

6. Bloques de circuitos combinacionales

angel@uc.cl

IEE2713 Sistemas Digitales

© Angel Abusleme 2019

Un nivel más arriba…

- Los circuitos digitales pueden ser
  - Combinacionales
  - Secuenciales (desde capítulo 7)
- Hasta el momento sólo hemos visto combinacionales
- Hemos aprendido lo básico:
  - Álgebra booleana
  - Compuertas lógicas
  - Optimización
  - Representación de números
- Ahora subiremos un nivel de jerarquía para hablar de circuitos combinacionales con funcionalidades más interesantes



6.1 – 6.5

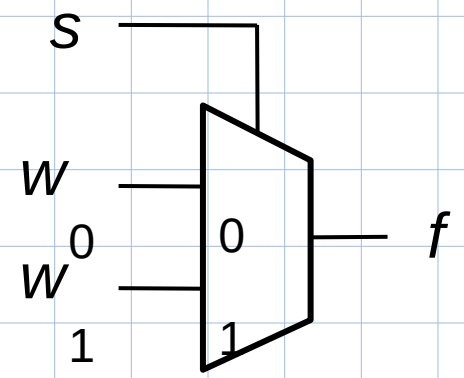
Principales bloques combinacionales

angel@uc.cl

IEE2713 Sistemas Digitales

6.1 Multiplexores

- Eligen una de entre diferentes entradas
- Cuentan con data input elegida por uno o más selectores (s) y una salida f



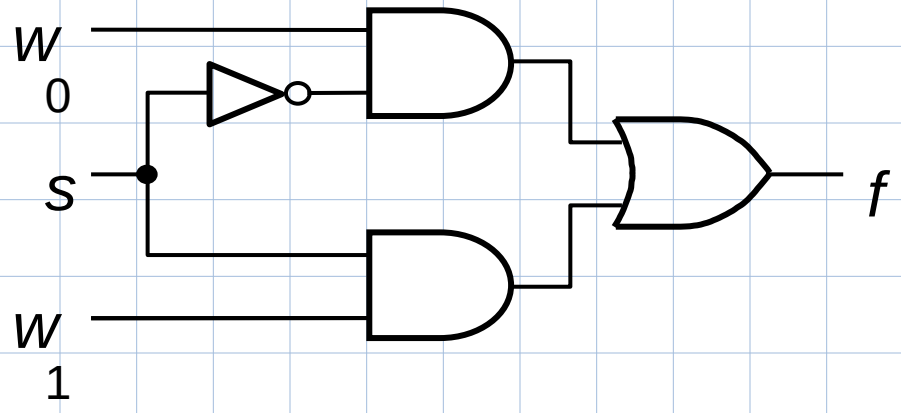
(a) Graphical symbol

s	f
0	w_0
1	w_1

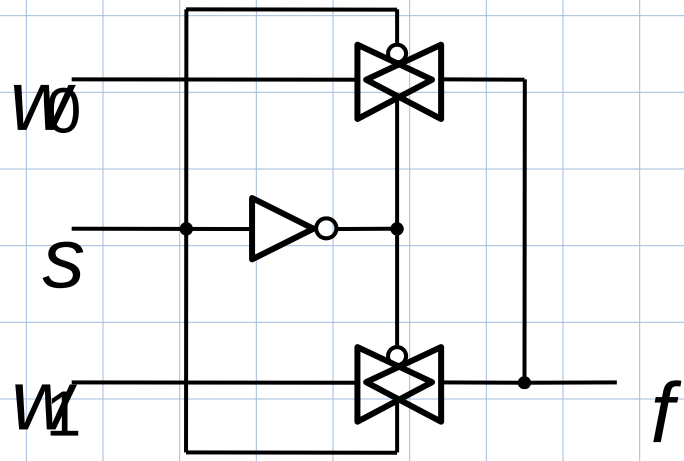
(b) Truth table

6.1 Multiplexores

- Hay diferentes formas de implementarlos



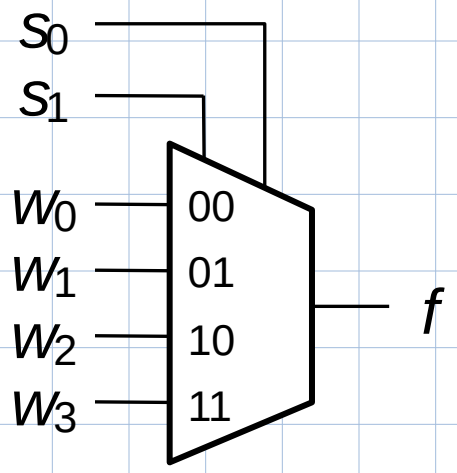
(c) Sum-of-products circuit



(d) Circuit with transmission gates

6.1 Multiplexores

- Pueden tener múltiples entradas



(a) Graphic symbol

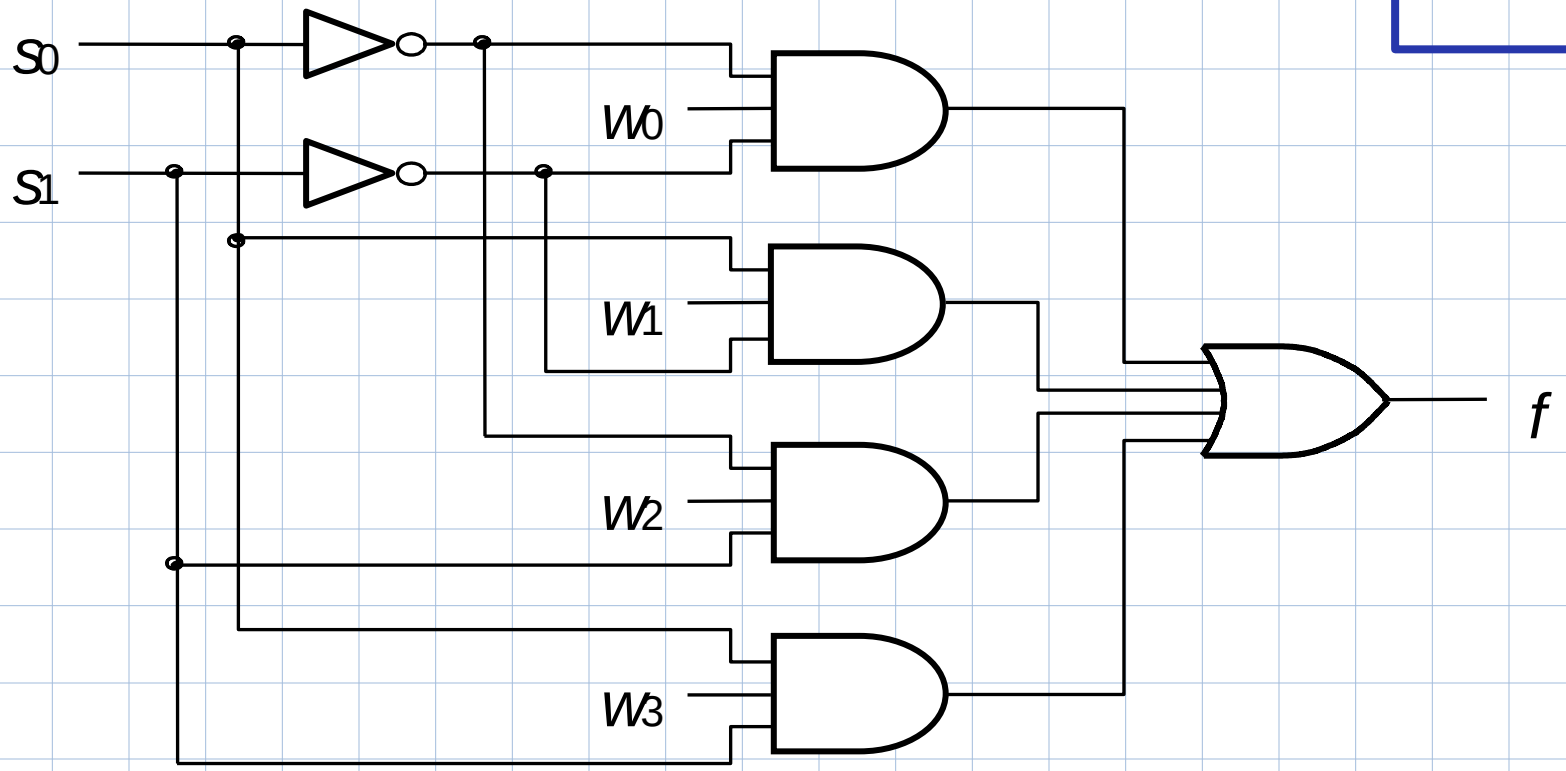
s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

