Santiago Chiale

santiagochiale@abc.gob.ar

LOGICA COMPUTACIONAL

Contenido

[Introducción 2](#_Toc194008592)

[CAPÍTULO 1: LÓGICA PROPOSICIONAL 4](#_Toc194008593)

[Antecedentes Históricos y Relevancia Contemporánea 4](#_Toc194008594)

[Conceptos Fundamentales 4](#_Toc194008595)

[Proposiciones 4](#_Toc194008596)

[Conectivos Lógicos 5](#_Toc194008597)

[Tablas de verdad 6](#_Toc194008598)

[Ejercicios Propuestos 8](#_Toc194008599)

[Ejercicio 1: Identificación de Proposiciones 8](#_Toc194008600)

[Ejercicio 2: Evaluación de Expresiones Lógicas 8](#_Toc194008601)

[Ejercicio 3: codepen 11](#_Toc194008602)

[Aplicación práctica: Compuertas lógicas en circuitos inteligentes 11](#_Toc194008603)

[Caso real: Activación de un extractor de aire por temperatura y humedad 11](#_Toc194008604)

[Conclusión 13](#_Toc194008605)

[CAPÍTULO 2: LÓGICA DE PREDICADOS 14](#_Toc194008606)

[Cuantificadores universal y existencial, interpretación en lenguajes de primer orden 14](#_Toc194008607)

[¿Qué es un lenguaje de primer orden? 14](#_Toc194008608)

[Sistemas formales: corrección y completitud, modelos e interpretaciones 15](#_Toc194008609)

[Corrección y completitud 15](#_Toc194008610)

[Modelos e interpretaciones 16](#_Toc194008611)

[Límites de la lógica de predicados: intratabilidad e inexpresabilidad en ciertos problemas 16](#_Toc194008612)

[Aplicación en la deducción natural, estrategias de formalización y deducción 16](#_Toc194008613)

[Ejercicio práctico (Aplicación Web) 17](#_Toc194008614)

[CAPÍTULO 3: LÓGICA DIGITAL 18](#_Toc194008615)

[Introducción al Álgebra de Boole y a las Compuertas Lógicas 18](#_Toc194008616)

[Dispositivos 21](#_Toc194008617)

[Multiplexores (MUX) 21](#_Toc194008618)

[Decodificadores 22](#_Toc194008619)

[Biestables (Flip-Flops) 22](#_Toc194008620)

[Memorias 23](#_Toc194008621)

[Microcontroladores 23](#_Toc194008622)

[Diseño a Mayor Escala: Módulos y Jerarquía 24](#_Toc194008623)

[Conclusiones 24](#_Toc194008624)

# Introducción

A lo largo de la historia, la humanidad ha buscado diversas formas de representar y procesar la información. ¿Cómo es posible que un dispositivo tan cotidiano como una calculadora “comprenda” que, al presionar los botones correspondientes a “5”, “+”, “3” y “=”, el resultado que debe mostrar es “8”?

¿De qué manera una computadora personal (PC) es capaz de ejecutar miles de instrucciones por segundo para gestionar tareas que van desde la edición de textos hasta la navegación por internet?

Y, finalmente, ¿cómo se llega al desarrollo de la inteligencia artificial (IA), que puede procesar millones de datos, reconocer rostros o jugar partidas de ajedrez con gran maestría?

La clave común en todos estos procesos –ya sea en la calculadora, en la PC o en la IA– reside en los principios lógicos y en los circuitos electrónicos que administran señales binarias, es decir, secuencias de “0” y “1”. La lógica computacional constituye la base fundamental en la que se diseñan y organizan estos dígitos, transformándolos en operaciones aritméticas, lógicas y, en última instancia, en acciones concretas: la pantalla muestra un resultado, el procesador ejecuta un programa y una red neuronal lleva a cabo una fase de aprendizaje.

Imagina que partimos de una calculadora: cada vez que se introducen dos números y se selecciona un símbolo de operación (como PLUS, MINUS, MULTIPLY, o DIVIDE), la máquina debe convertir esa entrada en impulsos eléctricos compuestos por niveles “alto” (1) y “bajo” (0). Lo que para el usuario es una simple suma o resta se transforma internamente en una serie de operaciones lógicas y aritméticas elementales, organizadas en circuitos mediante puertas lógicas como AND, OR, NOT, entre otras. Sin la estructura lógica basada en la teoría booleana, la calculadora no podría interpretar correctamente los dígitos ingresados.

El mismo principio se extiende a una computadora personal. Aquí se pasa de manejar un par de señales a coordinar millones de ellas, utilizando un procesador que ejecuta instrucciones almacenadas en la memoria. Detrás de la carcasa de una PC, tanto la CPU como la memoria se basan en biestables, buses y en compuertas lógicas más complejas, como multiplexores y decodificadores. Comprender el funcionamiento de estos “ladrillos digitales” nos brinda una visión clara de lo que ocurre cuando se inicia un programa, se guarda un archivo o se establece una conexión a internet. Cada clic y cada instrucción se implementa, en última instancia, mediante las mismas señales binarizadas que se utilizan en una calculadora, pero en una escala muchísimo mayor.

