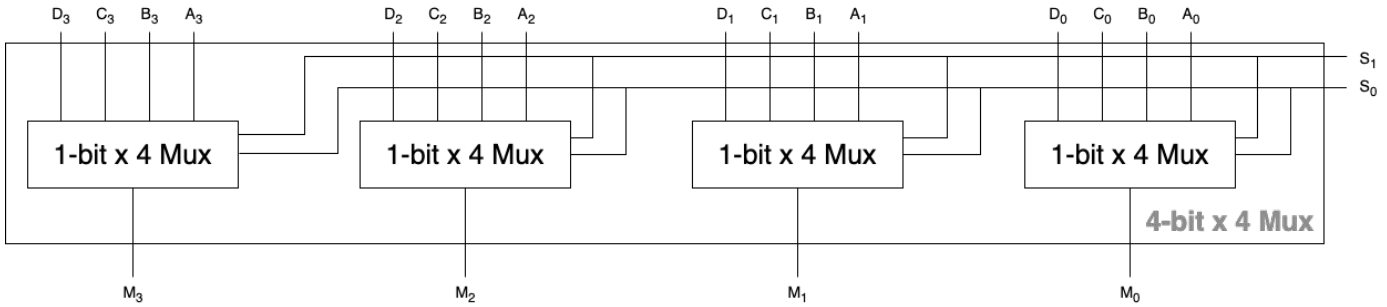
También vale la pena mencionar que si tenemos menos entradas seleccionables que combinaciones de *S* (Ej: 3 entradas distintas para una señal *S* de 2 bits), se puede asociar una de las entradas a más de una combinación de *S*; o bien usar una señal constante (Ej: 0).

* Si te cuesta entender por qué se construye así, fíjate que para cada combinación *S* solo una de las cuatro compuertas AND de la primera fila retorna 1, lo que habilita el paso de una sola señal de entrada y bloquea al resto, donde se propagan 0's que se ignoran por las compuertas OR.

Unidad aritmético-lógica - Circuitos de control

- **Multiplexor o Mux**

  Finalmente, usamos la abstracción del componente anterior para definir 4-bit x 4 Mux. Como se observa en la figura, solo aplicamos la operación del circuito anterior sobre cada uno de los cuatro bits de las señales de entrada *A*, *B*, *C* y *D*.



S_1	S_0	*M*
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

Unidad aritmético-lógica - Circuitos de control

- **Multiplexor o Mux**
  En resumen:

  - Si queremos aumentar la cantidad de bits de las señales de entrada de un multiplexor, le añadimos a su interior multiplexores de 1 bit.

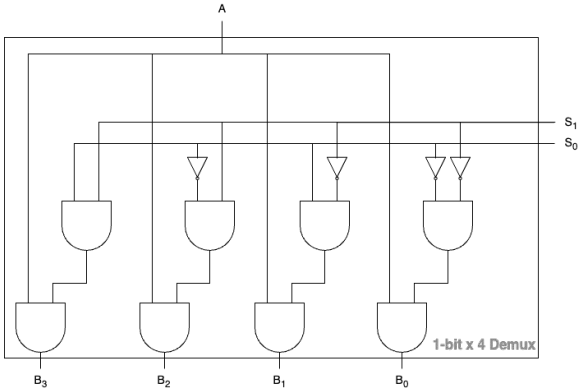
  - Si queremos aumentar la cantidad de señales de entrada, aumentamos la cantidad de bits de la señal de selección *S*.



Unidad aritmético-lógica - Circuitos de control

- **De-Multiplexor o Demux**

  Transmite una señal o bus de datos a una de múltiples salidas.



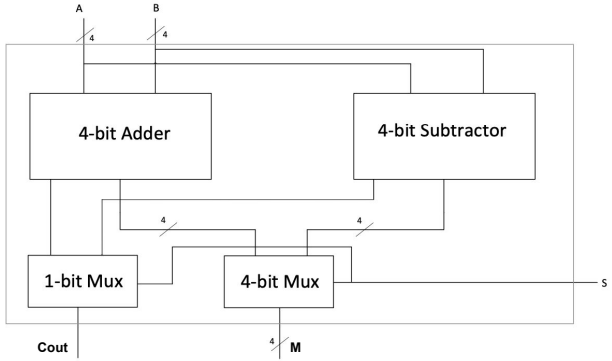
S1	S0	Output
0	0	$B_0 = A$
1	0	$B_1 = A$
0	1	$B_2 = A$
1	1	$B_3 = A$

También corresponde a una extensión de los *enablers*. Mientras una de las salidas poseerá el valor de *A*, el resto tendrá valor por defecto 0. Su extensión a *N*-bits de entrada es trivial, ya que se repite el componente para cada bit. La complejidad aumenta si aumenta la cantidad de salidas posibles, ya que aumenta el tamaño del bus de selección *S* (y así, las combinaciones de compuertas `AND` a utilizar).