- Las entradas pueden ser palabras de múltiples bits

6.1 Multiplexores

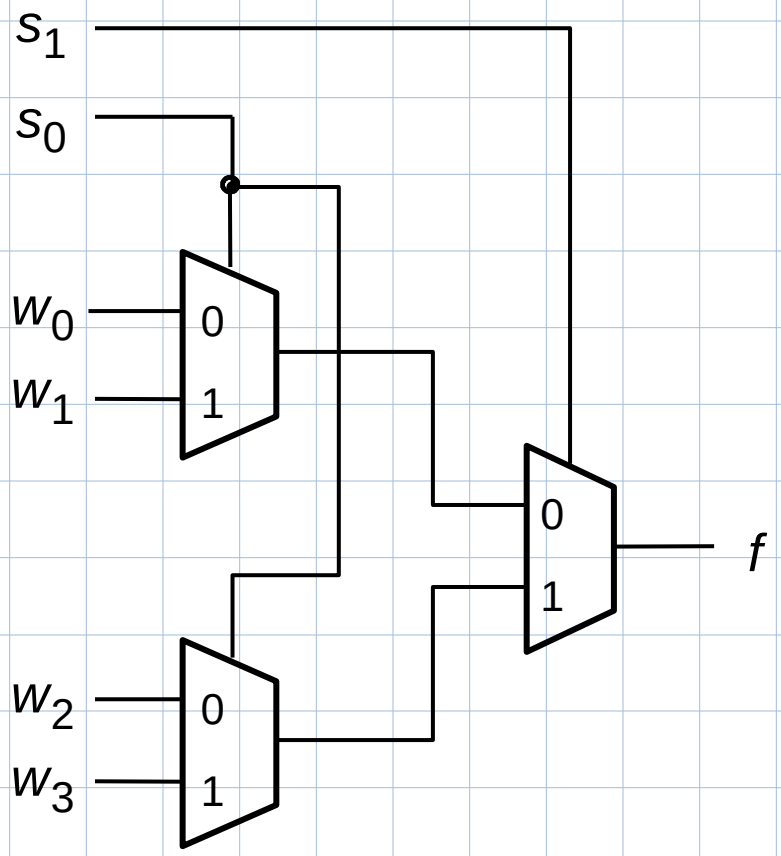
Implementación del circuito de la lámina anterior

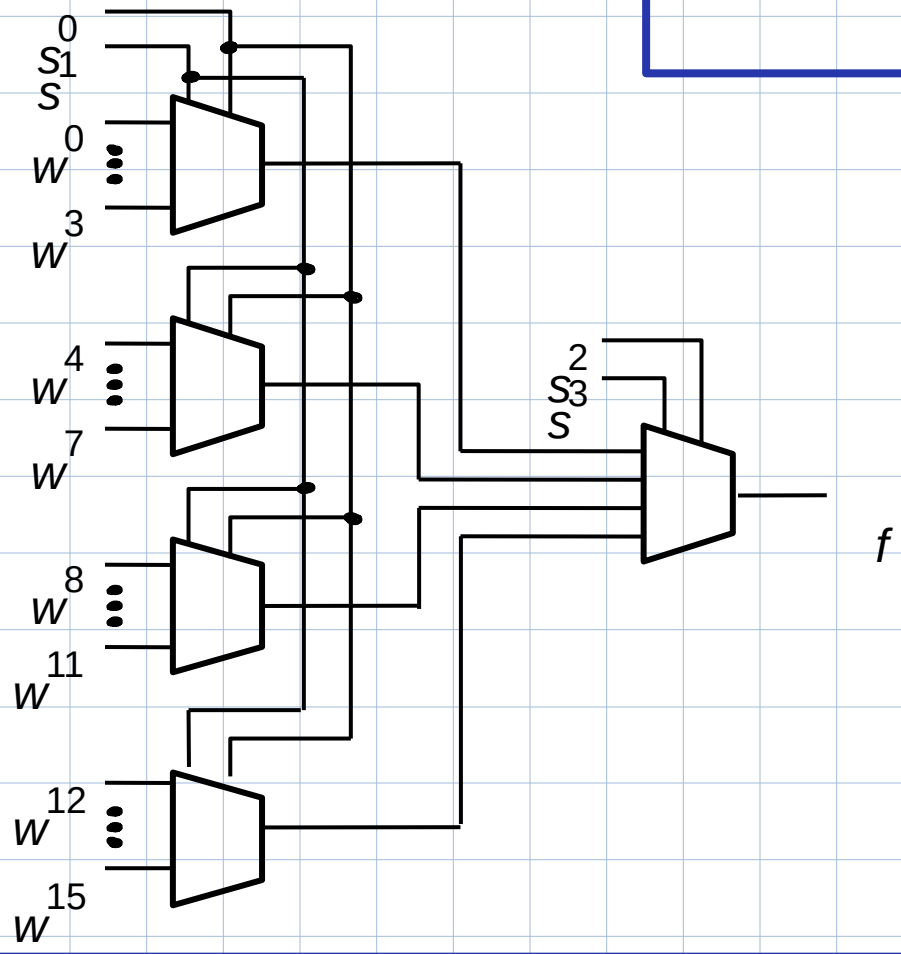


(c) Circuit

6.1 Multiplexores

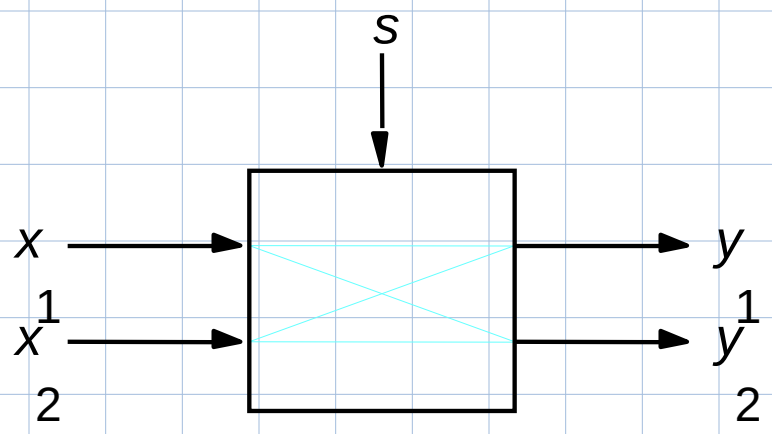
- Es posible tener implementaciones jerárquicas



