2. Introducción a los Circuitos lógicos

angel@uc.cl

Los dibujos, fotos, esquemáticos e imágenes incluidos en estas láminas vienen del sitio web del libro del curso, con permiso del autor.

IEE2713 Sistemas Digitales



2.1 – 2.4.1

Variables, funciones y compuertas

angel@uc.cl

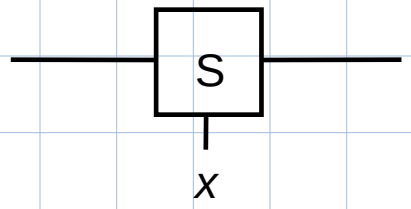
IEE2713 Sistemas Digitales

2.1 Variables y funciones

- Una variable binaria x puede adoptar 2 valores: 0 o 1
- Variables binarias pueden actuar sobre interruptores



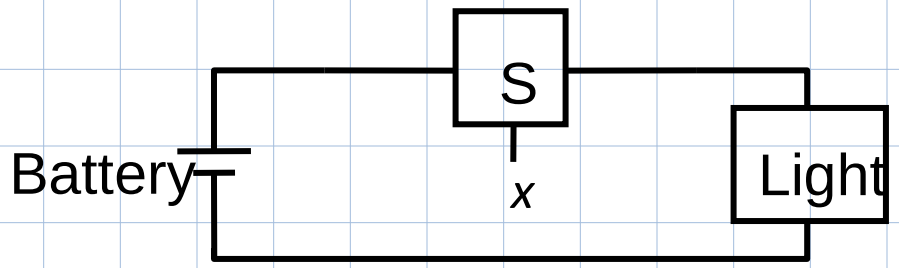
(a) Two states of a switch



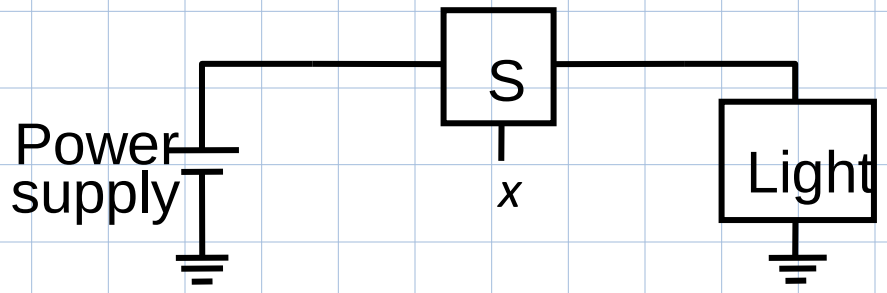
(b) Symbol for a switch

Cerrado = deja pasar la señal
Abierto = no deja pasar la señal

Interruptores comandados por variables binarias afectan otras partes del circuito



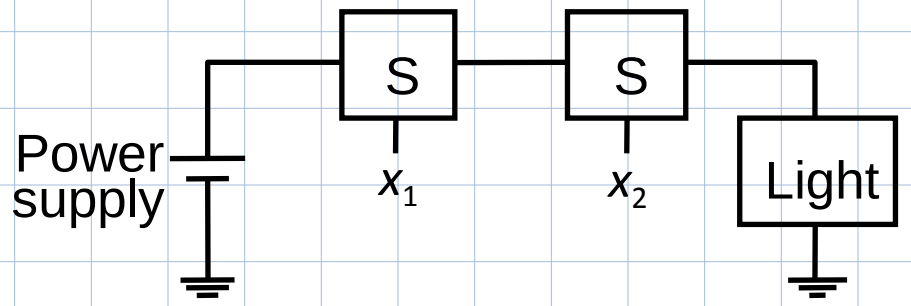
(a) Simple connection to a battery



(b) Using a ground connection as the return path

- El estado L de la luz es función de la variable binaria x
  - Nos referimos a L(x)
- En este caso, L(x) = x
  - L(x) es una función lógica
  - x es una variable de entrada

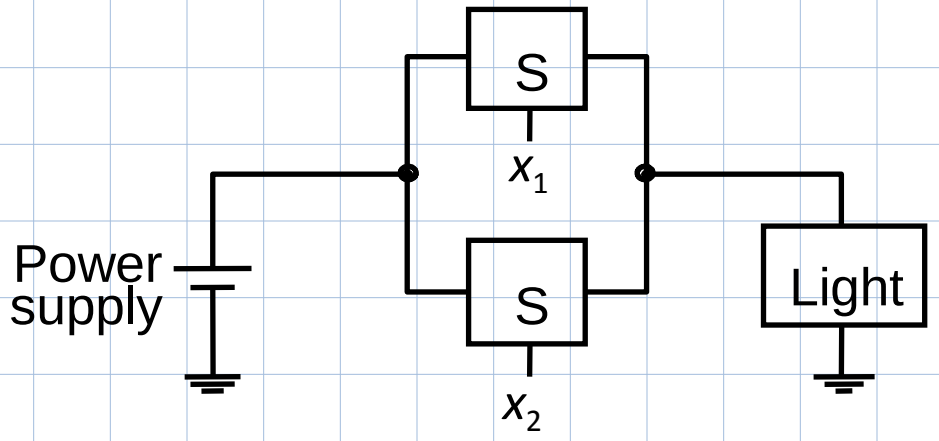
Conexión de los switches en serie



(a) The logical AND function (series connection)

- En serie: sólo enciende la luz si **ambos** switches están cerrados
  - L(x1, x2) = x1 · x2
  - L = 1 sólo si x1 = 1 y x2 = 1
    - Cualquier otro caso: L = 0
- El símbolo · corresponde al operador AND
- La figura implementa la **función lógica AND**

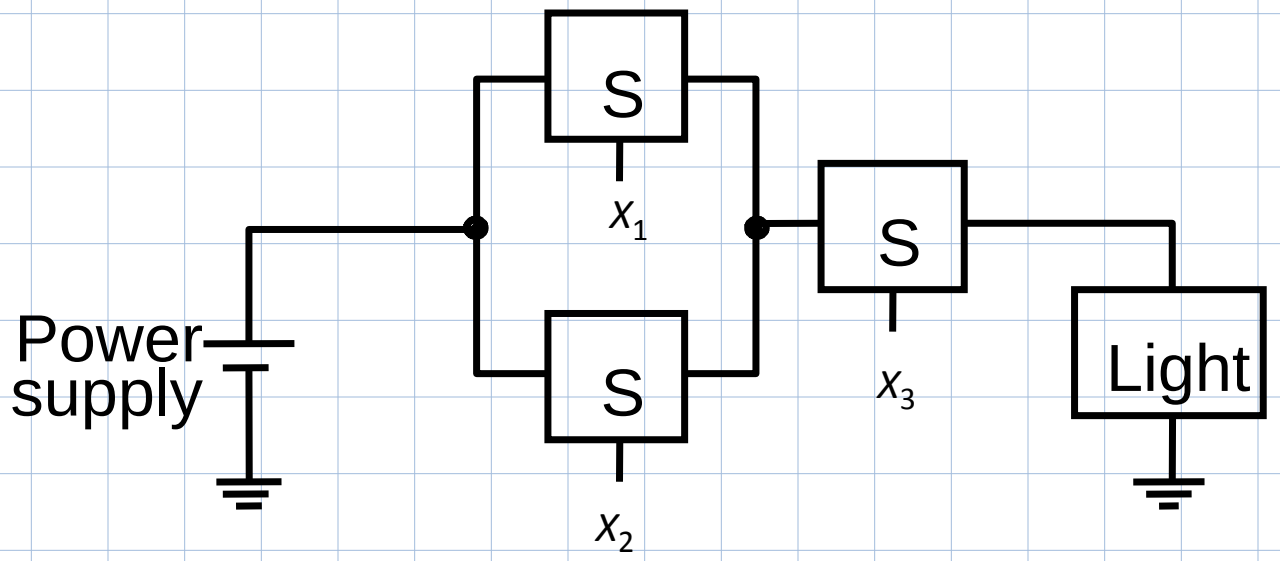
Conexión de los switches en paralelo



(b) The logical OR function (parallel connection)

- En paralelo: enciende la luz si **cualquiera** de los switches está cerrado
  - L(x1, x2) = x1 + x2
  - L = 1 si x1 = 1 o si x2 = 1
    - Cualquier otro caso: L = 0
- El símbolo + corresponde al operador OR
- La figura implementa la **función lógica OR**