Por último, la inteligencia artificial lleva esta lógica computacional a un nivel superior. En la IA se aplican redes neuronales, algoritmos de aprendizaje y se manejan grandes volúmenes de datos. Sin embargo, en el núcleo de todas estas operaciones complejas se encuentran las operaciones sobre bits –multiplicaciones de matrices, pasos de backpropagation y otros cálculos– que se implementan en hardware especializado (como GPUs, TPUs o ASICs) y que están diseñados con puertas lógicas, registros, sumadores y decodificadores.

¿Por qué es importante estudiar esta base de la lógica computacional?

Para desentrañar el funcionamiento interno de la tecnología: Lo que puede parecer un misterio o una “caja negra” se revela como un conjunto de principios claros basados en “0”, “1” y en el uso de puertas lógicas.

Para comprender las limitaciones inherentes a los sistemas digitales: Al dominar la representación de la información y las operaciones lógicas, se entiende por qué existen límites, como el desbordamiento numérico, la velocidad del reloj y la latencia de la memoria.

Para apreciar la continuidad conceptual entre dispositivos simples y sistemas avanzados: Desde una calculadora elemental hasta el “cerebro” de un sistema de IA, la base es la lógica digital que, ampliada con técnicas de programación y matemáticas avanzadas, permite alcanzar niveles de complejidad sorprendentes.

En este manual se explorarán:

La forma en que los enunciados lógicos y la teoría booleana se traducen en circuitos digitales mediante compuertas, flip-flops, decodificadores y multiplexores.

La importancia del diseño de estos circuitos en la evolución tecnológica, desde la calculadora hasta la computadora personal y, en última instancia, hasta la inteligencia artificial.

Cómo el razonamiento formal –tanto en la lógica proposicional como en la lógica de predicados– se utiliza para describir el comportamiento de sistemas y para verificar su correcta implementación en proyectos de ingeniería de software, data science e IA.

A medida que avances por cada capítulo, descubrirás progresivamente:

En primer lugar, los fundamentos de la lógica proposicional y la manipulación de circuitos básicos.

En segundo lugar, la lógica de predicados, que extiende la capacidad descriptiva mediante la introducción de cuantificadores.

Finalmente, cómo estas ideas abstractas se concretan en hardware real, abriendo el camino para comprender desde el funcionamiento de un microprocesador hasta cómo las redes neuronales pueden clasificar miles de imágenes por segundo o procesar lenguaje natural, gracias a la integración de lógica, electrónica y computación.

# CAPÍTULO 1: LÓGICA PROPOSICIONAL

## Antecedentes Históricos y Relevancia Contemporánea

El estudio de la lógica proposicional tiene raíces que se remontan a la Antigua Grecia, donde Aristóteles sentó las bases de lo que hoy conocemos como lógica silogística. Sin embargo, la idea de tratar las proposiciones como entidades formales, que pueden combinarse mediante conectores lógicos, comenzó a consolidarse a partir de los trabajos de George Boole en el siglo XIX, quien desarrolló lo que ahora se conoce como álgebra booleana. Posteriormente, durante el siglo XX, figuras como Hilbert, Frege y Russell sistematizaron la lógica formal, lo que permitió establecer los fundamentos de la lógica computacional que subyace en la informática moderna.

Hoy en día, la lógica proposicional es considerada la puerta de entrada al razonamiento formal. No solo tiene un profundo valor filosófico, sino que también es de enorme relevancia en áreas como la ingeniería de software, la inteligencia artificial, la verificación de circuitos digitales y la teoría de lenguajes formales.

## Conceptos Fundamentales

### Proposiciones

Una proposición es un enunciado declarativo que puede evaluarse como verdadero (V) o falso (F). No se consideran proposiciones aquellas expresiones que no admiten una clasificación clara de verdad, como preguntas, órdenes o exclamaciones.

Ejemplos de proposiciones:

* "7 es un número primo" → Puede ser evaluada matemáticamente como verdadera.
* "Haré café mañana" → Aunque su evaluación depende de circunstancias futuras, sigue siendo una proposición porque en algún momento podrá clasificarse como V o F.

Ejemplos de expresiones que no son proposiciones:

* "¿Hace calor hoy?" → No es declarativa, es una pregunta.
* "¡Qué hermoso día!" → Es una exclamación sin valor de verdad.
* "Cierra la puerta." → Es una orden, no tiene valor de verdad.

#### Variables Proposicionales

Las variables proposicionales se utilizan para representar enunciados de forma simbólica. Se denotan con letras como P, Q, R, S, ... y pueden tomar solo dos valores: verdadero o falso.

**Ejemplo:**

* P = "Está lloviendo"

Si está lloviendo, P es verdadero; si no, es falso.