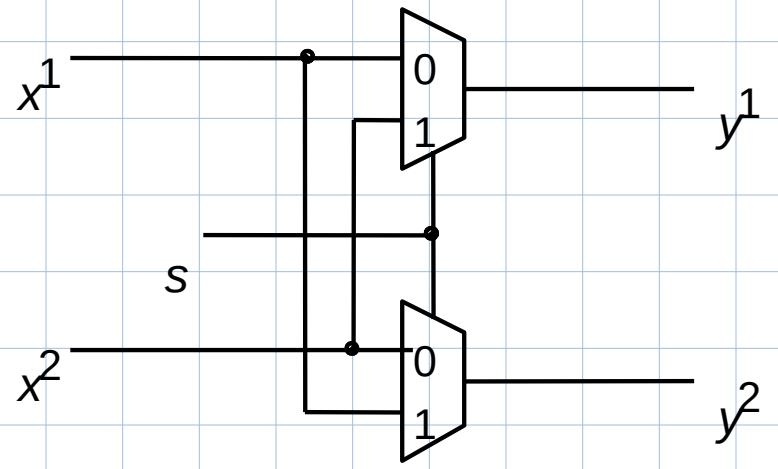


6.1 Multiplexores

- También podemos tener multiplexores con múltiples salidas



(a) A 2x2 crossbar switch

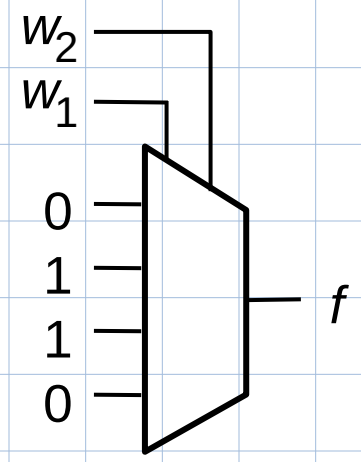


(b) Implementation using multiplexers

6.1.1 Síntesis de funciones lógicas usando multiplexores

- Los multiplexores pueden servir para implementar tablas de verdad

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0



(a) Implementation using a 4-to-1 multiplexer

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

w_1	f
0	w_2
1	$\bar{w}_2$

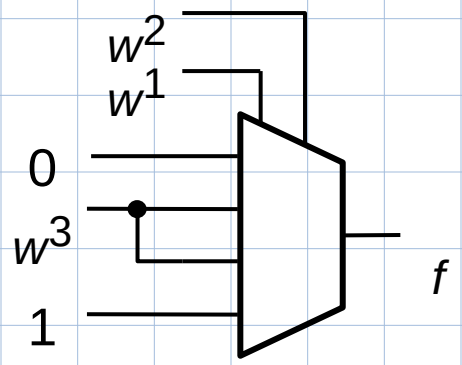
(b) Modified truth table

6.1.1 Síntesis de funciones lógicas usando multiplexores

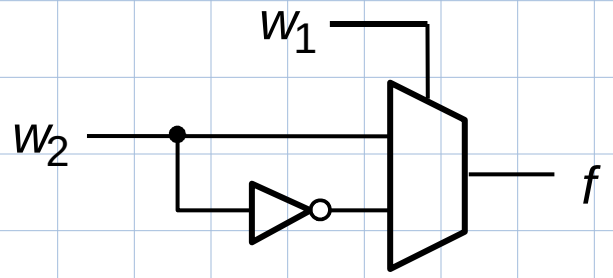
w^1	w^2	w^3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w^1	w^2	f
0	0	0
0	1	w^3
1	0	w^3
1	1	1

(a) Modified truth table



(b) Circuit

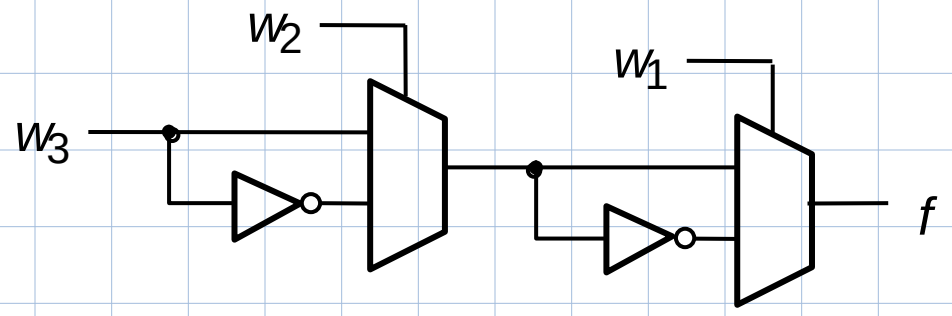


(c) Circuit

6.1.1 Síntesis de funciones lógicas usando multiplexores

w_1	w_2	w_3	f	
0	0	0	0	$w_2 \oplus w_3$
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	$\overline{w_2 \oplus w_3}$
1	0	1	0	
1	1	0	0	
1	1	1	1	

(a) Truth table

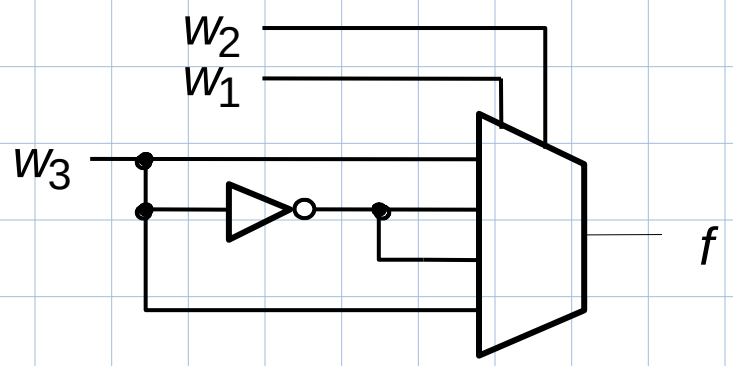


(b) Circuit

6.1.1 Síntesis de funciones lógicas usando multiplexores

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\bar{w}_3$
0	1	1	0	
1	0	0	1	$\bar{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

6.1.2 Síntesis de multiplexores usando expansión de Shannon

- **Teorema de expansión de Shannon:**
  Cualquier función booleana puede ser escrita de la forma

$$f(w_1, w_2, \ldots, w_n) = \overline{w_1}\, f(0, w_2, \ldots, w_n) + w_1\, f(1, w_2, \ldots, w_n)$$