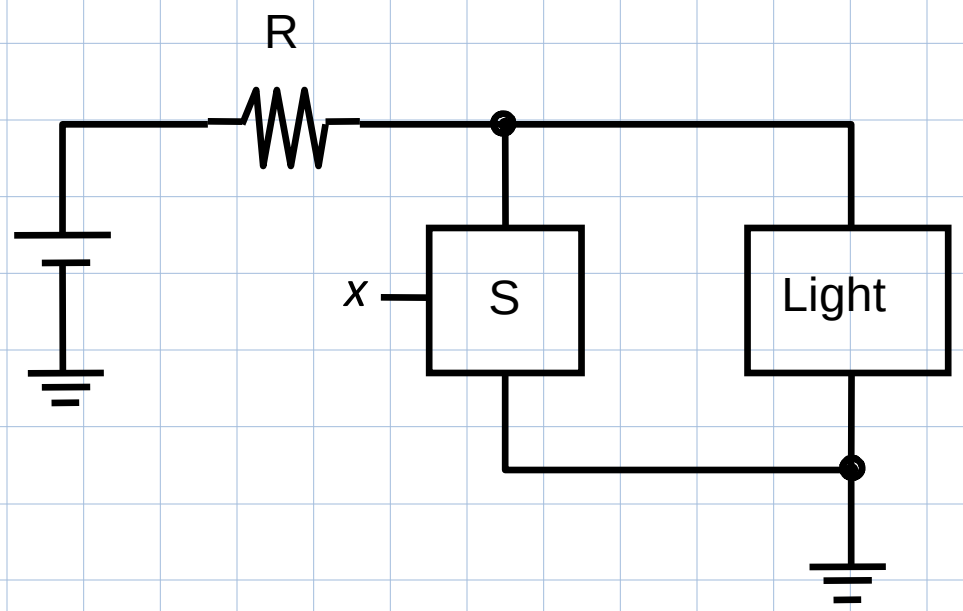
Es posible combinar funciones lógicas



- En este caso L(x1, x2, x3) = (x1+x2) · x3
- **¿Qué condición tiene que ocurrir para que la luz esté encendida?**

2.2 Inversión

- En el circuito de la figura
  - si x = 0 entonces L = 1
  - si x = 1 entonces L = 0
- Entonces podemos afirmar que L(x) = $\overline{x}$
- Alternativamente, $\overline{x}$ = ~x = `x = !x
- Los símbolos barra, ~, ` y ! corresponden al operador inversor
- L(x) corresponde a la función **inversión lógica**



2.3 Tablas de verdad

- Describen funciones lógicas en forma exhaustiva
  - Especifican las salidas para todas las combinaciones posibles de entradas
- Ejemplo: tablas de verdad para funciones AND y OR de dos variables
- Número de líneas crece exponencialmente con el número n de entradas (2^n)

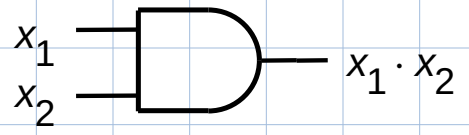
x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

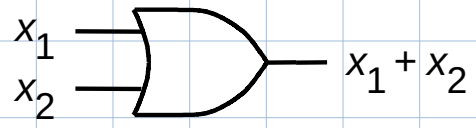
Ejemplo: OR y AND de 3 entradas

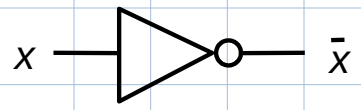
x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

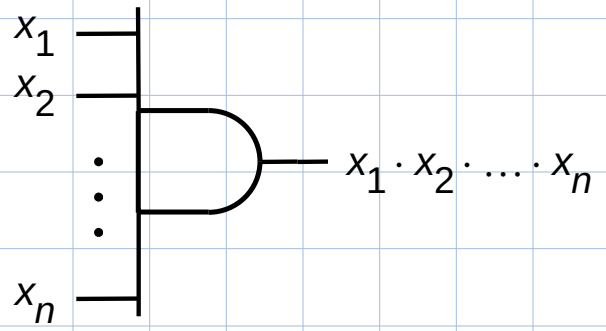
2.4 Compuertas lógicas y redes

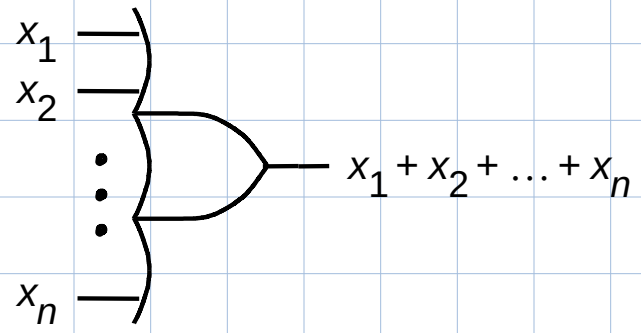
- Las compuertas lógicas son componentes circuitales que implementan funciones lógicas
  - Están implementadas mediante transistores
  - Requieren alimentación y tienen entradas y salidas





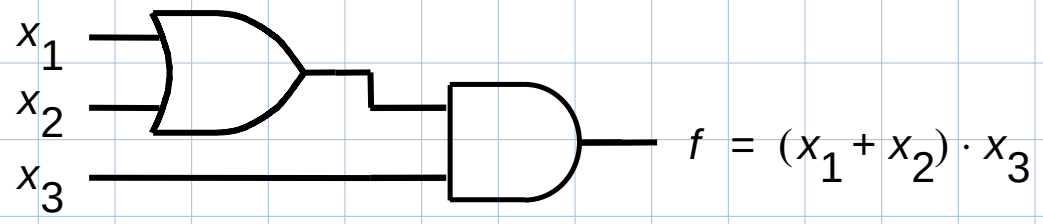






Esquemático

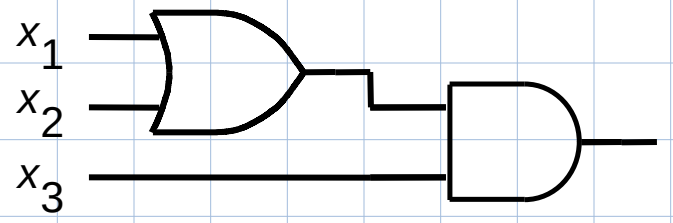
Una forma conveniente de describir un circuito digital es mediante un **esquemático** que muestra conexiones entre compuertas lógicas



Los esquemáticos pueden ser arbitrariamente complejos

2.4.1 Análisis de una red lógica

Tratemos de entender cómo funciona este circuito

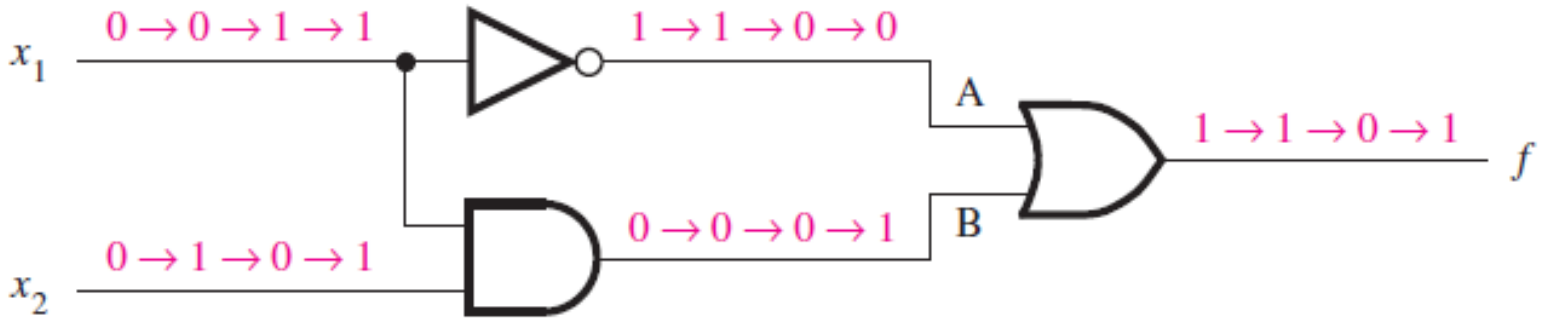


$$f = (x_1 + x_2) \cdot x_3$$

Una opción es probar todas las combinaciones de entradas y ver qué ocurre en la salida…

Analizar: determinar el funcionamiento de una red
Sintetizar: diseñar la red para una especificación dada

Procedimiento de análisis



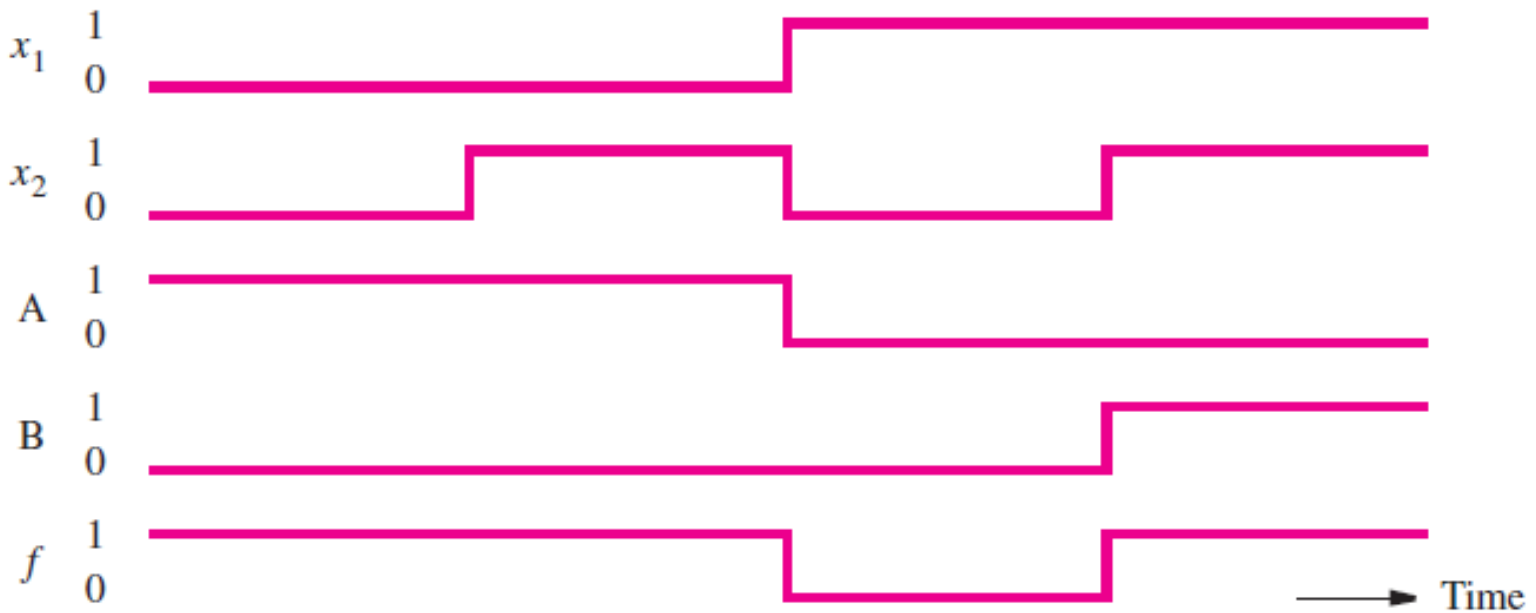
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