Las variables proposicionales permiten generalizar razonamientos y aplicar reglas lógicas de manera más eficiente.

### Conectivos Lógicos

Los conectivos lógicos permiten combinar proposiciones para formar expresiones más complejas. Sus principales operadores son:

* Negación (NOT, !P) → Invierte el valor de la proposición.
* Conjunción (AND, P & Q) → Es verdadera solo si ambas proposiciones son verdaderas.
* Disyunción (OR, P || Q) → Es verdadera si al menos una proposición es verdadera.
* Implicación (IMPLIES, P → Q) → Solo es falsa cuando P es verdadera y Q es falsa.
* Bicondicional (IFF, P ↔ Q) → Es verdadera si ambas proposiciones tienen el mismo valor de verdad.

#### Propiedades de los Conectivos Lógicos

Algunas propiedades fundamentales en la lógica proposicional son:

1. Conmutatividad

Estas leyes indican que el orden de los operandos no afecta el resultado:

P || Q === Q || P

P && Q === Q && P

1. Asociatividad

Estas propiedades dicen que la agrupación entre paréntesis no cambia el resultado:

(P || Q) || R === P || (Q || R)

(P && Q) && R === P && (Q && R)

1. Distributividad

Permite reorganizar expresiones combinando AND y OR:

P && (Q || R) === (P && Q) || (P && R)

P || (Q && R) === (P || Q) && (P || R)

1. Leyes de De Morgan

Muestran cómo distribuir la negación sobre una conjunción o disyunción:

!(P && Q) === !P || !Q

!(P || Q) === !P && !Q

### Tablas de verdad

Las tablas de verdad son herramientas fundamentales en lógica proposicional. Sirven para analizar el valor de verdad (Verdadero o Falso) de una fórmula lógica en función de todas las combinaciones posibles de valores de sus variables.

Se utilizan para:

* Verificar si una fórmula es siempre verdadera (tautología), siempre falsa (contradicción) o satisfacible.
* Estudiar la validez de argumentos.
* Analizar circuitos lógicos y condiciones en programación.

#### Como se construye una tabla de verdad

1. Identificar las variables proposicionales

Primero, contá cuántas variables contiene la fórmula (por ejemplo, P, Q, R).

* + Si hay n variables, la tabla tendrá 2n2^n2n filas, porque cada variable puede ser V (Verdadero) o F (Falso).
    - 1 variable → 2 combinaciones
    - 2 variables → 4 combinaciones
    - 3 variables → 8 combinaciones

1. Enumerar todas las combinaciones posibles

Listá todas las combinaciones de verdadero (V) y falso (F) para esas variables. Usá un patrón binario descendente.

Ejemplo con 2 variables:

|  |  |
| --- | --- |
| **P** | **Q** |
| F | F |
| F | V |
| V | F |
| V | V |

1. Evaluar la fórmula paso a paso

Agregar columnas intermedias si la expresión es compleja, para calcular por partes.

Ejemplo, para P && !Q, se puede construir así:

|  |  |  |  |
| --- | --- | --- | --- |
| **P** | **Q** | **!Q** | **P & !Q** |
| F | F | V | F |
| F | V | F | F |
| V | F | V | V |
| V | V | F | F |

1. Interpretar los resultados
   * Una vez completada la tabla:
   * Si la última columna es siempre V, la fórmula es una tautología.
   * Si es siempre F, es una contradicción.
   * Si tiene al menos una V, es satisfacible.

¿Para qué se usan?

* Verificación de circuitos digitales.
* Programación: análisis de condiciones lógicas en if, while, etc.
* Diseño lógico: ayuda a convertir expresiones en formas equivalentes o simplificadas.
* Lógica matemática: comprobar propiedades, leyes o validez de razonamientos.

Ejemplo para la conjunción (AND):

|  |  |  |
| --- | --- | --- |
| P | Q | P & Q |
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

## Ejercicios Propuestos

### Ejercicio 1: Identificación de Proposiciones

Dado el siguiente conjunto de enunciados, indicar cuáles son proposiciones y justificar la respuesta:

1. "El sol es una estrella."
2. "Apaga la luz."
3. "5 + 7 = 13."
4. "¿Vendrás mañana?"
5. "El cielo es azul y el agua es transparente."

### Ejercicio 2: Evaluación de Expresiones Lógicas

Para cada una de las siguientes expresiones lógicas, construí la tabla de verdad completa considerando todas las combinaciones posibles de los valores de P, Q y (si corresponde) R.