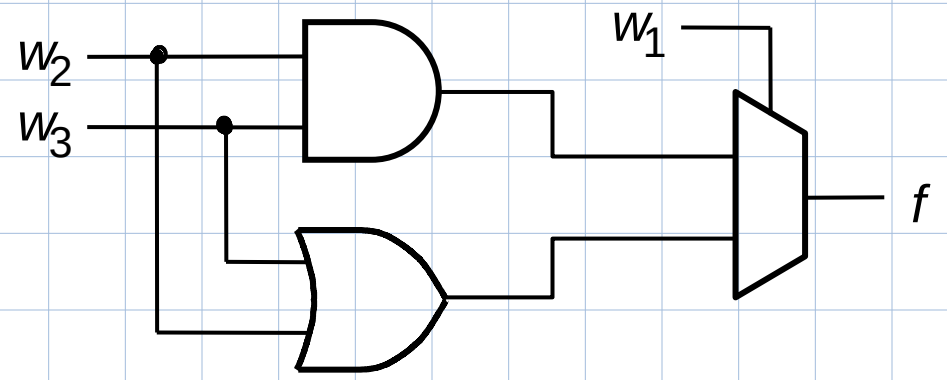
- Esta expansión puede ser hecha en términos de cualquiera de las n variables

6.1.2 Síntesis de multiplexores usando expansión de Shannon

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

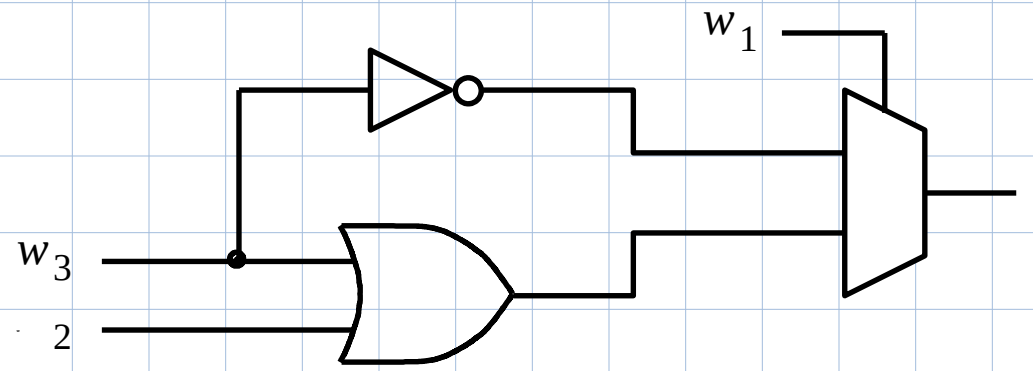
w_1	f
0	w_2w_3
1	$w_2 + w_3$

(b) Truth table

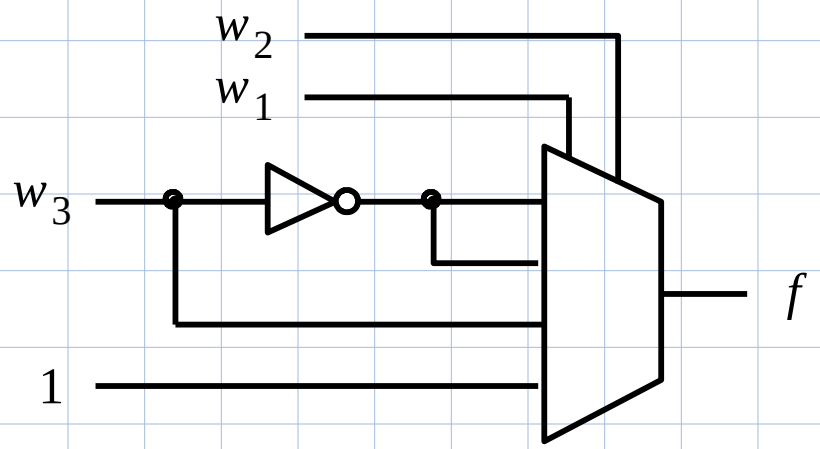


(b) Circuit

6.1.2 Síntesis de multiplexores usando expansión de Shannon

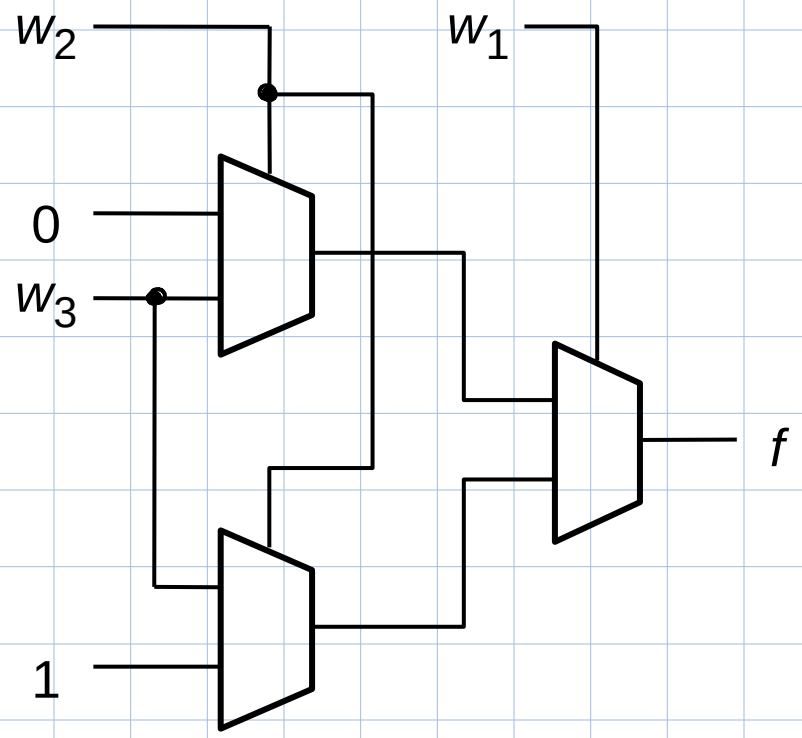


(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

6.1.2 Síntesis de multiplexores usando expansión de Shannon



6.2 Decodificadores

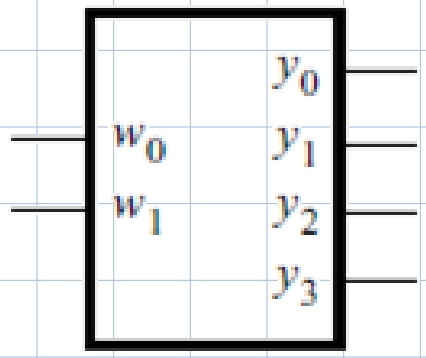
- Decodifican un número binario a una forma menos compacta
  - Pasa pocos bits de entrada a muchos bits de salida
  - Esto puede ser muy útil en algunos circuitos
- Existen varios tipos de codificadores, ej:
  - One-hot
  - Termómetro
  - Demultiplexores (caso especial)
- Pueden ser usados en circuitos complicados
  - En procesadores
  - En conversores D/A (IEE3433)