(b) Truth table

Diagrama de tiempo (timing diagram)

- Representación gráfica de entradas y salidas
- El tiempo avanza de izquierda a derecha (igual que en osciloscopio y analizador lógico)



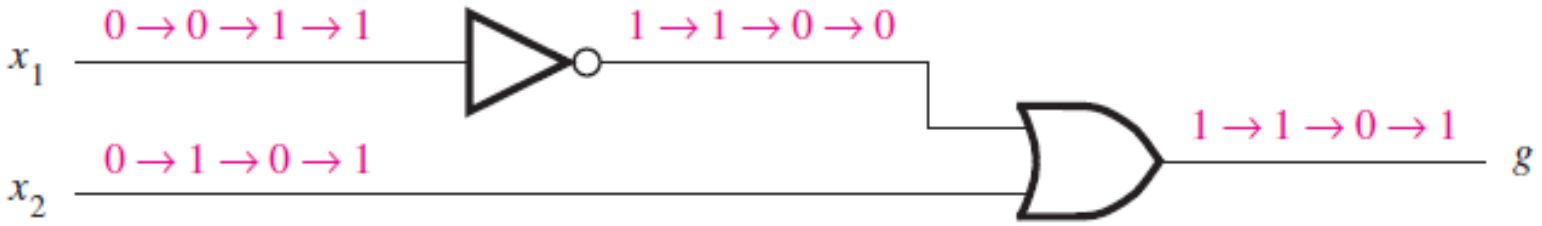
(c) Timing diagram

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

(b) Truth table

Redes funcionalmente equivalentes

- Analicemos este otro circuito:



(d) Network that implements $g = \bar{x}_1 + x_2$

- Escribamos la tabla de verdad correspondiente a su función lógica y comparemos con lámina anterior

x_1	x_2	$g(x_1, x_2)$
0	0	
0	1	
1	0	
1	1	

¿Conclusiones?



2.5 – 2.8.2

Álgebra Booleana

angel@uc.cl

IEE2713 Sistemas Digitales

2.5 Álgebra Booleana

Axiomas

1*a*.	$0 \cdot 0 = 0$
1*b*.	$1 + 1 = 1$
2*a*.	$1 \cdot 1 = 1$
2*b*.	$0 + 0 = 0$
3*a*.	$0 \cdot 1 = 1 \cdot 0 = 0$
3*b*.	$1 + 0 = 0 + 1 = 1$
4*a*.	If $x = 0$, then $\overline{x} = 1$
4*b*.	If $x = 1$, then $\overline{x} = 0$

Teoremas de una variable

5*a*.	$x \cdot 0 = 0$
5*b*.	$x + 1 = 1$
6*a*.	$x \cdot 1 = x$
6*b*.	$x + 0 = x$
7*a*.	$x \cdot x = x$
7*b*.	$x + x = x$
8*a*.	$x \cdot \overline{x} = 0$
8*b*.	$x + \overline{x} = 1$
9.	$\overline{\overline{x}} = x$

Álgebra Booleana - dualidad

- Para una expresión lógica, su dual es obtenida reemplazando todos los operadores + por ·, y vice versa, y reemplazando todos los 0s por 1s y vice versa
  - ¿Ejemplos?

- El dual de cualquier declaración verdadera – axioma o teorema – en álgebra Booleana, es también verdadera
  - ¿Ejemplos?

Álgebra Booleana – propiedades de dos o tres variables

10*a*.	$x \cdot y = y \cdot x$	*Commutative*
10*b*.	$x + y = y + x$	
11*a*.	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	*Associative*
11*b*.	$x + (y + z) = (x + y) + z$	
12*a*.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributive*
12*b*.	$x + y \cdot z = (x + y) \cdot (x + z)$	
13*a*.	$x + x \cdot y = x$	*Absorption*
13*b*.	$x \cdot (x + y) = x$	
14*a*.	$x \cdot y + x \cdot \bar{y} = x$	*Combining*
14*b*.	$(x + y) \cdot (x + \bar{y}) = x$	
15*a*.	$\overline{x \cdot y} = \bar{x} + \bar{y}$	*DeMorgan's theorem*
15*b*.	$\overline{x + y} = \bar{x} \cdot \bar{y}$	
16*a*.	$x + \bar{x} \cdot y = x + y$	
16*b*.	$x \cdot (\bar{x} + y) = x \cdot y$	

Es posible demostrar todos estos teoremas y propiedades mediante:

- **Inducción perfecta** – mirando todos los casos posibles
- **Manipulación algebraica** en base a axiomas

En teoría, basta con teoremas 5 y 8, y con propiedades 10 y 12 (postulados básicos de Huntington), para derivar todas las demás propiedades

Ejemplo 2.1

Demuestre que $(x_1 + x_3) \cdot (\overline{x_1} + \overline{x_3}) = x_1 \cdot \overline{x_3} + \overline{x_1} \cdot x_3$

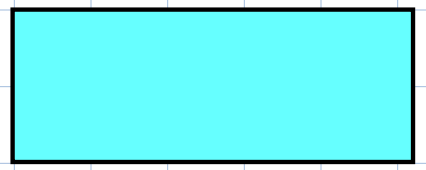
Ejemplo 2.2

Demuestre que

$$x_1 \cdot \overline{x_3} + \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_3 + \overline{x_2} \cdot x_3 = \overline{x_1} \cdot \overline{x_2} + x_1 \cdot x_2 + x_1 \cdot \overline{x_2}$$

2.5.1 Diagrama de Venn

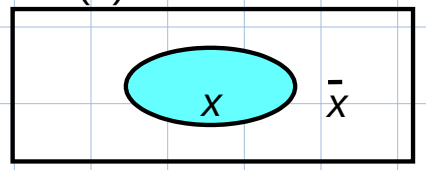
Pueden ser utilizados para verificar de manera intuitiva la equivalencia entre dos expresiones lógicas



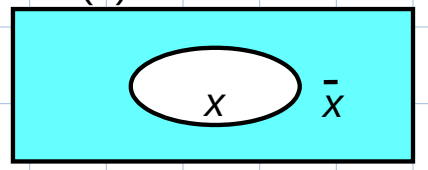
(a) Constant 1



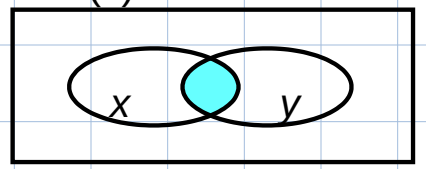
(b) Constant 0



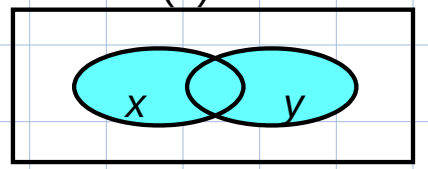
(c) Variable x



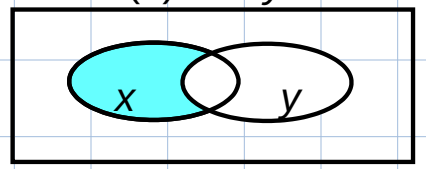
(d) $\bar{x}$



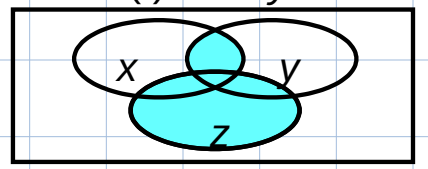
(e) $x \cdot y$



(f) $x + y$



(g) $x \cdot \bar{y}$



(h) $x \cdot y + z$

2.5.2 Notación y terminología

- Por su similaridad con las operaciones aritméticas suma y multiplicación, las operaciones OR y AND son llamadas coloquialmente *suma lógica* y *multiplicación lógica*
  - Ejemplo: suma de productos

$$x_1 \cdot \bar{x}_2 \cdot x_3 + \bar{x}_1 \cdot x_4 + x_2 \cdot x_3 \cdot \bar{x}_4$$

  - Ejemplo: producto de sumas

$$(\bar{x}_1 + x_3) \cdot (x_1 + \bar{x}_3) \cdot (\bar{x}_2 + x_3 + x_4)$$

- Para evitar ambigüedad entre operaciones lógicas y aritméticas, a veces empleamos los símbolos ˅ y ^ en lugar de OR y AND