#### 1. P || Q

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **P || Q** |
| F | F | F |
| F | V | V |
| V | F | V |
| V | V | V |

#### 2. !P && !Q

Tabla intermedia:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P** | **Q** | **!P** | **!Q** | **!P & !Q** |
| F | F | V | V | V |
| F | V | V | F | F |
| V | F | F | V | F |
| V | V | F | F | F |

Resultado:

|  |  |  |
| --- | --- | --- |
| **P** | **Q** | **!P & !Q** |
| F | F | V |
| F | V | F |
| V | F | F |
| V | V | F |

#### 3. (!P || Q) || R

Tabla intermedia:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **R** | **!P** | **(!P || Q)** | **(!P || Q) || R** |
| F | F | F | V | V | V |
| F | F | V | V | V | V |
| F | V | F | V | V | V |
| F | V | V | V | V | V |
| V | F | F | F | F | F |
| V | F | V | F | F | V |
| V | V | F | F | V | V |
| V | V | V | F | V | V |

Resultado:

|  |  |  |  |
| --- | --- | --- | --- |
| **P** | **Q** | **R** | **(!P || Q) || R** |
| F | F | F | V |
| F | F | V | V |
| F | V | F | V |
| F | V | V | V |
| V | F | F | F |
| V | F | V | V |
| V | V | F | V |
| V | V | V | V |

#### 4. !(P & Q) || R

Tabla intermedia:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **R** | **P & Q** | **! (P & Q)** | **! (P & Q)|| R** |
| F | F | F | F | V | V |
| F | F | V | F | V | V |
| F | V | F | F | V | V |
| F | V | V | F | V | V |
| V | F | F | F | V | V |
| V | F | V | F | V | V |
| V | V | F | V | F | F |
| V | V | V | V | F | V |

Resultado:

|  |  |  |  |
| --- | --- | --- | --- |
| **P** | **Q** | **R** | **! (P & Q)|| R** |
| F | F | F | V |
| F | F | V | V |
| F | V | F | V |
| F | V | V | V |
| V | F | F | V |
| V | F | V | V |
| V | V | F | F |
| V | V | V | V |

Instrucciones para los alumnos:

* Enumerá todas las combinaciones posibles de valores de P, Q (y R cuando sea necesario).
* Evaluá paso a paso cada subexpresión si es necesario.
* Indicá si el resultado final de la expresión es V (verdadero) o F (falso) en cada caso.

### Ejercicio 3: codepen

Escriba una sencilla aplicación web que simule los ejercicios anteriores.

Parámetros de diseño:

* Debe estar escrita en JavaScript
* En el front debe contener inputs para colocar los valores con las etiquetas correspondientes (P, Q y R)
* Estos inputs solo deben admitir valores V o F o bien 1 o 0
* El resultado de la proposición debe verse en otro input pero sobre el cual no se pueda ingresar valores

Aclaraciones

* Se puede realizar un grupo de inputs para cada ejercicio o se puede reutilizar los mismos inputs indicando el tipo de operación a realizar con una lista desplegable al lado de cada uno (Pensar en la escalabilidad)

## Aplicación práctica: Compuertas lógicas en circuitos inteligentes

### Caso real: Activación de un extractor de aire por temperatura y humedad

Supongamos que diseñamos un sistema inteligente que activa un extractor cuando:

- La temperatura supera los 30°C  
- Y la humedad supera el 70%  
  
Este sistema puede construirse con una compuerta lógica AND, ya que ambas condiciones deben cumplirse.

Tabla de verdad (Lógica proposicional)

|  |  |  |
| --- | --- | --- |
| Temperatura alta (A) | Humedad alta (B) | Salida (A AND B) |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 ✅ |

#### Implementación en electrónica digital

##### Circuito con compuerta AND (CI 7408)

• 7408 es un circuito integrado que contiene 4 compuertas AND de 2 entradas.  
• Entradas A y B se conectan a sensores digitales.  
• La salida activa un relé/transistor para encender el extractor.

##### Simulación simple del sistema lógico

# Versión en Python

# Simulación simple del sistema lógico:

temperatura = int(input("¿Temperatura alta? (1=SÍ, 0=NO): "))

humedad = int(input("¿Humedad alta? (1=SÍ, 0=NO): "))

extractor = temperatura and humedad

if extractor:

print("✅ Activar extractor")

else:

print("⛔ No activar extractor")

##### Versión en Arduino (microcontrolador)

# Versión en Arduino (microcontrolador)

sensorTemp = 2

sensorHum = 3

extractor = 13

def setup():

pinMode(sensorTemp, INPUT)

pinMode(sensorHum, INPUT)

pinMode(extractor, OUTPUT)

def loop():

A = digitalRead(sensorTemp)

B = digitalRead(sensorHum)

if A and B:

digitalWrite(extractor, HIGH) # Encender

else:

digitalWrite(extractor, LOW) # Apagar

delay(1000) # Esperar 1 segundo

Este ejemplo muestra cómo:

- La lógica proposicional se implementa directamente con compuertas lógicas.

- Los circuitos digitales usan estos principios para ejecutar decisiones físicas.

- La inteligencia artificial embebida y el análisis de datos también requieren estructuras lógicas para inferir comportamientos.