6.2 Decodificadores one-hot

- Sus inputs representan un número binario que es decodificado para activar la salida correspondiente

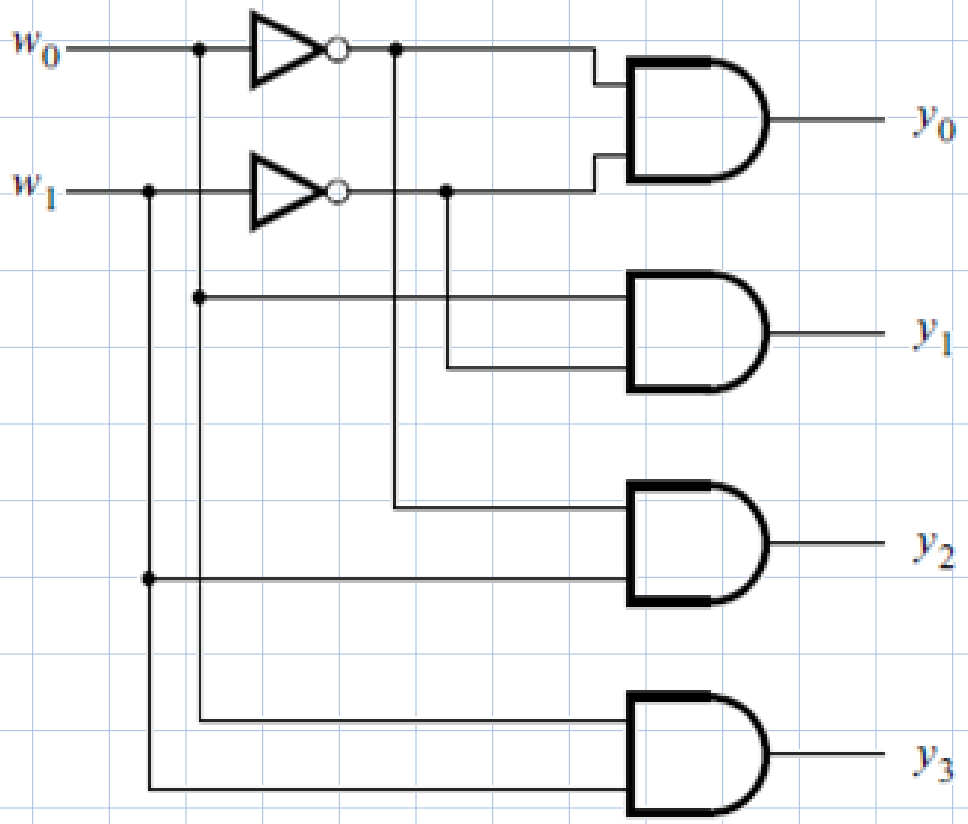
w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a) Truth table



(b) Graphical symbol

6.2 Decodificadores one-hot



(c) Logic circuit

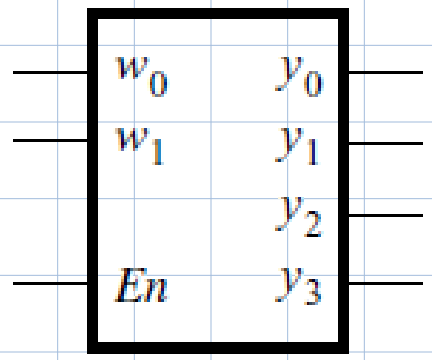
¿Cómo escala la complejidad de un decodificador a medida que aumenta el número de entradas?

6.2 Decodificadores con enable

- Enable (En) permite encender o apagar todas las salidas, independientemente de las entradas

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

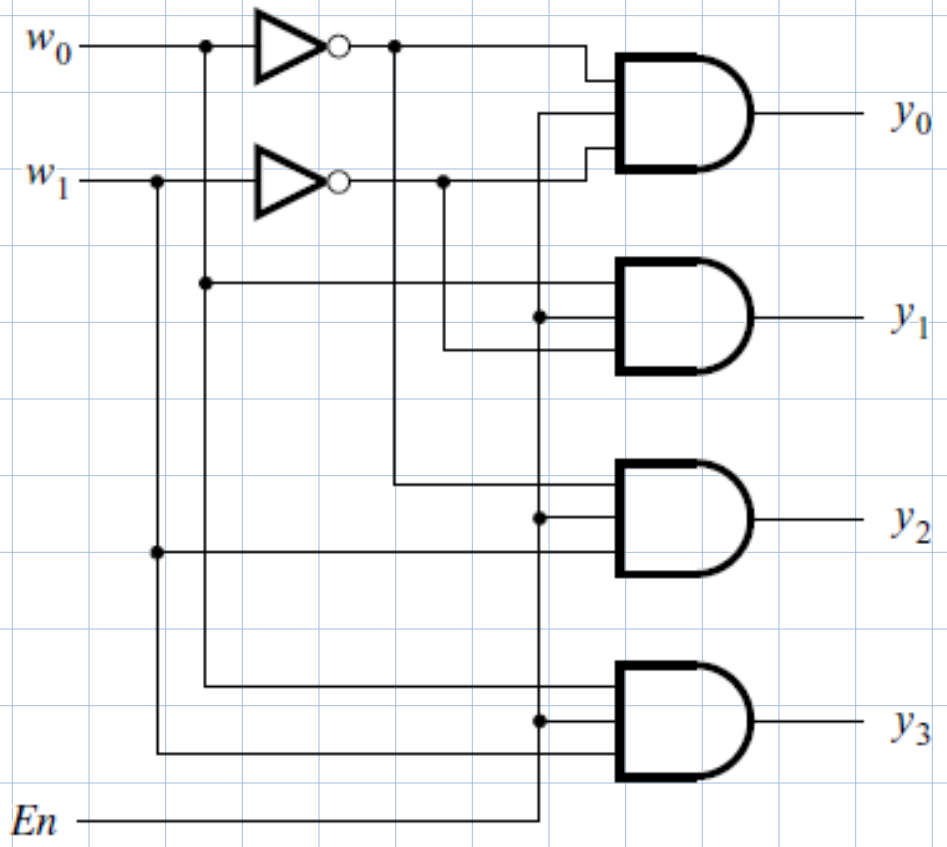
(a) Truth table



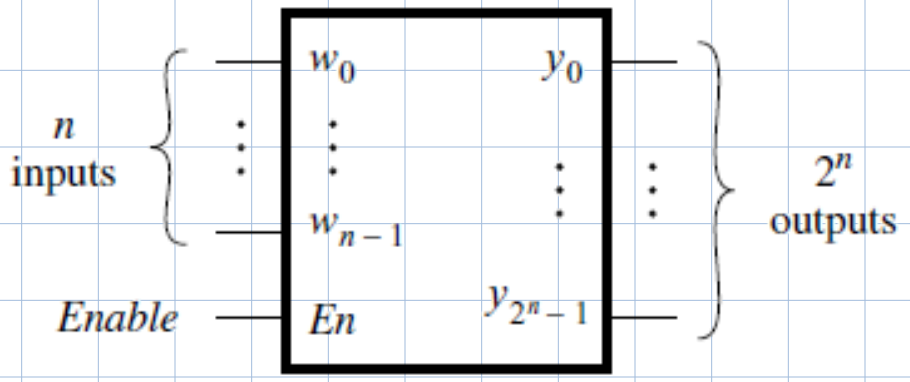
(b) Graphical symbol

El enable o habilitador puede ser aplicado a cualquier compuerta lógica. Algunas compuertas lógicas presentan salida con estado Z cuando son deshabilitadas.