2.5.3 Precedencia de operaciones

- Es posible construir cualquier expresión lógica combinando OR, AND y NOT
- Los paréntesis evitan la ambigüedad en la precedencia
- Para evitar llenarnos de paréntesis podemos definir un orden de precedencia: NOT, luego AND y finalmente OR
- Para simplificar las expresiones lógicas, generalmente se omite el operador ·
  - Ejemplo:

$$x_1 x_2 + \overline{x}_1 \overline{x}_2$$

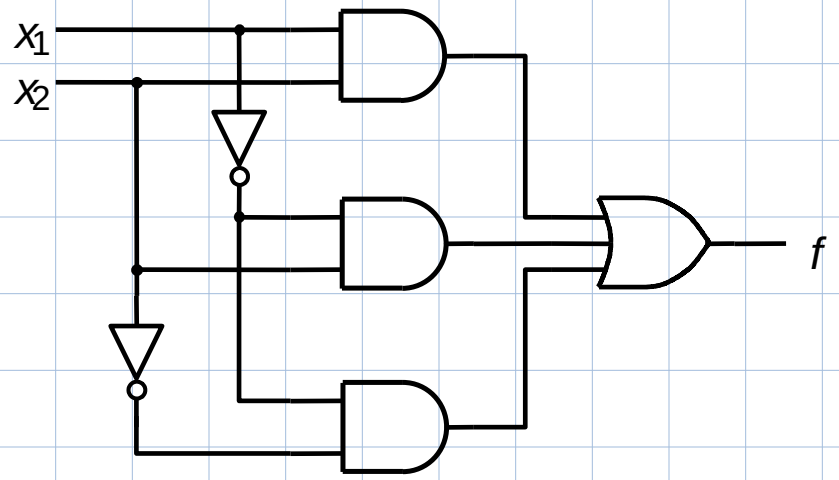
2.6 Síntesis usando compuertas AND, OR y NOT

- Comencemos con un ejemplo: tratemos de sintetizar (crear) un circuito que implemente la tabla de verdad
- Vamos a escribir una función lógica y luego la llevaremos a una implementación con compuertas
- Una opción sería crear funciones lógicas tipo productos que produzcan un 1 lógico cuando se cumple cada condición afirmativa de la tabla de verdad
  - Luego hacemos un OR entre esos términos

$$f(x_1, x_2) =$$

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

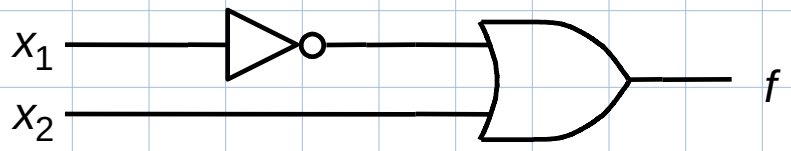
Resultado



(a) Canonical sum-of-products

¿Podremos simplificarlo mediante manipulación algebraica?

Realización de mínimo costo



(b) Minimal-cost realization

- Wow!
- Conclusiones:
  - Es muy fácil (y muy mecánico) implementar cualquier función lógica mediante una suma de productos para cada línea de la tabla de verdad que produzca un 1 lógico
    - Cada producto incluye todas las variables de entrada
  - Hay muchas formas de implementar circuitalmente una misma función lógica
  - Es posible simplificar mediante manipulación algebraica

2.6.1 Sumas de productos y productos de sumas

- Cualquier función lógica f expresada mediante su tabla de verdad puede ser realizada mediante:
  - Una expresión lógica que considere todas las líneas de la tabla en que f = 1
  - O una expresión lógica que considere todas las líneas de la tabla en que f = 0
- Minterm m_i de una función lógica de n variables – definición
  - Es cualquier producto lógico en que todas las variables aparecen una vez, ya sea negada o no negada
  - Para cada fila de la tabla, incluimos x_i si x_i=1, o bien incluimos ~x_i si x_i = 0

Minterm

- Considere una función lógica con n variables de entrada
- Un minterm es cualquier **producto lógico** en que todas las variables aparecen una vez, ya sea negada o no negada
  - Para cada fila de la tabla, incluimos x_i si $x_i = 1$, o bien incluimos $\sim x_i$ si $x_i = 0$
- Ejemplo: minterms para esta tabla

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

- Para simplificar la notación, el minterm de la fila i es designado como m_i

Suma de productos

- Una función lógica f puede ser expresada como una suma de los minterms correspondientes a las filas de la tabla de verdad **que producen f = 1**
- La expresión resultante es correcta y única, pero no necesariamente mínima
  - Esta expresión se denomina **suma de productos canónica para la función f**
- El primer paso de la síntesis de funciones lógicas es llegar a la suma de productos canónica
  - Luego se simplifica con manipulaciones algebraicas

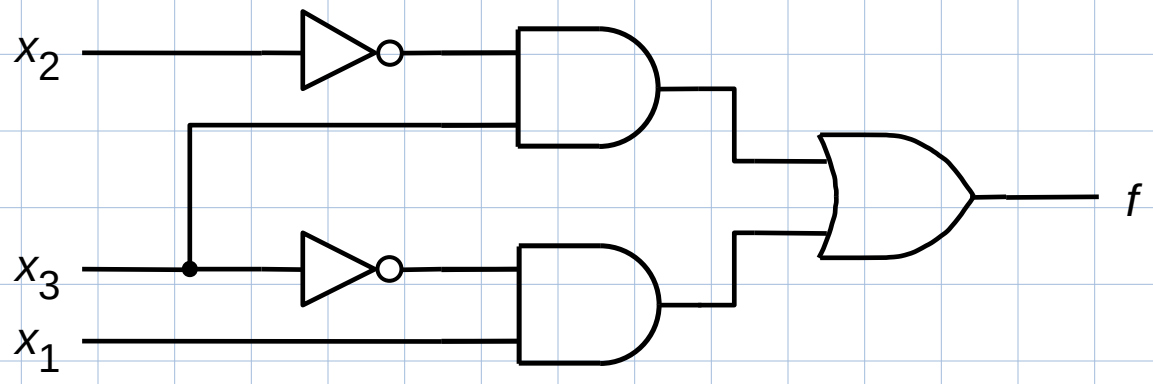
Ejemplo – suma de productos

- Considere la tabla de verdad
- La síntesis incluye los minterms 1, 4, 5 y 6:

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

- Mediante manipulación algebraica llegamos a:

¿Cómo sería la implementación con compuertas lógicas?



Una buena indicación del costo de un circuito lógico es la suma del número de compuertas y el número total de entradas de todas las compuertas – en este caso, 13

- Es posible usar notación de minterms para expresar funciones lógicas en forma abreviada – ejemplo:

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

Maxterm – el dual del minterm

- Considere una función lógica con n variables de entrada
- Un maxterm es cualquier **suma lógica** en que todas las variables aparecen una vez, ya sea negada o no negada
  - Para cada fila de la tabla, incluimos x_i si $x_i = 0$, o bien incluimos $\sim x_i$ si $x_i = 1$
- Ejemplo: maxterms para esta tabla

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

- Para simplificar la notación, el maxterm de la fila i es designado como M_i

Producto de sumas

- Una función lógica f puede ser expresada como un producto de los maxterms correspondientes a las filas de la tabla de verdad **que producen f = 0**
- La expresión resultante es correcta y única, pero no necesariamente mínima
  - Esta expresión se denomina **producto de sumas canónico para la función f**
- El primer paso de la síntesis de funciones lógicas puede ser expresar un producto de sumas canónico
  - Luego se simplifica con manipulaciones algebraicas

~$m_i = M_i$

Esto es útil – por ejemplo, para la tabla de verdad:

$$\overline{f}(x_1, x_2, x_3) = m_0 + m_2 + m_3 + m_7$$

$$\overline{f}(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1 x_2\overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1 x_2 x_3$$

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Negando la expresión, obtenemos:

$$f(x_1, x_2, x_3) = \overline{m_0 + m_2 + m_3 + m_7} = \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7$$

Sabiendo que ~$m_i = M_i$, entonces

$$f(x_1, x_2, x_3) = M_0 \cdot M_2 \cdot M_3 \cdot M_7 = (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

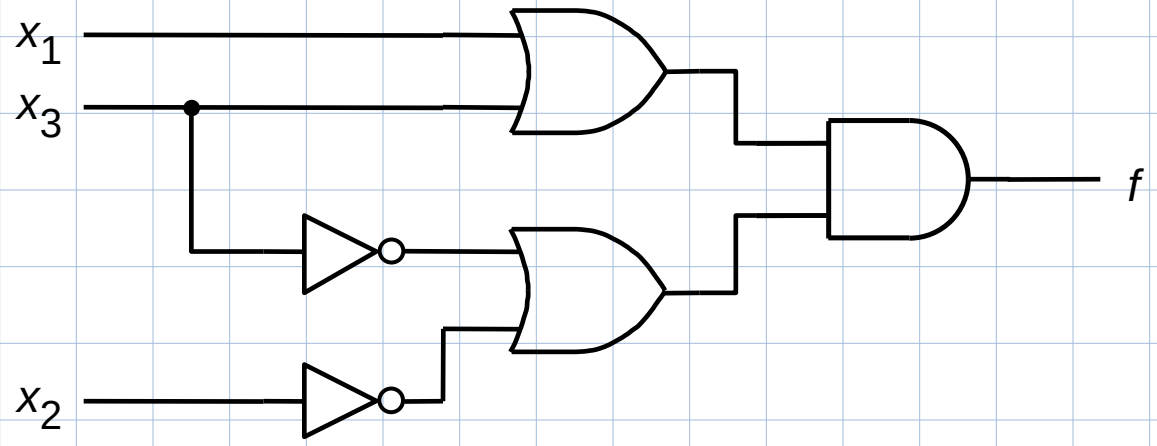
Ejemplo – producto de sumas

- Considere la tabla de verdad
- La síntesis incluye los maxterms 0, 1, 3 y 7:

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

- Mediante manipulación algebraica llegamos a:

¿Cómo sería la implementación con compuertas lógicas?