No lo usaremos de inmediato, pero nos servirá en la próxima clase.

Unidad aritmético-lógica - Sumador/Restador

Con nuestras componentes de control definidas, ahora podemos unir nuestro sumador y restador en una única componente.



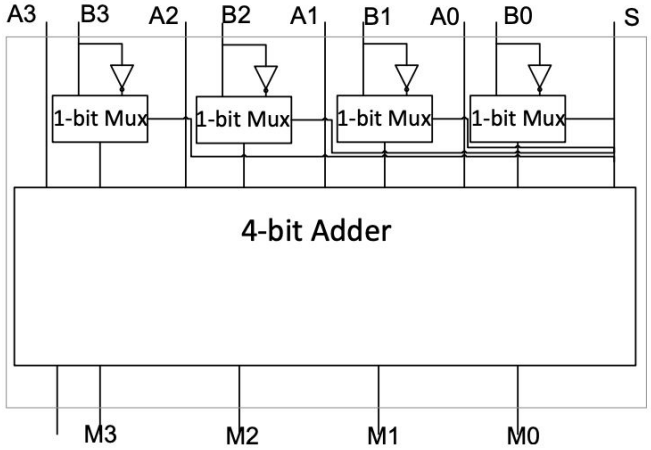
S	*M*	C_{out}
0	*A+B*	*A+B* C_{out}
1	*A-B*	*A-B* C_{out}

¿Es esto lo óptimo? **No**, podemos mejorar. Vamos.



Unidad aritmético-lógica - Sumador/Restador

Ahora definimos una única componente donde operan directamente *A* y *B*, con la salvedad de que el *input* de *B* depende de la operación seleccionada.

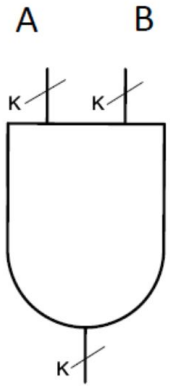


S	*M*	C_{out}
0	*A+B*	*A+B* C_{out}
1	*A-B*	*A-B* C_{out}

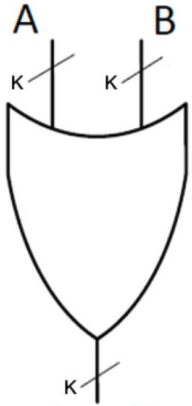
Se respetan las salidas anteriores y utilizamos menos componentes. Cabe destacar, además, que se usa el bit de selección como la señal C_{in} (es decir, se suma una unidad directamente si se escoge la operación de resta, respetando el complemento de 2).

Unidad aritmético-lógica - Operadores lógicos *bitwise*

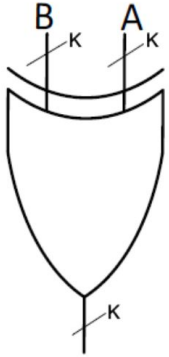
Para los operadores lógicos, nos abstraemos a través de los buses de entrada y salida, ya que en estos casos se observa que se aplican los componentes para cada bit de entrada (***bitwise*** - bit a bit).



A and B



A or B



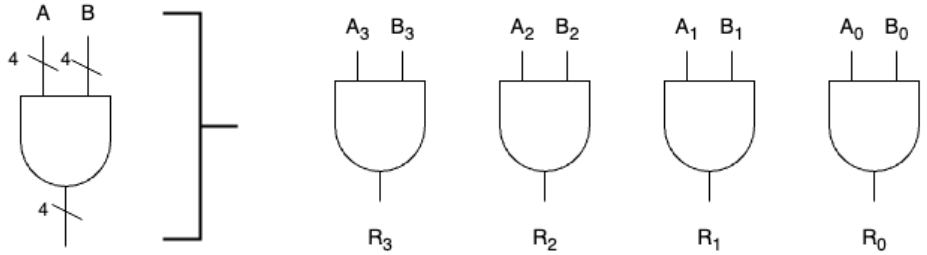
A xor B



not(A)

Unidad aritmético-lógica - Operadores lógicos *bitwise*

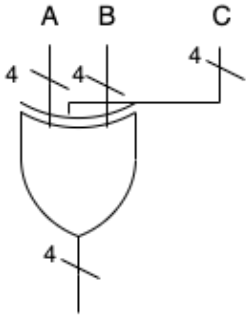
Ejemplo para *A* `AND` *B* con $k = 4$.