6.2 Decodificadores con enable



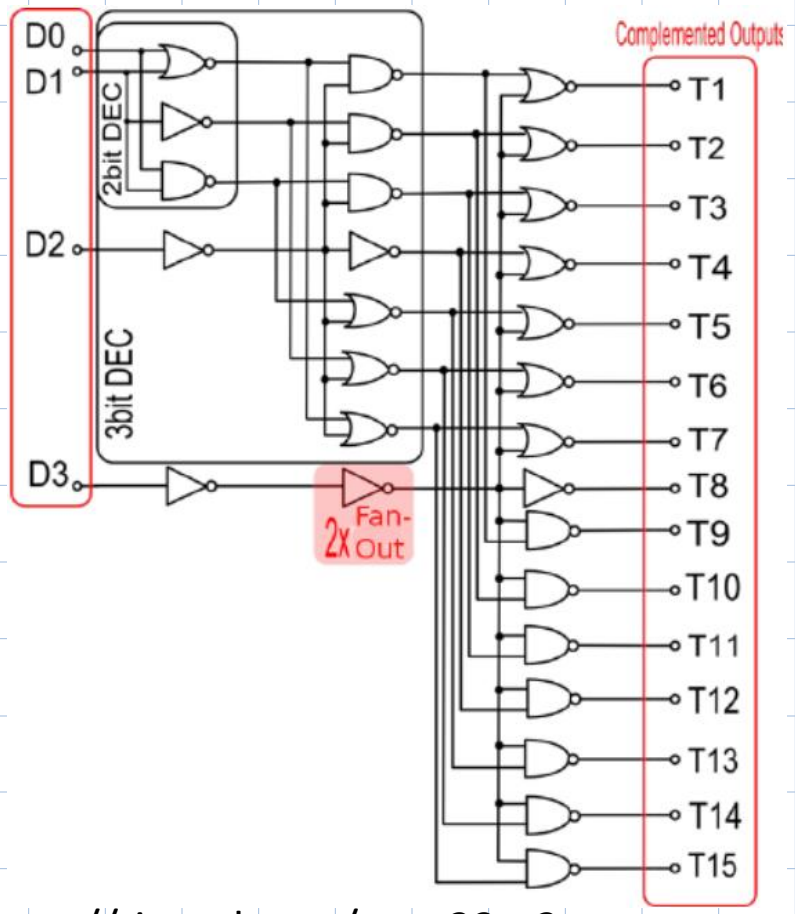
(c) Logic circuit



(d) An n-to-2^n decoder

6.2 Decodificadores termómetro

- La entrada es binaria, la salida es termómetro

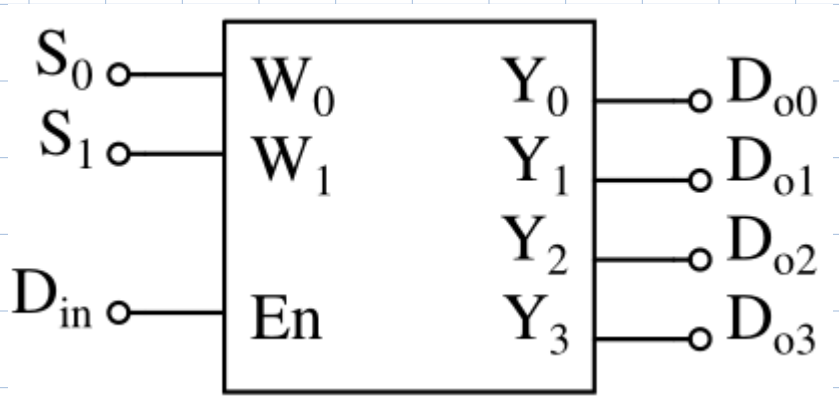


https://tinyurl.com/yxeq23m2

Ejemplo: uso en conversor D/A termómetro

6.2.1 Demultiplexores

- Permiten distribuir 1 entrada a n salidas
- Es posible implementarlos usando decodificadores
  - Ejemplo: demux 1:4 implementado con un decodificador