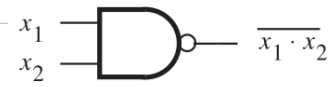
- Es posible usar notación de maxterms para expresar funciones lógicas en forma abreviada – ejemplo:

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

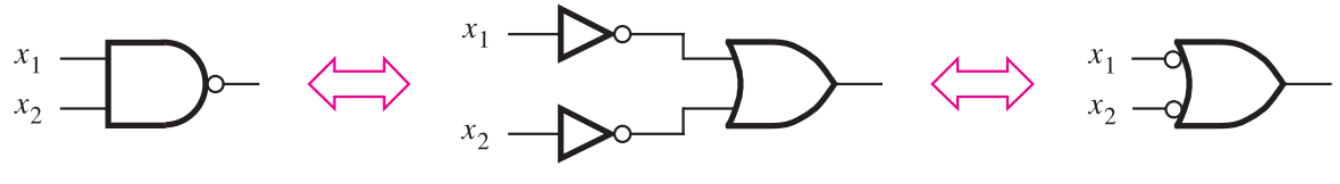
2.7 Circuitos lógicos con NAND y NOR

- NANDs y NORs son compuertas más simples de implementar que ANDs y ORs

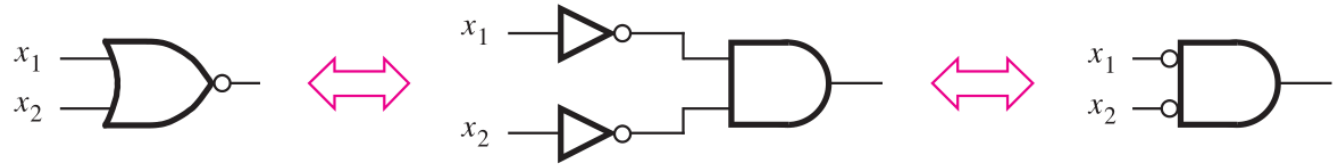




- Las leyes de DeMorgan nos indican que:



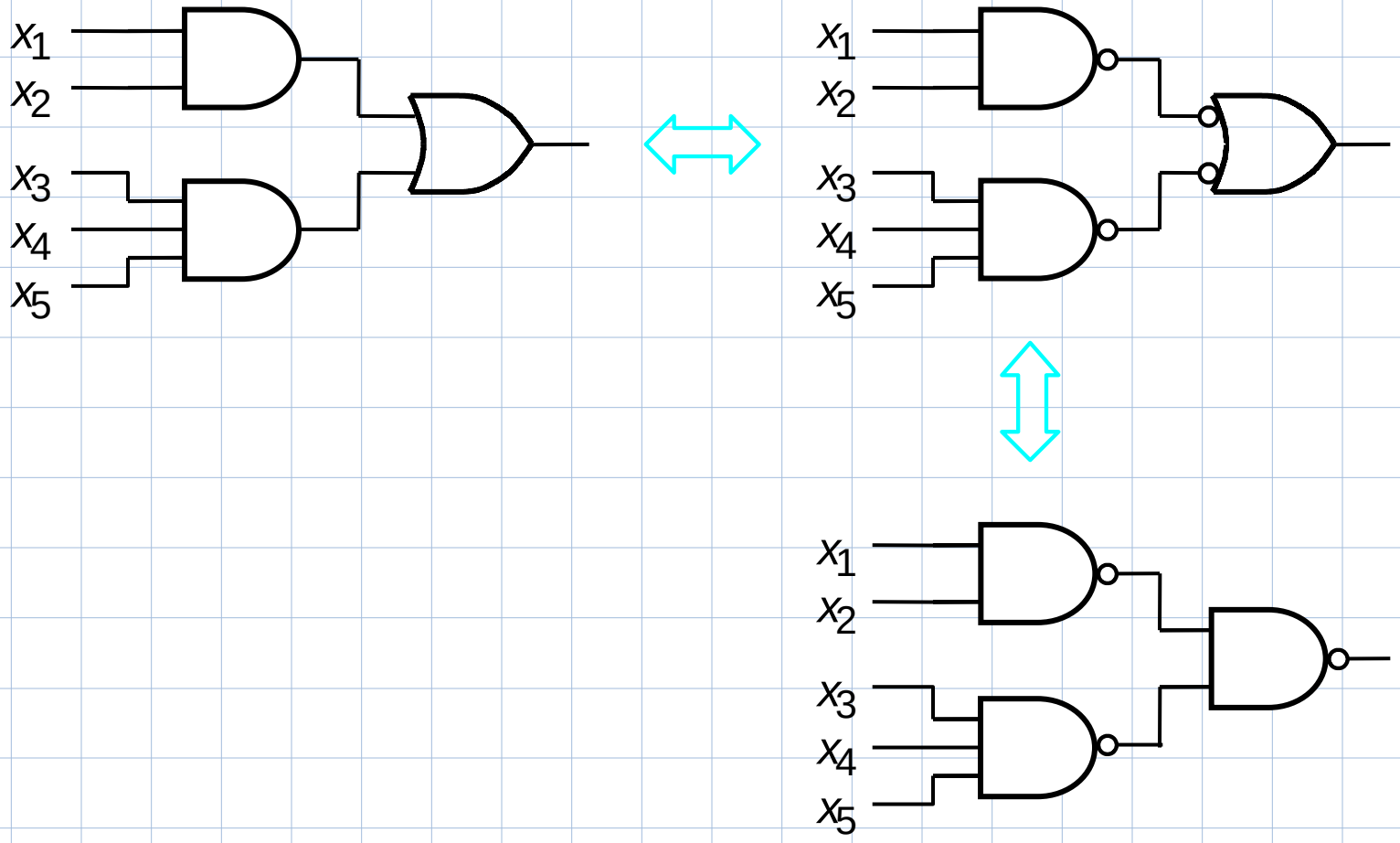
(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$



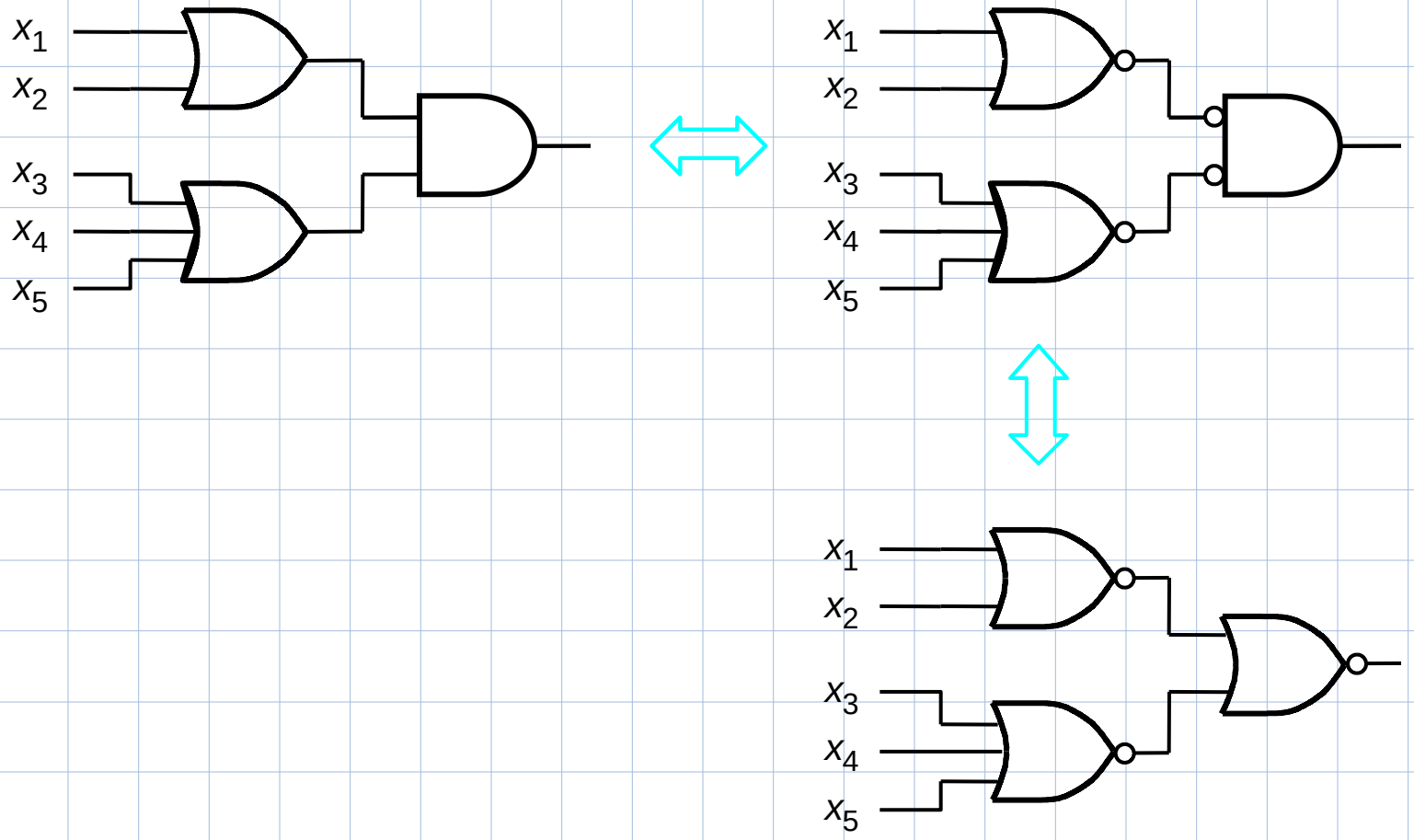
(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Aplicando esto, ¿cómo nos quedan las sumas de productos y los productos de sumas?