## Conclusión

Este ejemplo muestra cómo:

- La lógica proposicional se implementa directamente con compuertas lógicas.  
- Los circuitos digitales usan estos principios para ejecutar decisiones físicas.  
- La inteligencia artificial embebida y el análisis de datos también requieren estructuras lógicas para inferir comportamientos.

# CAPÍTULO 2: LÓGICA DE PREDICADOS

## Cuantificadores universal y existencial, interpretación en lenguajes de primer orden

La lógica de predicados, también conocida como lógica de primer orden, constituye una herramienta fundamental para representar conocimiento de forma estructurada y precisa. A diferencia de la lógica proposicional, que se limita a manipular enunciados completos como "P" o "Q", la lógica de predicados permite analizar la estructura interna de dichos enunciados al introducir **variables**, **predicados** y **cuantificadores**.

### ¿Qué es un lenguaje de primer orden?

Un **lenguaje de primer orden** es un sistema formal que nos permite construir fórmulas lógicas a partir de un conjunto de símbolos. Incluye:

* **Símbolos de constante**: representan objetos concretos (ej. ana, producto1).
* **Símbolos de función**: denotan operaciones sobre objetos (ej. edad(x), compras(x)).
* **Símbolos de predicado**: describen propiedades o relaciones (ej. Usuario(x), Compra(x, y)).
* **Variables**: x, y, z, que recorren el dominio de discurso.

#### Cuantificadores

* **FOR ALL x** (universal): la fórmula es verdadera para todos los elementos del dominio.
* **THERE EXISTS x** (existencial): hay al menos un elemento para el que la fórmula es verdadera.
* **Conectivos lógicos**: ! (negación), & (conjunción), || (disyunción), → (implicación), ↔ (bicondicional).

Este lenguaje permite representar expresiones complejas como:

// FOR ALL x (Usuario(x) -> RecibeRecomendacion(x))

JavaScript

usuarios.forEach(u => {

if (u.esUsuario && !u.recibeRecomendacion) cumple = false;

});

// THERE EXISTS x (Cliente(x) && !Pago(x))

JavaScript

let existeMoroso = clientes.some(c => c.esCliente && !c.pagado);

#### Interpretación

Para que una fórmula tenga sentido, se debe asignar un significado concreto a sus símbolos, lo que se denomina **interpretación**. Esta incluye:

* Un **dominio** D (el conjunto de objetos del cual se habla).
* Asignaciones de significado para constantes, funciones y predicados dentro del dominio.

Por ejemplo, en un sistema de IA que analiza patrones de compra, el dominio puede ser el conjunto de usuarios de una tienda, y un predicado como Compra(x, y) indica si el usuario x compró el producto y.

## Sistemas formales: corrección y completitud, modelos e interpretaciones

Los **sistemas formales** en lógica de primer orden definen el conjunto de reglas sintácticas y semánticas que permiten construir argumentos válidos.

Comprenden:

* **Un lenguaje formal** (como el descrito en 2.1)
* **Reglas de inferencia** (para deducir nuevas fórmulas)
* **Axiomas** (enunciados asumidos como verdaderos)

### Corrección y completitud

Un sistema es **correcto** si todo lo que se puede derivar formalmente (mediante pruebas) es verdadero en todas las interpretaciones posibles. Esto asegura que las deducciones son confiables.

Es **completo** si todas las fórmulas verdaderas en todas las interpretaciones pueden ser demostradas mediante el sistema. Esto asegura que el sistema es expresivo.

// Si Usuario(x) && CompraFrecuente(x) entonces VIP(x)

JavaScript

if (usuario.esUsuario && usuario.compraFrecuente) {

usuario.vip = true;

}

### Modelos e interpretaciones

Un **modelo** es una interpretación en la que todas las fórmulas de una teoría (conjunto de fórmulas) se cumplen. Por ejemplo, una base de datos con usuarios y sus compras puede ser vista como un modelo si satisface todas las reglas lógicas definidas para el análisis de datos.

## Límites de la lógica de predicados: intratabilidad e inexpresabilidad en ciertos problemas

Aunque poderosa, la lógica de predicados presenta límites tanto **computacionales** como **expresivos**.

#### Intratabilidad

Verificar si una fórmula arbitraria en lógica de primer orden es válida puede ser **semi-decidible**. Esto hace que algunas consultas lógicas no puedan resolverse eficientemente en la práctica.

// Algoritmo que podría no terminar si se usan datos infinitos o no acotados

JavaScript

function resolver(formula) {

while (!encontrado && !agotado) {

intentarProbar(formula);

}

}

#### Inexpresabilidad

Hay propiedades que no pueden ser representadas en la lógica de primer orden.

Ejemplo:

"Para cada subconjunto de productos, existe un cliente que los ha comprado todos" → requiere **lógica de segundo orden**.

## Aplicación en la deducción natural, estrategias de formalización y deducción

La **deducción natural** es una metodología que formaliza cómo razonamos de manera intuitiva. En el contexto de IA y análisis de datos, permite automatizar inferencias lógicas dentro de sistemas expertos, validaciones de hipótesis y construcción de sistemas de reglas.