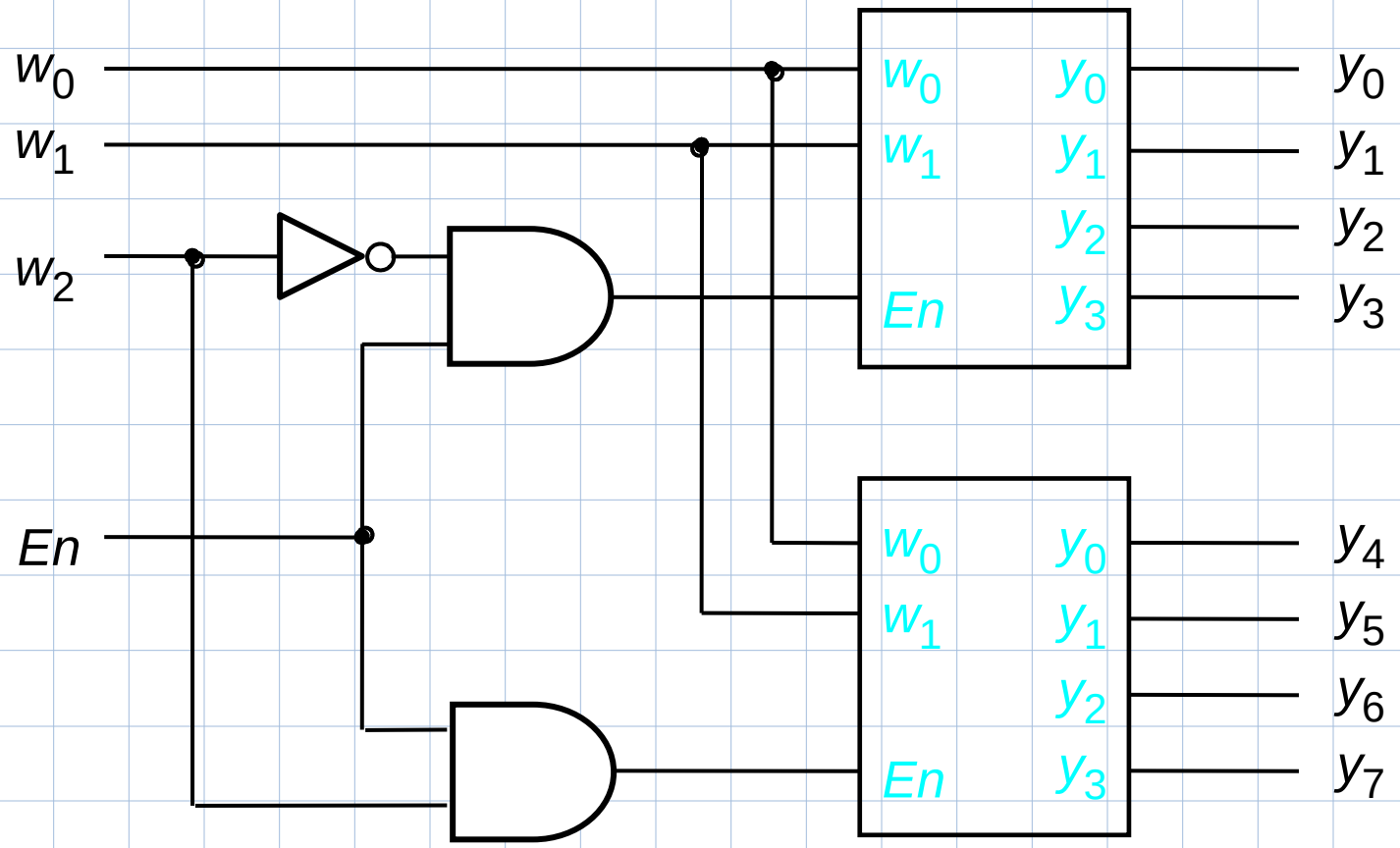


- Las salidas que no están activadas producen 0 - **¿no es obvio?**

¿Cuál es la diferencia entre un decodificador y un demux?

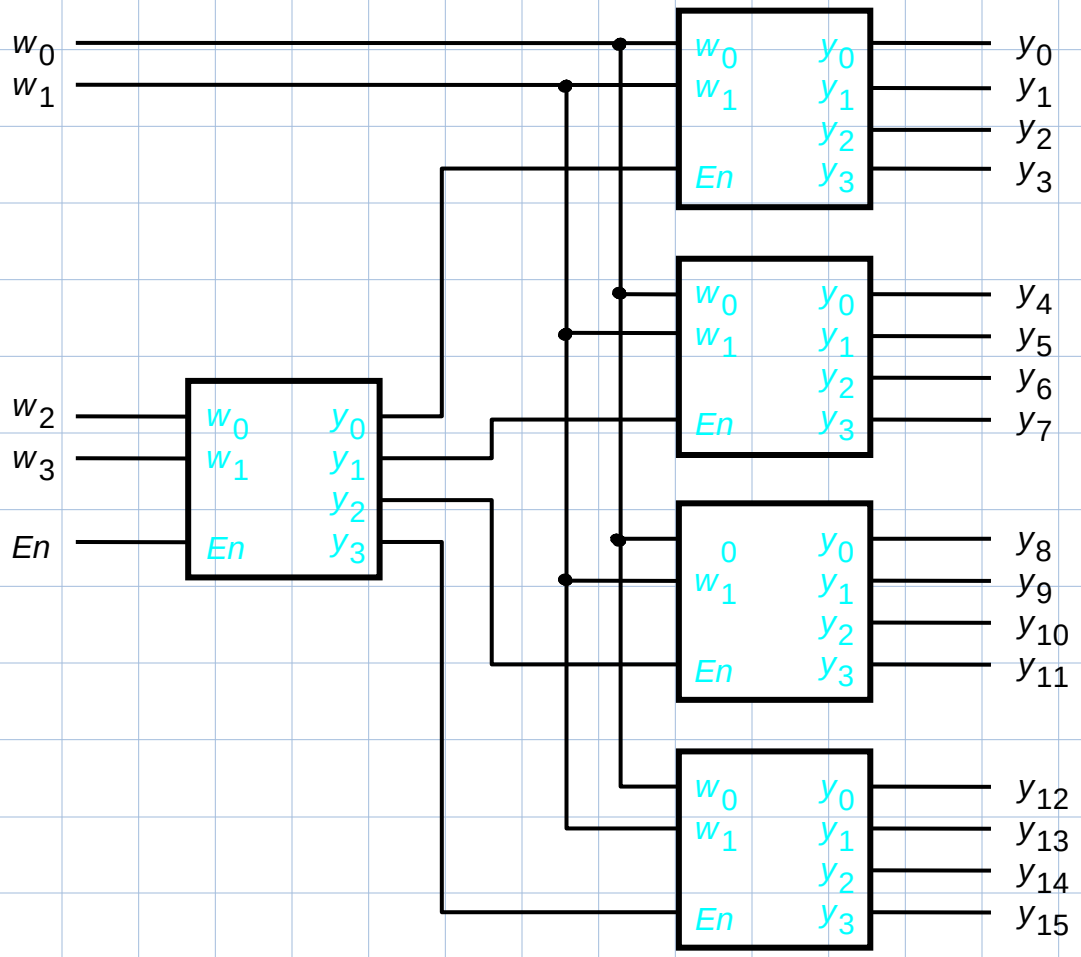
6.2.1 Demultiplexores

¿Qué es esto?



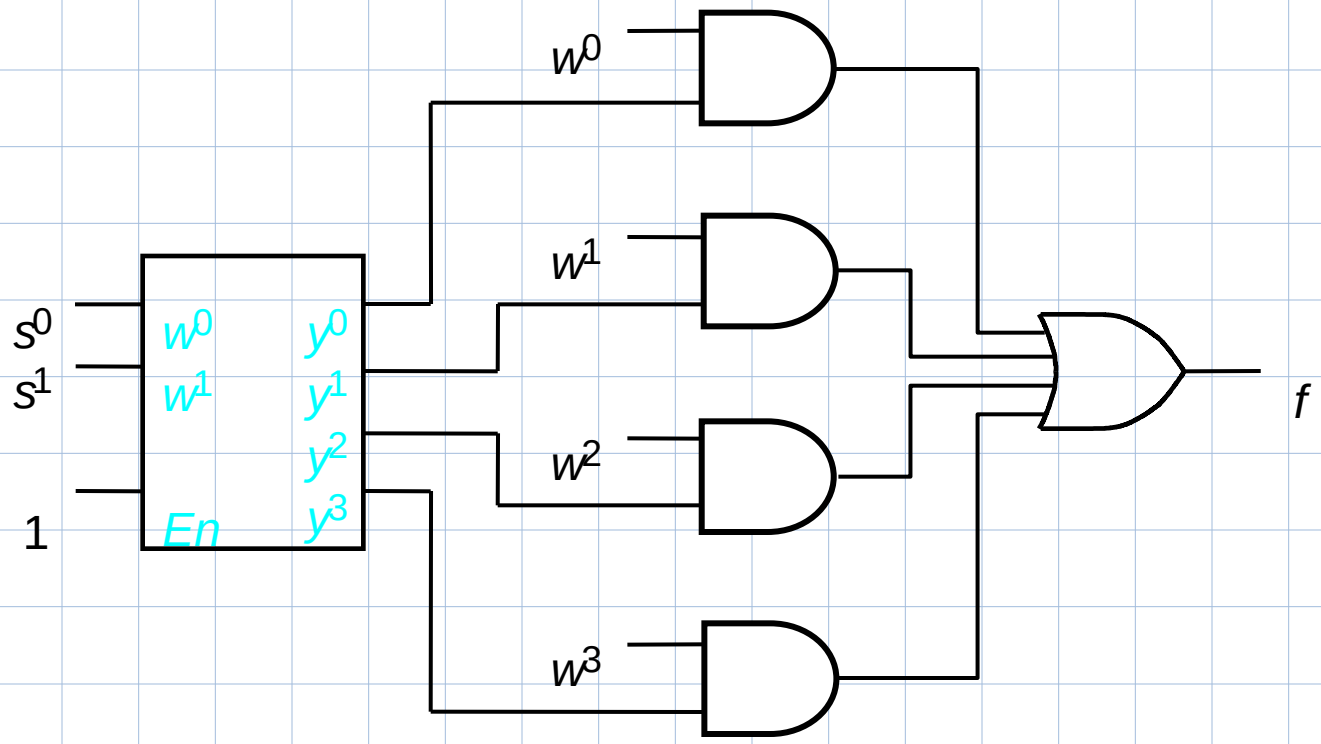
6.2.1 Demultiplexores

- Etc.