Sumas de productos con NANDs



Productos de sumas con NORs



2.8 Ejemplos de diseño

- En general usaremos los siguientes pasos:
  - Especificación de la función del circuito
  - Descripción mediante tablas de verdad o funciones lógicas
  - Síntesis e implementación
  - Verificación

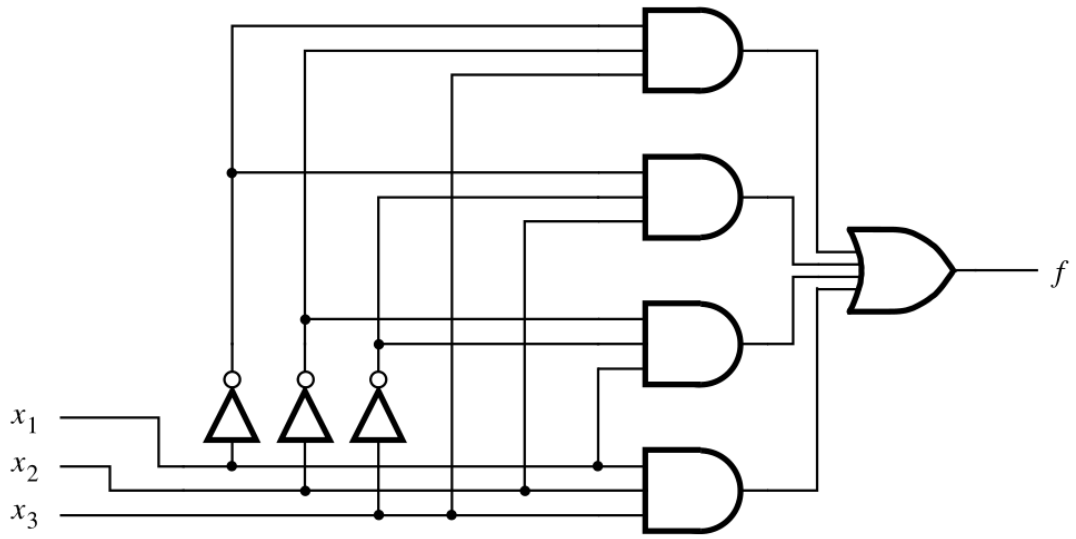
2.8.1 Control de luces

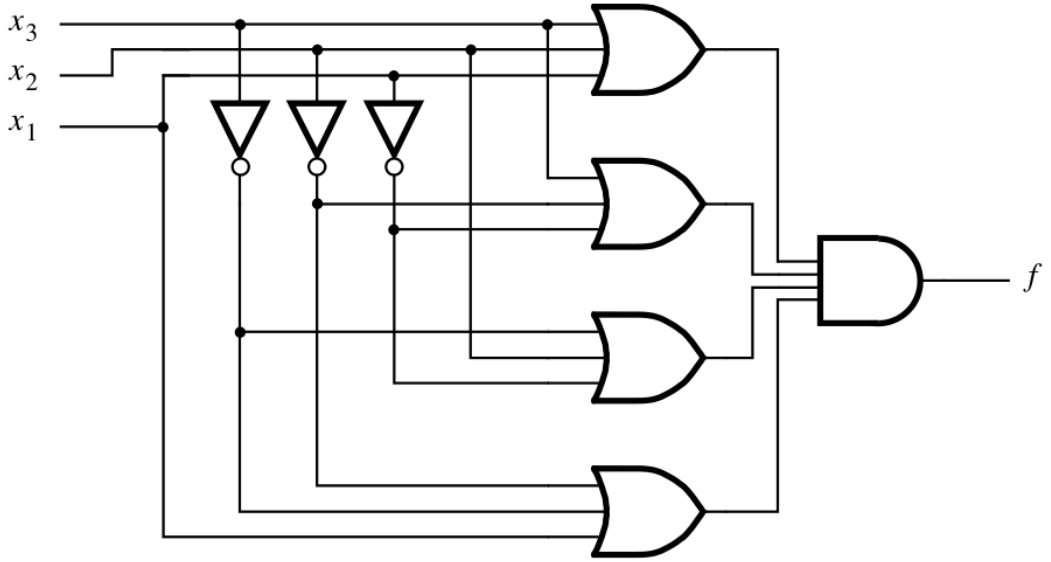
- Supongamos una habitación con tres puertas con un interruptor cerca de cada puerta
- Queremos diseñar un circuito que nos permita cambiar el estado de las luces (encendido o apagado) cambiando el estado de cualquiera de los interruptores
- La tabla de verdad parece cumplir el objetivo…

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Implementaciones

- Ambas implementaciones son equivalentes y tienen igual costo
- También podríamos realizarlas con NANDs o NORs
- Podemos probarlas “a mano” o usando un simulador

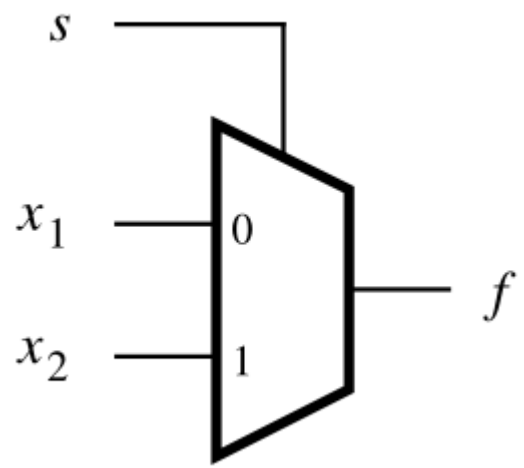




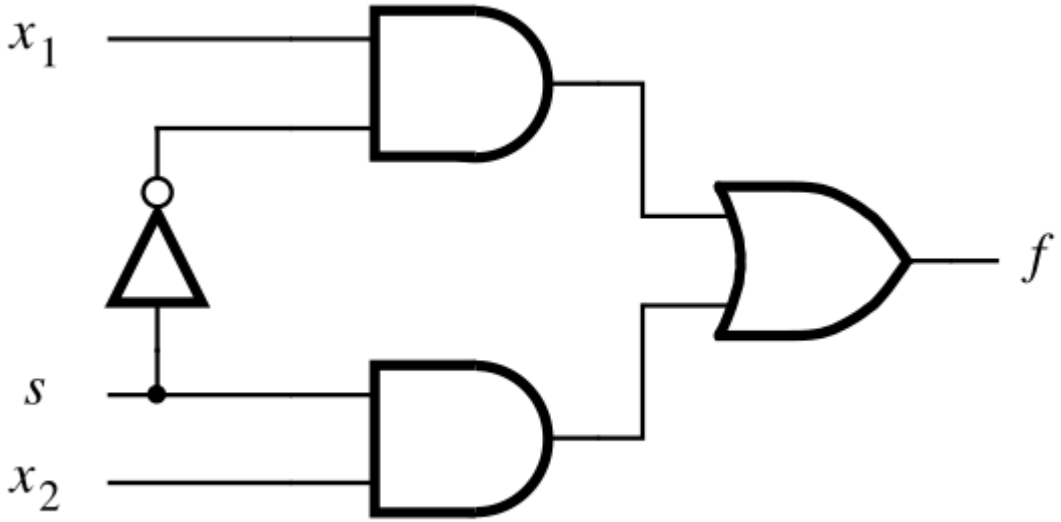
2.8.2 Circuitos multiplexores

- Un multiplexor es un circuito capaz de conectar cualquiera de sus entradas x_i a la salida f
  - "Multiplexa" sus entradas en una única salida
- La elección de la entrada es comandada mediante un selector s
- Los multiplexores son muy usados en sistemas digitales
- Intentemos diseñar un multiplexor de 2 entradas

$s\ x_1 x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1



Implementación



s	$f(s, x_1, x_2)$
0	x_1
1	x_2

- Este circuito también puede ser implementado mediante NANDs
- Nuevamente podemos probar “a mano” o con un simulador