**Reglas clave de deducción natural**

**FOR ALL (eliminación)**:

// FOR ALL x P(x)

clientes.forEach(c => {

if (!P(c)) valido = false;

});

**THERE EXISTS (introducción)**:

let cumple = clientes.some(c => P(c));

### Ejercicio práctico (Aplicación Web)

Crea una aplicación web interactiva donde:

* Se permita ingresar un conjunto de individuos (x1, x2, x3...)
* Para cada uno, se indique si cumple ciertas condiciones (por ejemplo, Compras > 3)
* Se evalúe una fórmula como:

// FOR ALL x (Compras(x) > 3 → Recomendar(x))

let valido = usuarios.every(u => u.compras <= 3 || u.recomendar);

## CAPÍTULO 3: LÓGICA DIGITAL

### Introducción al Álgebra de Boole y a las Compuertas Lógicas

La lógica digital se fundamenta en la aplicación de los principios de la lógica booleana al diseño de circuitos electrónicos. En estos circuitos, la información se representa mediante señales binarias, donde el "0" se asimila a falso y el "1" a verdadero.

#### Antecedentes Históricos

El desarrollo de la lógica digital comenzó con los estudios de George Boole en el siglo XIX, quien formalizó un álgebra basada en operaciones como AND, OR y NOT. Posteriormente, en 1937, Claude Shannon demostró cómo aplicar este álgebra para analizar y optimizar circuitos de conmutación eléctrica, lo que representó un hito importante en la ingeniería de la información.

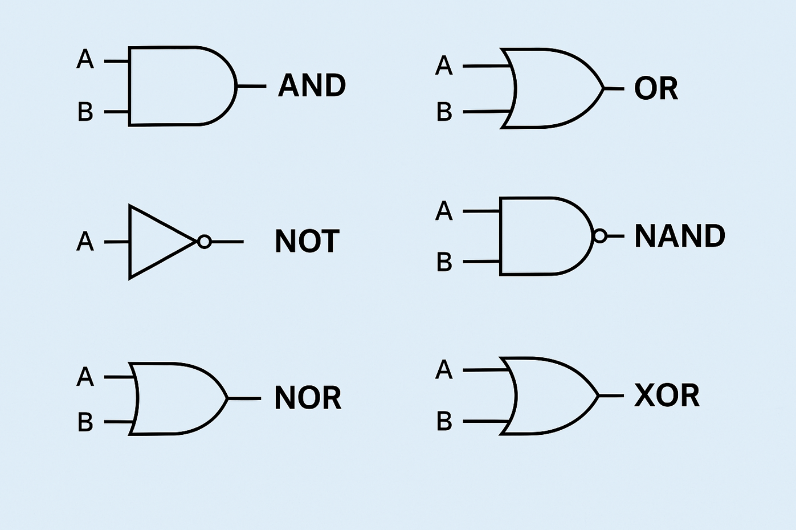
#### Variables, Operaciones y Diagramas

En el contexto de circuitos digitales, cada señal de entrada o salida se considera una variable binaria (con valor 0 o 1). Para combinar estas señales se utilizan compuertas lógicas, que implementan operaciones booleanas tales como:

* AND: La salida es 1 solo si todas las entradas son 1.
* OR: La salida es 1 si al menos una de las entradas es 1.
* NOT: La salida es la inversión del valor de la entrada.
* NAND: Equivale a NOT (AND); es decir, primero se realiza la operación AND y luego se niega el resultado.
* NOR: Equivale a NOT (OR); se realiza la operación OR y luego se niega.
* XOR (Exclusive OR): La salida es 1 si un número impar de entradas es 1 (por ejemplo, en la versión de dos entradas, si exactamente una es 1).

Asimismo, en los diagramas de circuitos se utiliza una notación gráfica específica para representar cada compuerta.

Por ejemplo, la compuerta AND se dibuja con una forma que puede asemejarse a una "D" invertida, mientras que la compuerta NOT se representa mediante un triángulo con un pequeño círculo (indicador de inversión) en su salida.



#### Diseño y Optimización de Circuitos Digitales

El diseño de circuitos digitales se basa en la manipulación de funciones booleanas, que son expresiones en variables (por ejemplo, A, B, C) combinadas mediante compuertas lógicas. Cada salida de un circuito se define mediante una función booleana.

Por ejemplo, un decodificador 3:8 se describe mediante 8 funciones booleanas, cada una activada por combinaciones específicas de las entradas A, B y C.

Considera la siguiente función booleana de tres variables:

F(A, B, C) = (A AND NOT B) OR (B AND C).

La implementación de esta función con compuertas lógicas reflejará la estructura de la expresión original. No obstante, la expresión inicial puede no ser la más óptima en términos de cantidad de compuertas utilizadas. Por ello, se busca optimizar la implementación para reducir costos, disminuir la latencia y mejorar la eficiencia energética.