$R = (A_3 \text{ AND } B_3)(A_2 \text{ AND } B_2)(A_1 \text{ AND } B_1)(A_0 \text{ AND } B_0) = R_3R_2R_1R_0$

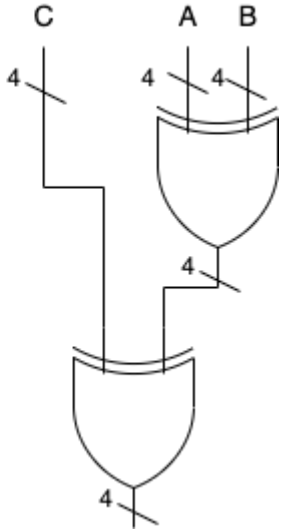
Unidad aritmético-lógica - Operadores lógicos *bitwise*

¡No es lo mismo que aumentar el número de buses de entrada! Por ejemplo, asumamos que queremos realizar la operación *A* XOR *B* XOR *C*.





¡FORMA INCORRECTA!



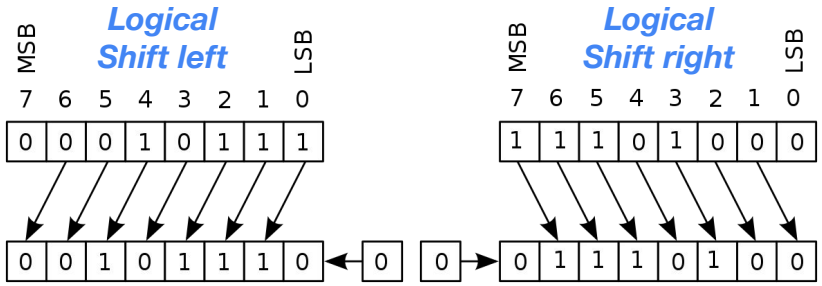


FORMA CORRECTA

Unidad aritmético-lógica - *Logical shifting*

Las operaciones de ***shifting*** corresponden al desplazamiento de los bits de un número, ya sea hacia la izquierda o hacia la derecha.

- shift_left(0100b) = 1000b
- shift_left(1001b) = 0010b
- shift_right(0100b) = 0010b
- shift_right(1001b) = 0100b

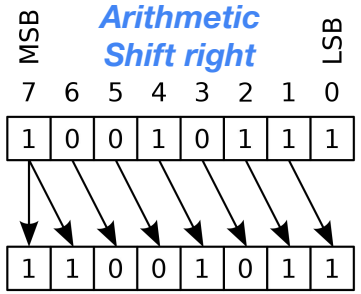


Estas operaciones son análogas a la multiplicación por 2 (*shift left*) y división **entera** por 2 (*shift right*). Cabe destacar que, dependiendo de la operación, se pierde el bit más o menos significativo.

Unidad aritmético-lógica - *Arithmetic Shift Right*

Paréntesis: Existen algunas implementaciones de *shift right* en las que el bit más significativo no solo se traslada hacia la derecha, sino que además **se copia** en el bit más significativo de la salida.

- shift_arithmetic_right(1100b) = 1110b
- shift_arithmetic_right(1001b) = 1100b



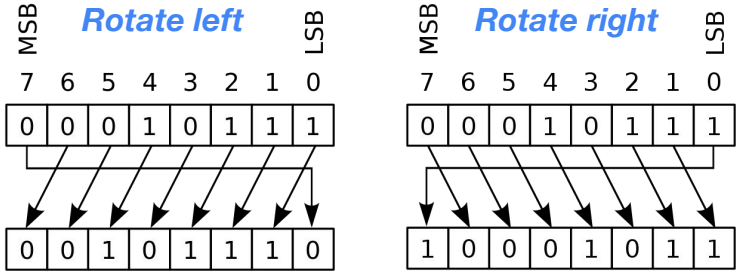
Este se conoce como ***arithmetic shift right***:
Mantiene el signo del número operado y funciona como división entera por 2 (aproximando siempre al entero **menor**). En nuestra ALU, no obstante, **utilizaremos el *logical shift right***.

Unidad aritmético-lógica - *Rotating*

Las operaciones de ***rotating*** son equivalentes al *shifting*, solo que el bit que se pierde ahora "rota" a la posición más o menos significativa, según la dirección de desplazamiento.

* Este solo tiene utilidad en el sentido de que evita que se pierdan bits. En términos matemáticos, no tiene usos prácticos.

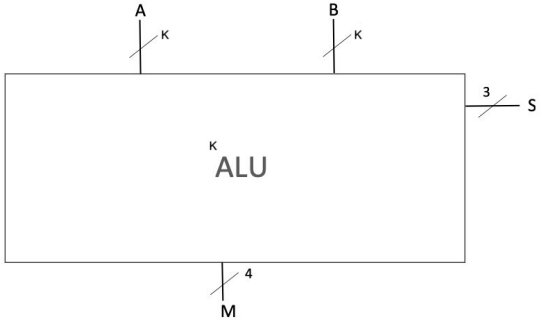
- rotate_left(1000b) = 0001b
- rotate_left(0101b) = 1010b
- rotate_right(0100b) = 0010b
- rotate_right(0011b) = 1001b



Ejercicio: Elabore diagramas para las componentes de *shifting* y *rotating* con buses de entrada de 4 bits.