Esta tabla de verdad más compacta describe el comportamiento del multiplexor



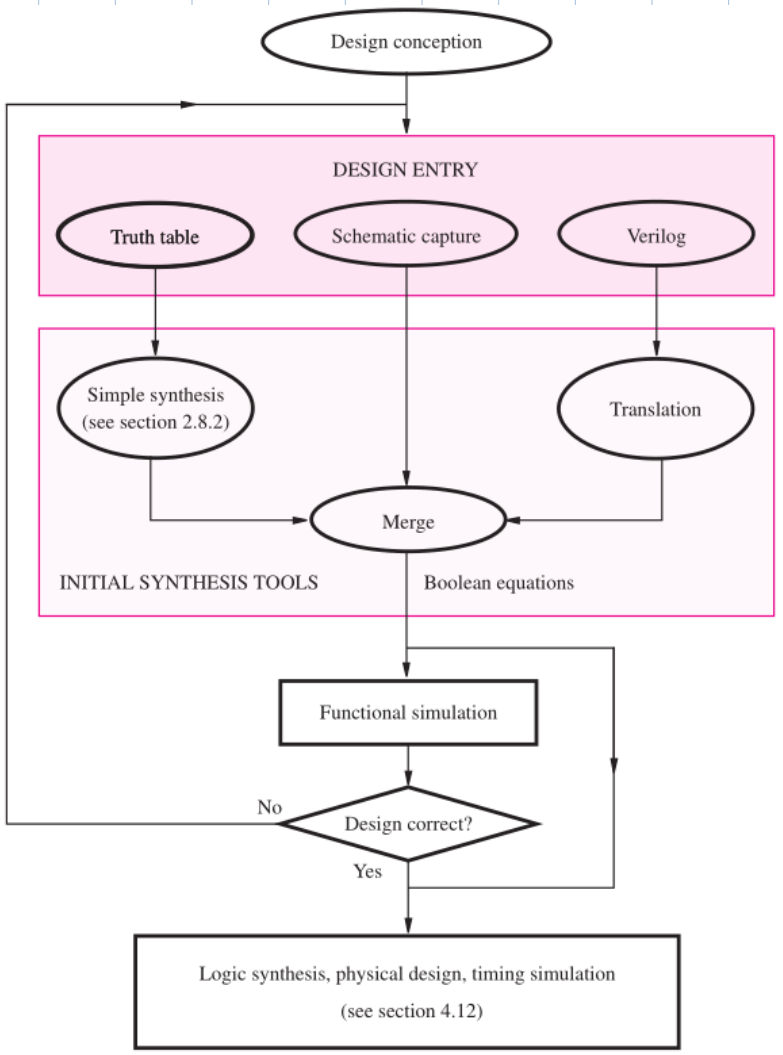
2.9 – 2.11

Introducción a las herramientas de CAD

angel@uc.cl

IEE2713 Sistemas Digitales

2.9 Introducción a las herramientas de diseño asistido por computador (CAD)



- Dependen del tipo de implementación
  - ASIC, FPGA, CPLD, TTL, CMOS, etc.
- En general tienen pasos en común:
  - Ingreso del diseño
  - Síntesis y optimización
  - Simulación funcional
- El flujo de diseño está amarrado a la tecnología objetivo

2.9.1 Ingreso del diseño

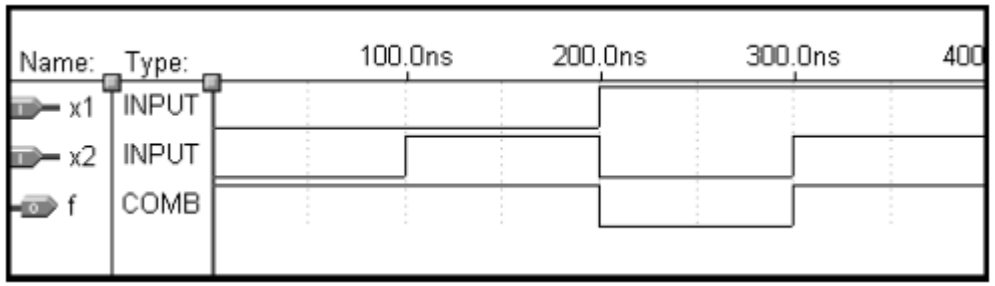
- Hay varias formas de hacerlo:
  - Tablas de verdad o diagrama de tiempo
    - Simple
    - Práctico para diseños pequeños solamente
  - Capturador de esquemático
    - Intuitivo
    - Extenso uso de jerarquía – crucial para diseñar circuitos grandes
  - Lenguaje de descripción de hardware (HDL)
    - Útil para circuitos complejos
    - Portable
    - Jerarquía es crucial
    - Muchas opciones, las más comunes son Verilog y VHDL
- Suele incluir simulaciones de comportamiento

2.9.2 Síntesis

- Muchas opciones:
  - Generación de un circuito a partir de una tabla de verdad
  - Compilación o traducción de una descripción en Verilog en un circuito con compuertas lógicas
  - Optimización lógica (también llamado síntesis lógica)
- La síntesis puede finalizar con la implementación:
  - Mapeo a la tecnología
  - Síntesis del layout o diseño físico

2.9.3 Simulación funcional

- Utiliza el diseño generado por la síntesis
- Requiere un vector de estímulos ingresado por el diseñador
- No toma en cuenta la temporización de las señales
- Suele entregar resultados en formato de texto o como diagramas de tiempo

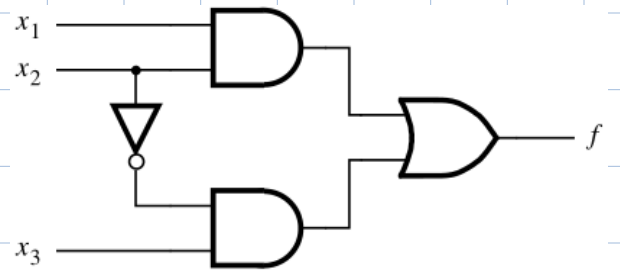


2.10 Introducción a Verilog

- Fue producido en los 80's
- De dominio público desde 1990
- Originalmente diseñado para simulación y verificación
  - Luego se le agregaron capacidades de síntesis
- Verilog es muy complicado
  - Pero es posible hacer muchísimo con lo básico
- Verilog es texto plano y tiene un parecido con el lenguaje C
- Trataremos de aprender Verilog en paralelo al material del curso y lo usaremos para describir circuitos digitales
- Hay 2 estilos principales para describir hardware:
  - Estructural
  - De comportamiento

2.10.1 Especificación estructural de circuitos lógicos

- Especificamos directamente la estructura del circuito – conexiones entre módulos
- Podemos usar las primitivas de Verilog, por ejemplo, **and**, **not**, **or**, **nand** y **nor**
- Los circuitos son modulares (descritos en módulos)
- Cada módulo tiene ports de entrada y salida
  - Son declarados al principio
- El módulo finaliza con endmodule
- Podemos **instanciar** este mismo módulo muchas veces en un sistema digital jerárquico

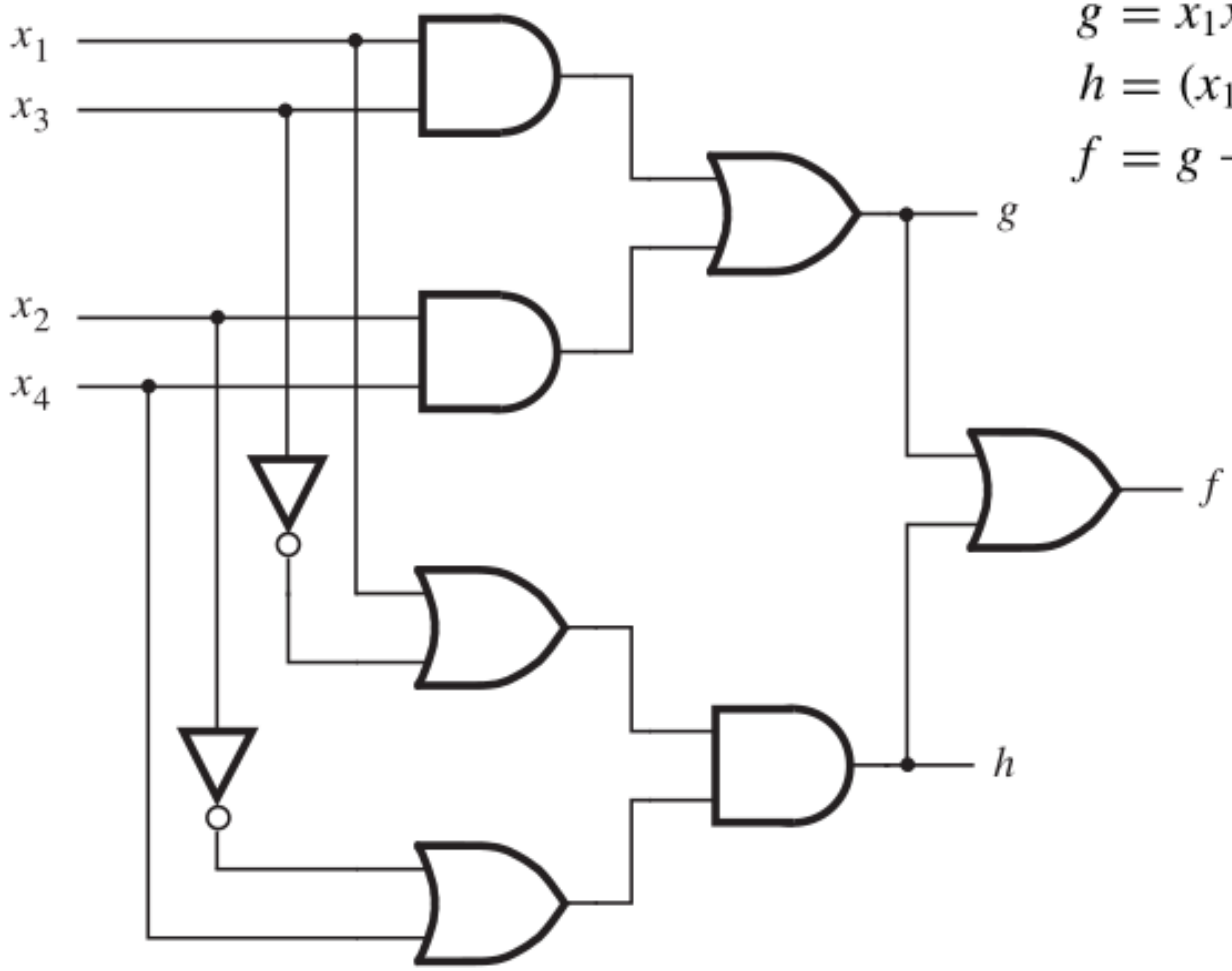



```
module example1 (x1, x2, x3, f);
    input  x1, x2, x3;
    output  f;

    and (g, x1, x2);
    not (k, x2);
    and (h, k, x3);
    or (f, g, h);

endmodule
```