##### Ejemplo Ilustrativo: Minimización con Mapas de Karnaugh

Supongamos que se tiene la función booleana F definida mediante la siguiente tabla de verdad para tres variables (A, B, C):

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | F |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

En esta tabla, F es 1 para las combinaciones:

A = 0, B = 0, C = 1

A = 0, B = 1, C = 1

A = 1, B = 0, C = 1

A = 1, B = 1, C = 0

A = 1, B = 1, C = 1

Para simplificar esta función se puede utilizar un Mapa de Karnaugh (K-map) de 3 variables. Su disposición típica se organiza con las combinaciones de B y C en columnas y los valores de A en filas, de modo que se puedan agrupar las celdas adyacentes en bloques de tamaño 1, 2 o 4 (siempre en potencias de 2). La agrupación de estas celdas permite encontrar una expresión booleana minimizada.

Una posible simplificación de F puede ser:

F = (NOT A AND B) OR (B AND C).

Esta expresión es significativamente más compacta que la que se obtendría listando todos los minterms originales.

La implementación del circuito se realiza de la siguiente manera:

Primer término: se implementa la función "NOT A AND B".

Segundo término: se implementa la función "B AND C".

Finalmente, ambos resultados se combinan utilizando una compuerta OR.

Cabe mencionar que existen diversas estrategias de agrupamiento en un mapa de Karnaugh, y otra forma equivalente podría ser, por ejemplo, "B AND (C OR NOT A)".

#### Conexión con Puertas Universales: NAND y NOR

Una vez obtenida la expresión minimizada, es posible rediseñar el circuito utilizando únicamente compuertas NAND. Esto se logra mediante las siguientes equivalencias:

* NOT X: Se implementa como "NAND(X, X)".
* X AND Y: Se puede obtener como "NOT (NAND(X, Y))".
* X OR Y: Se obtiene a partir de "NOT (NAND(NOT X, NOT Y))".

Este método permite reducir la variedad de componentes en el circuito, simplificando la manufactura, ya que se utiliza un único tipo de compuerta (NAND o NOR) en toda la implementación.

##### Caso Práctico: Trabajo con Simuladores Gratuitos

Un ejercicio integral que se sugiere para los estudiantes es diseñar y optimizar un circuito digital utilizando simuladores gratuitos, como Logisim o Digital. El proceso general incluye los siguientes pasos:

* Objetivo del Trabajo Práctico
* Diseñar un circuito que reciba tres bits de entrada (A, B, C) y produzca una salida F, definida por una tabla de verdad.
* Optimizar la expresión booleana resultante utilizando Mapas de Karnaugh para obtener la forma más simplificada posible.

Implementar la versión final en un simulador de circuitos y verificar que el comportamiento del circuito coincide con la tabla de verdad inicial.

##### Herramientas y Simuladores Gratuitos

Se pueden utilizar diversas herramientas, entre las que destacan:

* Logisim (y su versión Logisim-evolution): Una herramienta gráfica que permite arrastrar compuertas y cables para construir el circuito.
* Digital (versión en línea): Un simulador web gratuito que facilita el diseño y la prueba de circuitos lógicos.

Además, existen alternativas como CircuitVerse o Logicly (en versión demo), cada una con sus propias ventajas.

###### Instrucciones para el Trabajo Práctico

Definir la Tabla de Verdad:

* El docente proporcionará una tabla con las combinaciones posibles de las entradas A, B y C, y la correspondiente salida F.

Elaborar la Expresión Booleana:

-Basándose en la tabla de verdad, se debe construir la forma disyuntiva inicial (lista de minterms) sin optimización.

Optimización:

* Utilizar un Mapa de Karnaugh para agrupar las celdas correspondientes a los valores de F iguales a 1.
* Obtener la expresión booleana minimizada a partir de estas agrupaciones.

Implementación en el Simulador:

* Crear los pines de entrada (A, B, C) y la salida (F).
* Agregar las compuertas lógicas necesarias (ya sean AND, OR, NOT o bien únicamente NAND) según la expresión final.
* Conectar y etiquetar los componentes para facilitar la verificación.
* Probar todas las combinaciones de entrada y confirmar que la salida F se comporta de acuerdo con la tabla inicial.

Documento de Entrega:

-Incluir la tabla de verdad, capturas de pantalla del Mapa de Karnaugh y del circuito implementado en el simulador, junto con una breve reflexión sobre la importancia de la optimización en el diseño de circuitos digitales.

## Dispositivos

### Multiplexores (MUX)

¿Qué son?  
Un multiplexor es un componente que selecciona una de varias entradas de datos y la dirige a una única salida.

¿Para qué sirve?  
Actúa como un 'selector de caminos'. Por ejemplo, si tenés varias fuentes de información y necesitás elegir una, el MUX se encarga de eso.