Unidad aritmético-lógica

Ahora poseemos todas las componentes necesarias para nuestra ALU (para efectos de este curso, $k = 8$).



S2	S1	S0	M
0	0	0	Suma
0	0	1	Resta
0	1	0	And
0	1	1	Or
1	0	0	Not
1	0	1	Xor
1	1	0	Shift left
1	1	1	Shift right



Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.



Ejercicios

Implemente, utilizando solo las compuertas lógicas `AND`, `OR` y `NOT`, el conectivo binario condicional ($\rightarrow$), que está definido por la siguiente tabla de verdad:

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

***Interrogación* 1, 2014-2**

Ejercicios

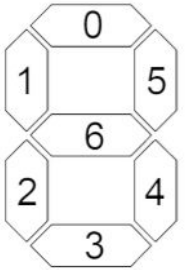
Implemente un circuito 2 bit *Multiplier* que realice la multiplicación entre dos valores de 2 bits.

Apuntes del curso

Ejercicios

Construya con compuertas lógicas el controlador de un *display* de 7 segmentos. Este controlador recibe como *input* solo un bit, *i.e.*, el *display* de 7 segmentos solo mostrará 0 o 1.

***Interrogación* 1, 2011-2**



Ejercicios

¿Qué número entero es generado al realizar cuatro operaciones *shift right* seguidas de cinco operaciones *rotate left* a un registro de 8 bits que inicialmente almacena el número entero 79?

***Interrogación* 1, 2013-1**

Ejercicios

Un comparador de números es un circuito que, dados dos números *A* y *B* en representación posicional, indica cuál es mayor o si estos son iguales. El circuito posee tres salidas, donde la primera entrega un 1 solo si *A* es mayor, la segunda un 1 solo si ambas son iguales, y la tercera un 1 solo si *B* es mayor.

Diseñe un comparador de números naturales de *N* bits.

***Tarea* 2, 2018-2**

Ejercicios

Extienda el circuito anterior para realizar la comparación entre el valor absoluto de dos números *A* y *B* de *N* bits.

***Tarea* 2, 2018-2**

Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?





DCC

DEPARTAMENTO DE CIENCIA
DE LA COMPUTACIÓN

IIC2343

Arquitectura de Computadores

Clase 2 - Operaciones Aritméticas y Lógicas

Profesor: Germán Leandro Contreras Sagredo

Anexo - Videos complementarios

Por si desean profundizar más en los temas de esta clase.



- Introducción a la teoría de la información de Claude Shannon.
- Demostración y explicación de un relé.
- Construcción de circuito OR y NOR con relés.
- ¿Qué es un transistor?
- Construcción de circuitos según tablas de verdad con *minterms*.

Anexo - Resolución de ejercicios

¡Importante!

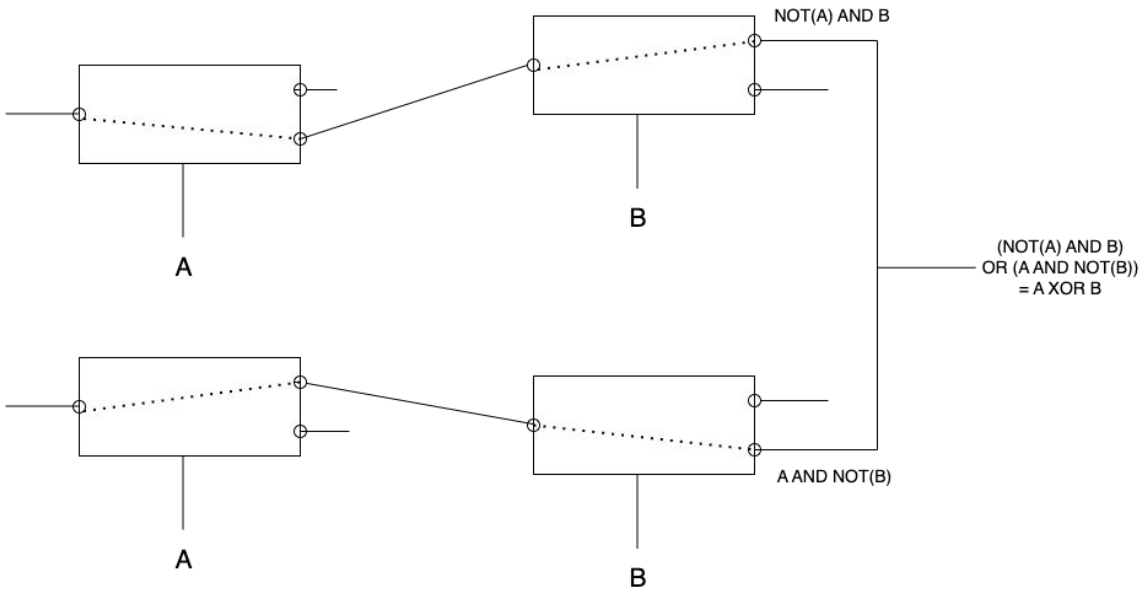
Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.