Ejemplo código Verilog



$$g = x_1x_3 + x_2x_4$$
$$h = (x_1 + \bar{x}_3)(\bar{x}_2 + x_4)$$
$$f = g + h$$

```
module example2 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    and (z1, x1, x3);
    and (z2, x2, x4);
    or (g, z1, z2);
    or (z3, x1, ~x3);
    or (z4, ~x2, x4);
    and (h, z3, z4);
    or (f, g, h);

endmodule
```

2.10.2 Especificación de comportamiento de circuitos lógicos

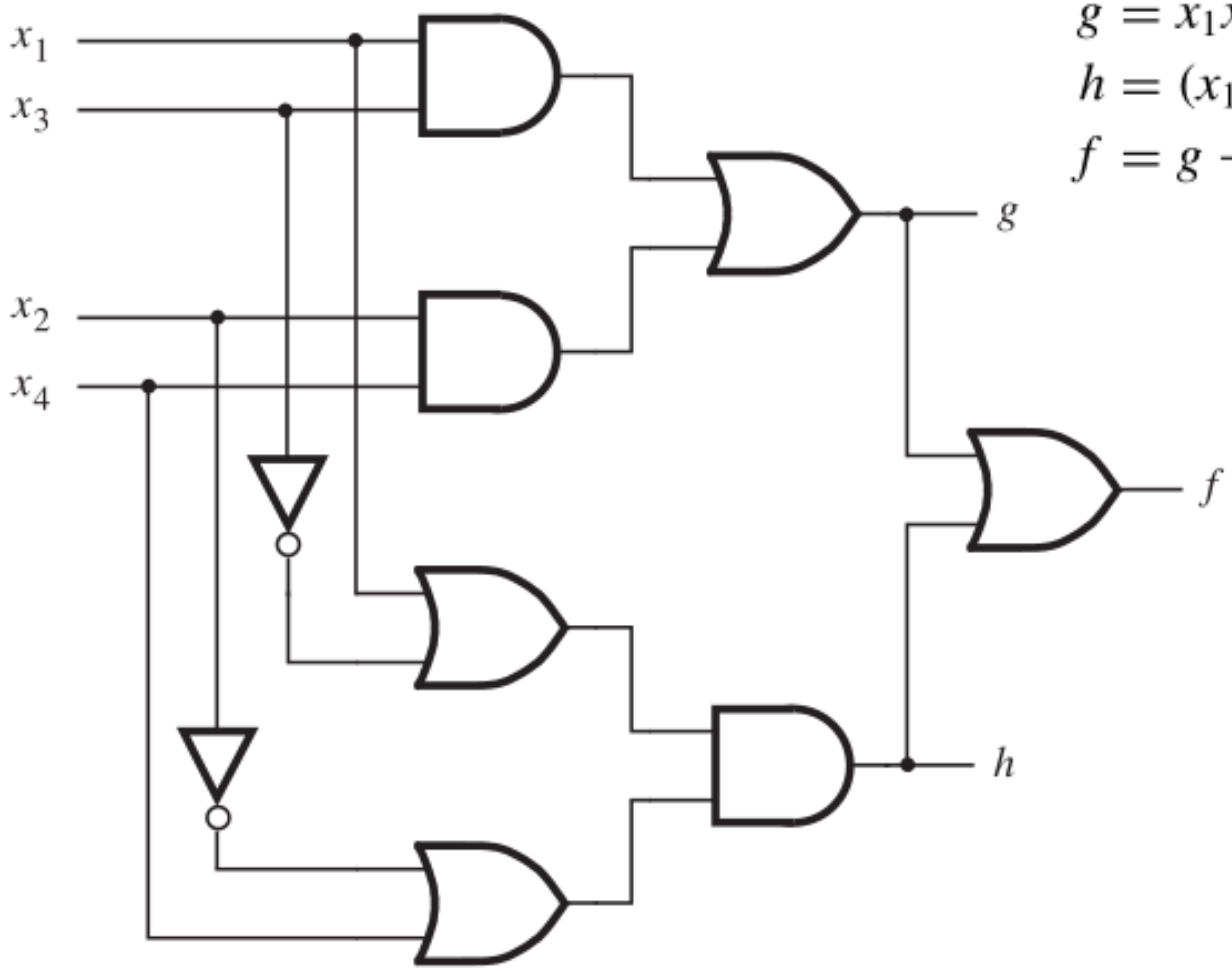
- Describimos el comportamiento que queremos que tenga el circuito
  - El compilador se encarga de definir la estructura
- Usamos **&** para AND, **|** para OR, y **~** para NOT
- Ejemplo: implementemos $f = x_1x_2 + \bar{x}_2x_3$
- La palabra assign realiza una asignación continua y **concurrente** de la salida
  - Como una conexión interna permanente
- La siguiente lámina muestra el ejemplo de la lámina anterior

```
module example3 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;

    assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

Ejemplo código Verilog



$$g = x_1x_3 + x_2x_4$$
$$h = (x_1 + \bar{x}_3)(\bar{x}_2 + x_4)$$
$$f = g + h$$

```
module example4 (x1, x2, x3, x4, f, g, h);
    input x1, x2, x3, x4;
    output f, g, h;

    assign g = (x1 & x3) | (x2 & x4);
    assign h = (x1 | ~x3) & (~x2 | x4);
    assign f = g | h;

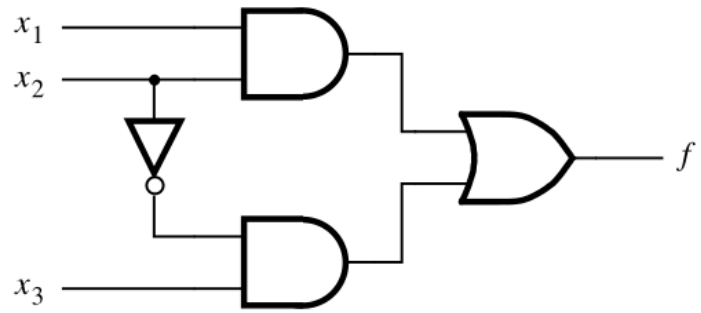
endmodule
```

Especificación a más alto nivel

- Miremos nuevamente el circuito multiplexor:

$$f = g + h$$
$$= x_1 x_2 + \bar{x}_2 x_3$$

- Descrito con palabras, la salida es x_1 cuando x_2=1, o x_3 cuando x_2=0
- Podemos entonces usar la declaración **if-else**




```
// Behavioral specification
module  example5 (x1, x2, x3, f);
    input  x1, x2, x3;
    output f;
    reg f;

    always @(x1 or x2 or x3)
        if (x2 == 1)
            f = x1;
        else
            f = x3;

endmodule
```

Más sobre el ejemplo anterior

- La declaración if-else es un procedimiento
  - Lo veremos con más detalle más adelante
- Los procedimientos van en un bloque **always**
  - Sus contenidos son evaluados en orden
  - Se activa con la **lista de sensibilidad** @(...)
    - Cualquier **cambio** en la lista re-evalúa el bloque always
- La salida del procedimiento es una **variable**
  - La declaramos con la palabra **reg** – detalles cap 5
  - Se asocia a un espacio en memoria
- El CAD compila y diseña el circuito real

```
// Behavioral specification
module  example5 (x1, x2, x3, f);
    input  x1, x2, x3;
    output f;
    reg f;

    always @(x1 or x2 or x3)
        if (x2 == 1)
            f = x1;
        else
            f = x3;

endmodule
```

2.10.3 ADVERTENCIA: cómo NO ESCRIBIR código Verilog

- Con los procedimientos, nos vemos tentados a escribir código Verilog como si fuera un programa
  - Verilog no es para programar
  - Debemos evitar variables internas que ocuparán más recursos del FPGA
  - Es difícil predecir cómo será compilado un código escrito estilo programa
- Se sugiere seguir el estilo de los ejemplos del curso
- Regla de oro: si al diseñador no le es fácil determinar qué circuito es descrito por el código, entonces es improbable que la síntesis vaya a realizar el circuito que el diseñador está tratando de describir