Ejemplo real:  
En una CPU, se usa para seleccionar cuál dato enviar al registro o al bus.

Esquema básico:  
- Entradas: 2ⁿ líneas de datos  
- Selectores: n líneas de control  
- Salida: 1 línea

Tabla de verdad simple (MUX 2:1):

|  |  |  |  |
| --- | --- | --- | --- |
| S | I0 | I1 | Salida |
| 0 | A | B | A |
| 1 | A | B | B |

### Decodificadores

¿Qué son?  
Son componentes que toman una entrada binaria y activan una única salida correspondiente.

¿Para qué sirve?  
Transforman una señal codificada en varias líneas de salida. Son muy usados para direccionar memorias o activar dispositivos específicos.

Ejemplo real:  
Un decodificador 3 a 8 activa una de 8 salidas, útil para direccionar una celda entre 8 posibles.

### Biestables (Flip-Flops)

¿Qué son?  
Son circuitos capaces de almacenar un bit. Pueden mantener su estado hasta que se lo modifique.

Tipos comunes:  
- RS (Set-Reset)  
- D (Data o Delay)  
- JK  
- T (Toggle)

¿Para qué sirve?  
Son la base de las memorias y registros. Cada flip-flop puede guardar un '1' o un '0'.

Ejemplo real:  
En un registro de 8 bits, se usan 8 flip-flops en paralelo.

### Memorias

¿Qué son?  
Dispositivos capaces de almacenar información digital. Se construyen con miles o millones de flip-flops o celdas más complejas.

Tipos básicos:  
- RAM (Volátil): pierde los datos al apagar.  
- ROM (No volátil): conserva datos, pero no se puede escribir fácilmente.

¿Dónde se usa?  
En todos los sistemas digitales: celulares, notebooks, servidores.

### Microcontroladores

¿Qué son?  
Pequeños computadores integrados en un solo chip. Incluyen procesador, memoria, y periféricos de entrada/salida.

¿Para qué sirve?  
Automatizan tareas específicas. Son el “cerebro” de muchísimos dispositivos.

Ejemplo real:  
Arduino es una plataforma basada en microcontroladores. Se usa para controlar luces, motores, sensores, etc.

¿Por qué es importante este tema para un analista de datos?

Aunque no armes circuitos, saber cómo fluye y se almacena la información a bajo nivel te da:

- Mejor comprensión del rendimiento de los sistemas  
- Capacidad de optimizar recursos  
- Cultura general sólida en tecnología

## Diseño a Mayor Escala: Módulos y Jerarquía

En sistemas digitales complejos, el diseño se organiza en módulos:

* Módulos combinacionales: Tales como sumadores, comparadores o multiplexores, que se diseñan con funciones booleanas optimizadas.
* Módulos secuenciales: Incluyen registros, contadores y máquinas de estado, que combinan la lógica combinacional con elementos de memoria (como flip-flops).
* Estos módulos se integran en un diagrama a nivel superior (top-level) para describir cómo interactúan entre sí.

En proyectos grandes, se suelen emplear lenguajes de descripción de hardware (HDL) como VHDL o Verilog, que permiten generar de forma automática la síntesis lógica y la optimización del circuito.

Relación con la Computación y la Ciencia de Datos

Aunque en la ciencia de datos el enfoque principal no es el nivel de hardware, es fundamental entender que la eficiencia y fiabilidad de un procesador dependen de la correcta implementación de circuitos digitales.

Por ejemplo:

* Aceleradores de IA: Dispositivos como las Tensor Processing Units (TPU) o las GPUs están diseñados con circuitos digitales optimizados para operaciones matemáticas intensivas, como la multiplicación de matrices.
* Paralelismo en software: Los principios de optimización y reducción de redundancias en circuitos tienen su análogo en el diseño de algoritmos y estructuras de datos en software.

Comprender estos conceptos permite apreciar la estrecha relación entre el diseño digital y el rendimiento de sistemas de cómputo modernos.

## Conclusiones

El diseño y la optimización de circuitos digitales representan el puente entre la teoría abstracta de la lógica booleana y su materialización en hardware. Los métodos de minimización, como el uso de Mapas de Karnaugh o el algoritmo de Quine-McCluskey, buscan reducir la cantidad de compuertas o la complejidad del circuito, lo que tiene un impacto directo en los costos, el consumo energético y el rendimiento.

La posibilidad de emplear compuertas universales, como NAND o NOR, y el diseño modular a través de componentes como multiplexores o decodificadores, sientan las bases para la creación de sistemas digitales de gran envergadura, como memorias, unidades aritméticas lógicas (ALU) y microcontroladores. En última instancia, estos conceptos son fundamentales para entender la infraestructura que sustenta la computación moderna.

Al realizar el trabajo práctico utilizando simuladores gratuitos, no solo se comprende la transformación de una expresión lógica en un circuito tangible, sino que también refuerzan el vínculo entre la teoría y la práctica en el ámbito de la lógica digital.