Ejercicios - Respuesta

Usando relés, crea las siguientes compuertas lógicas:

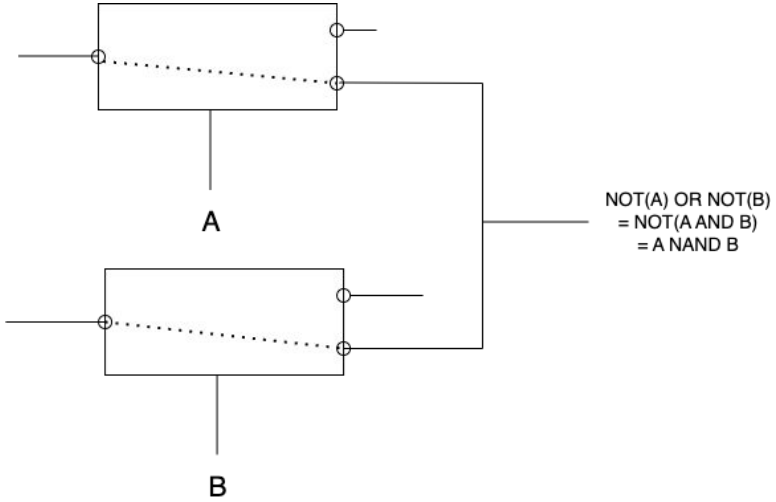
- XOR.



Ejercicios - Respuesta

Usando relés, crea las siguientes compuertas lógicas:

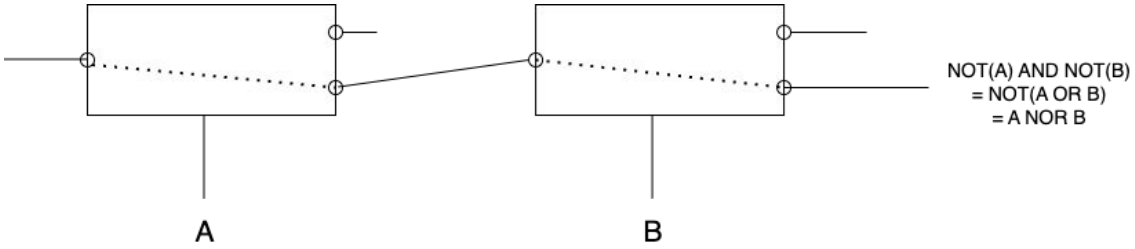
- NAND = NOT AND.



Ejercicios - Respuesta

Usando relés, crea las siguientes compuertas lógicas:

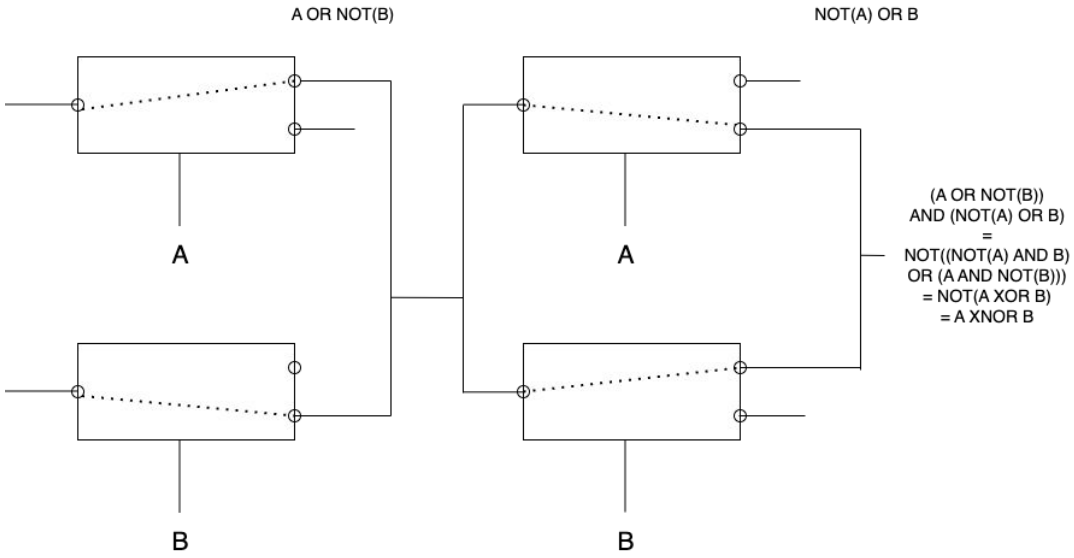
- NOR = NOT OR.



Ejercicios - Respuesta

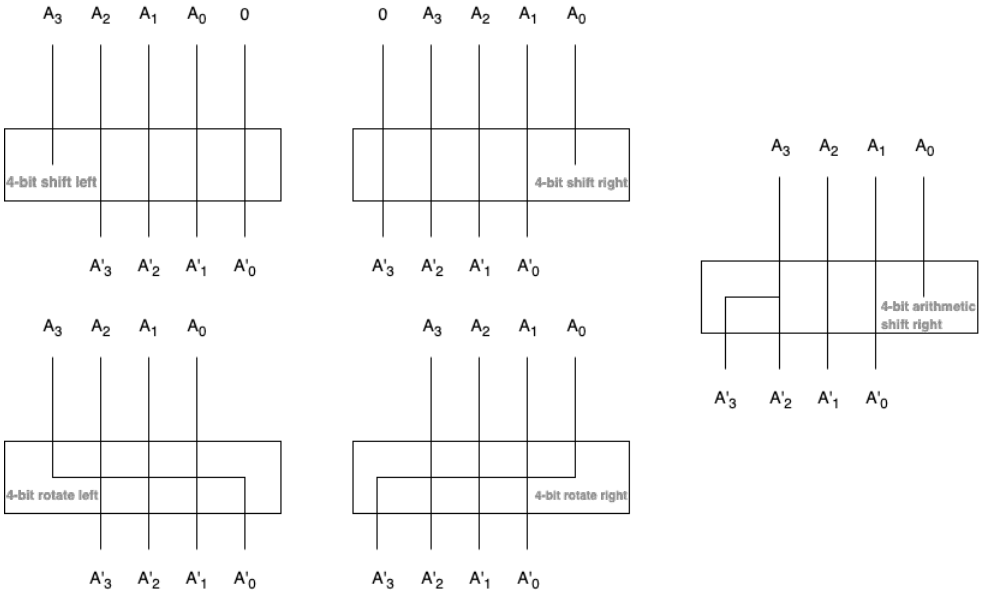
Usando relés, crea las siguientes compuertas lógicas:

- XNOR = NOT XOR.



Ejercicios - Respuesta

Elabore diagramas para las componentes de *shifting* y *rotating* con buses de entrada de 4 bits.



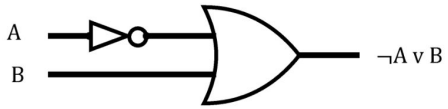
Ejercicios - Respuesta

Implemente, utilizando solo las compuertas lógicas `AND`, `OR` y `NOT`, el conectivo binario condicional (→), que está definido por la siguiente tabla de verdad:

Se puede implementar fácilmente si recordamos que:

$$A \to B \Leftrightarrow \neg A \vee B$$

Esto se puede corroborar con la equivalencia existente en sus tablas de verdad.
Ahora, la fórmula lógica se puede implementar de la siguiente forma con conectivos lógicos:



A	B	$A \to B$
0	0	1
0	1	1
1	0	0
1	1	1

Ejercicios - Respuesta

Implemente un circuito 2 bit *Multiplier* que realice la multiplicación entre dos valores de 2 bits.

Para ver esto de forma sencilla, primero vemos cómo implementar una multiplicación entre dos números de un bit:

A	B	$A * B$
0	0	0
0	1	0
1	0	0
1	1	1

Es fácil ver que la multiplicación la podemos realizar a partir de la compuerta lógica `AND`. Ahora, como nos piden una multiplicación entre dos números de dos bits, tenemos que seguir el método de multiplicación tradicional. Para cumplir este objetivo es importante recordar la composición de los *half-adders*, en la que la suma entre dos números se representa por la compuerta `XOR`, mientras que el *carry* resultante se hace con la compuerta `AND`[1].

[1] **¿Por qué?** Porque si ambos bits de entrada son iguales a 1, entonces sabemos que la suma resultará en 0, "sobrando" una unidad (como lo vemos en la suma tradicional).

Sigue en la siguiente diapositiva.

Ejercicios - Respuesta

Implemente un circuito 2 bit *Multiplier*, que realice la multiplicación entre dos valores de 2 bits.

IMPORTANTE: Aquí la respuesta depende también de la interpretación del ejercicio. Por ejemplo, si se quiere respetar la cantidad de bits por número, se pueden entregar solo las salidas C0 y C1 en conjunto con C2" como Cout. En contexto de evaluación de este curso, nos aseguraremos de que esto siempre esté explicitado.

Sean A y B dos números de dos bits de la forma A_1A_0 y B_1B_0. Si queremos obtener el número $C = A * B$, seguimos el siguiente procedimiento:

- Multiplicamos el bit menos significativo de B por los dos bits de A. De esta forma, tendremos A_0 `AND` $B_0 = C_0$, y A_1 `AND` $B_0 = C_1'$.
- Multiplicamos ahora el bit más significativo de B por los dos bits de A. De esta forma, tendremos A_0 `AND` $B_1 = C_1''$ y A_1 `AND` $B_1 = C_2'$
- El bit menos significativo de nuestro resultado será C_0. Luego, el bit siguiente se obtiene de la siguiente forma: $C_1 = C_1'$ `XOR` C_1'' (tal como lo hacemos con la suma). Como esto nos puede generar un carry, tomamos $C_2'' = C_1'$ `AND` C_1''
- Ahora, el siguiente bit lo conseguimos como la suma entre el producto de los bits más significativos de cada número, sumado al carry anterior: $C_2 = C_2'$ `XOR` C_2''.
- Finalmente, como nuestro número puede tener máximo 4 bits (11 * 11 = 1001), tomamos el bit más significativo del resultado como el carry de la última suma: $C_3 = C_2'$ `AND` C_2''.

Finalmente, nuestro circuito queda de la siguiente forma:

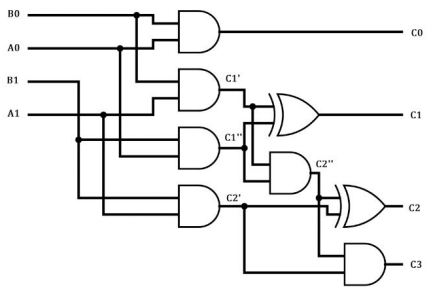


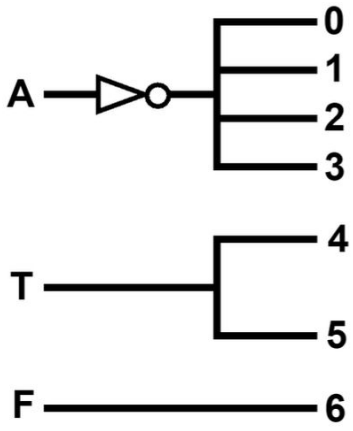
Figura 2: Resultado del circuito descrito.

Notar que este diagrama se puede simplificar haciendo uso de *half-adders* y *full-adders*.

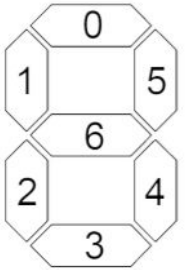
Ejercicios - Respuesta

Construya con compuertas lógicas el controlador de un *display* de 7 segmentos. Este controlador recibe como *input* solo un bit *A*, *i.e.*, el *display* de 7 segmentos solo mostrará 0 o 1.

Un circuito controlador del *display* puede ser el siguiente:



Donde *T* corresponde a una señal constante de 1 lógico (*true*) y F a una señal constante de 0 lógico (*false*).



Ejercicios - Respuesta

¿Qué número entero es generado al realizar cuatro operaciones *shift right* seguidas de cinco operaciones *rotate left* a un registro de 8 bits que inicialmente almacena el número entero 79?

Para interpretar bien el resultado, consideramos la representación binaria para números enteros con signo (pues, por enunciado, sabemos que debemos considerar números negativos).

Tenemos que $79 = 01001111$. Ahora, iremos listando los números resultantes:

- `shift right`: $00100111 \rightarrow 39$
- `shift right`: $00010011 \rightarrow 19$
- `shift right`: $00001001 \rightarrow 9$
- `shift right`: $00000100 \rightarrow 4$
- `rotate left`: $00001000 \rightarrow 8$
- `rotate left`: $00010000 \rightarrow 16$
- `rotate left`: $00100000 \rightarrow 32$
- `rotate left`: $01000000 \rightarrow 64$
- `rotate left`: $10000000 \rightarrow -128$

Notemos que en esta secuencia el uso de `rotate left` no difiere de `shift left`, ya que no hubo ningún bit entre medio que pudiera ser conservado. Si se hiciera un `rotate left` más:

- `rotate left`: $00000001 \rightarrow 1$

Aquí sí se aprecia la diferencia, donde el bit más significativo ahora se vuelve el menor en vez de perderse, como es en el caso de `shift left`.

Ejercicios - Respuesta

Un comparador de números es un circuito que, dados dos números *A* y *B* en representación posicional, indica cuál es mayor o si estos son iguales. El circuito posee tres salidas, donde la primera entrega un 1 solo si *A* es mayor, la segunda un 1 solo si ambas son iguales, y la tercera un 1 solo si *B* es mayor.

Diseñe un comparador de números naturales de *N* bits.

Solución en la siguiente diapositiva.

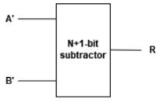
Ejercicios - Respuesta

Respuesta: Es de gran utilidad implementar un restador que tenga de entradas los números A y B. Ahora, como no contamos con representaciones negativas, es de utilidad que nuestro restador sea de $N+1$ bits y, para A, B definimos A', B' tales que tienen un bit adicional igual a 0 en su posición más significativa. Ahora, Considerando $R = A' - B'$, podemos realizar el siguiente análisis del resultado a partir del bit más significativo:

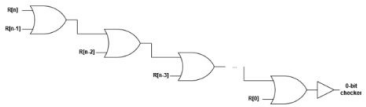
- Si $R_N = 1$, el resultado es negativo (lo que implica que A era menor a B).
- Si $R_N = 0$, el resultado es positivo o cero. Para determinar si es 0, basta con utilizar una secuencia de compuertas $\vee$ para R, de forma que retorna 0 si, y solo si todos los bits en R son iguales a cero.

Entonces, nuestro circuito se ve como sigue:

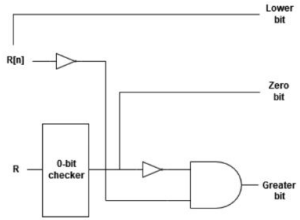
Restador de $N+1$ bits



Circuito para detectar $R = 0$



Circuito final



De esta forma, tenemos que:

- `0-bit checker` es un bit igual a 1 si, y solo si todos los bits de R son iguales a cero (*i.e.* $A = B$).
- `Lower bit` es igual a 1 si, y solo si el bit más significativo del resultado es igual a 1 (lo que implica $A < B$).
- `Greater bit` es igual a 1 si, y solo si el bit más significativo del resultado es igual a 0 (lo que implica que $A \geq B$) **y, además,** existe al menos un bit igual a 1 en el resultado (*i.e.* $A \neq B \rightarrow A > B$).
- `Zero bit` es igual a 1 si, y solo si el bit más significativo del resultado es igual a 0 (lo que implica que $A \geq B$) **y, además,** todos los bits del resultado son iguales a cero (*i.e.* $A = B$).

Ejercicios - Respuesta

Extienda el circuito anterior para realizar la comparación entre el valor absoluto de dos números *A* y *B* de *N* bits.

Basta con usar el circuito anterior conectado a un selector de valor absoluto para *A* y para *B*. De esta forma, realizamos la comparación de valores con números positivos (por lo que el comparador de números naturales entregará el valor esperado).

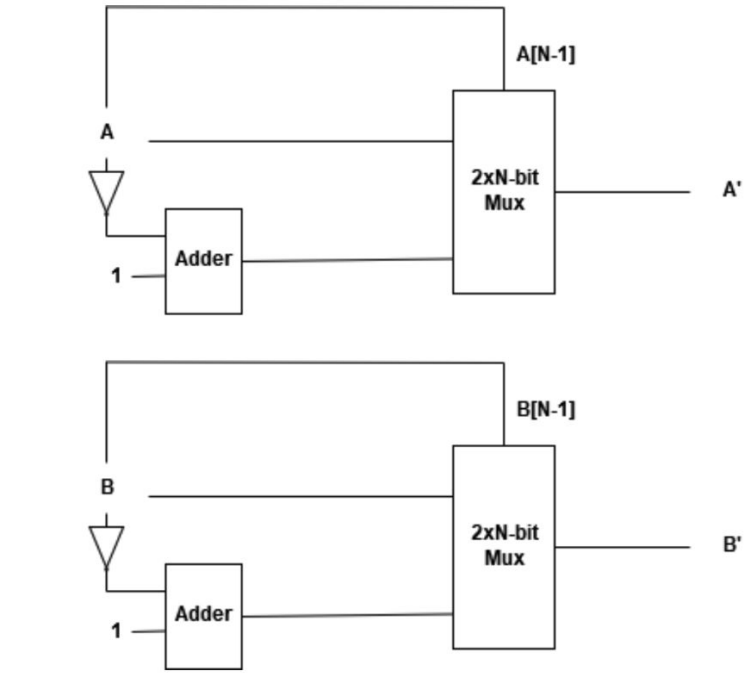
Circuito de selección en la siguiente diapositiva.

Ejercicios - Respuesta

¿Por qué usamos el bit más significativo de *A* y *B*?

Porque en representación de números enteros, este representa el signo. Así, si *A*[*N-1*] = 0, entonces *A* es positivo y seleccionamos su valor original. En otro caso, al ser *A*[*N-1*] = 1 seleccionaremos su complemento de 2 (que en este caso será su valor positivo).

Circuito de valor absoluto



En este caso, A' y B' son los valores absolutos de A y B, respectivamente.