6.2.1 Demultiplexores

- Un demultiplexor nos permite construir un mux

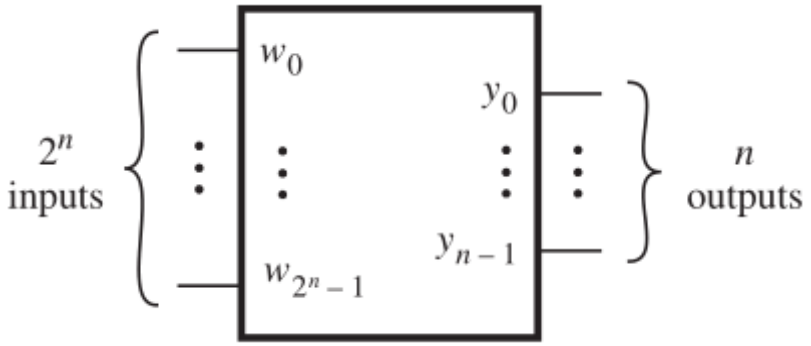


6.3 Codificadores (encoders)

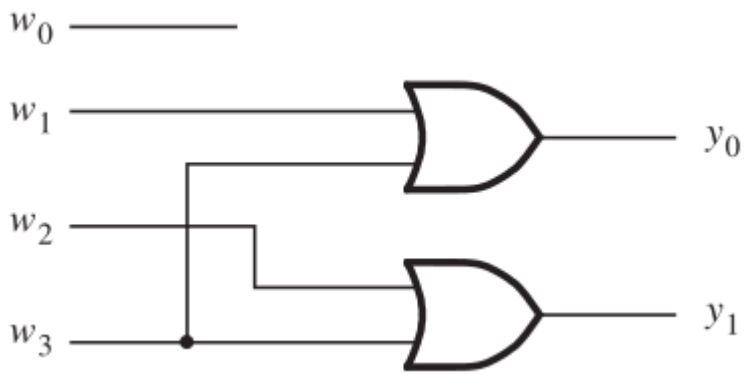
- Función contraria a la de un decodificador: codifican el número para representarlo en forma más compacta (en binario)
  - Pasa de muchos bits de entrada a pocos bits de salida
- Existen varios tipos de codificadores, por ejemplo:
  - One-hot (también llamados binarios)
  - De prioridad
  - Wallace
- Pueden ser usados como parte de circuitos más complicados
  - En procesadores
  - En conversores A/D (IEE3433)

6.3.1 Codificadores binarios (deberían llamarse codificadores one-hot)

- Codifican información de 2^n entradas en un código de salida de n bits binarios
  - A veces la información no viene en forma binaria, sino en alguna otra representación
- Se utiliza para reducir el número de bits necesarios para representar la información
- La entrada viene en forma **one-hot**



w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

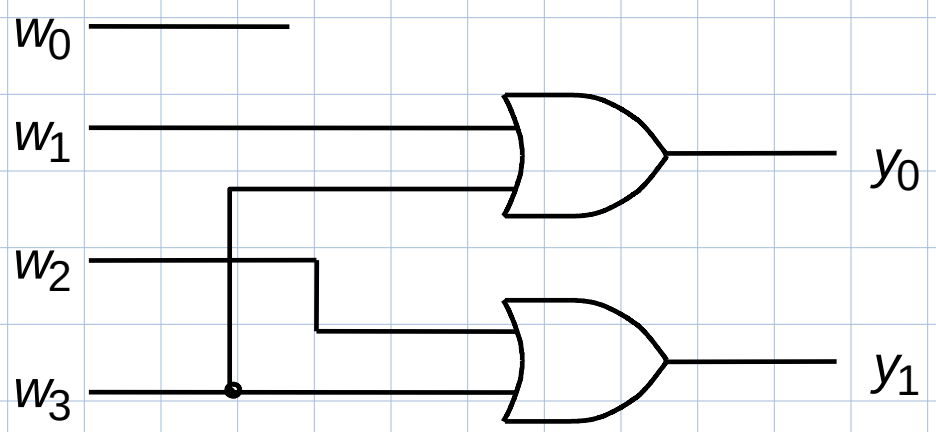


6.3.1 Codificadores binarios

Tabla de verdad e implementación

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

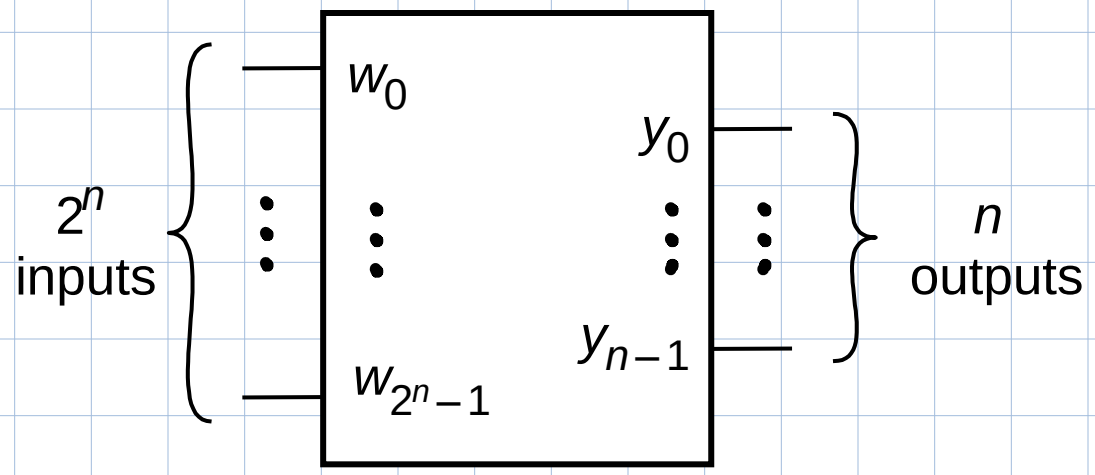
(a) Truth table



(b) Circuit

6.3.2 Codificadores de prioridad

- La salida del encoder indica la entrada activa es más significativa (de mayor prioridad)
- Cuando una entrada con alta prioridad es encendida, las otras entradas de menor prioridad son ignoradas (don’t care)
- La codificación de la entrada según prioridad también se denomina **codificación termómetro**
- Son muy usados en ADCs flash



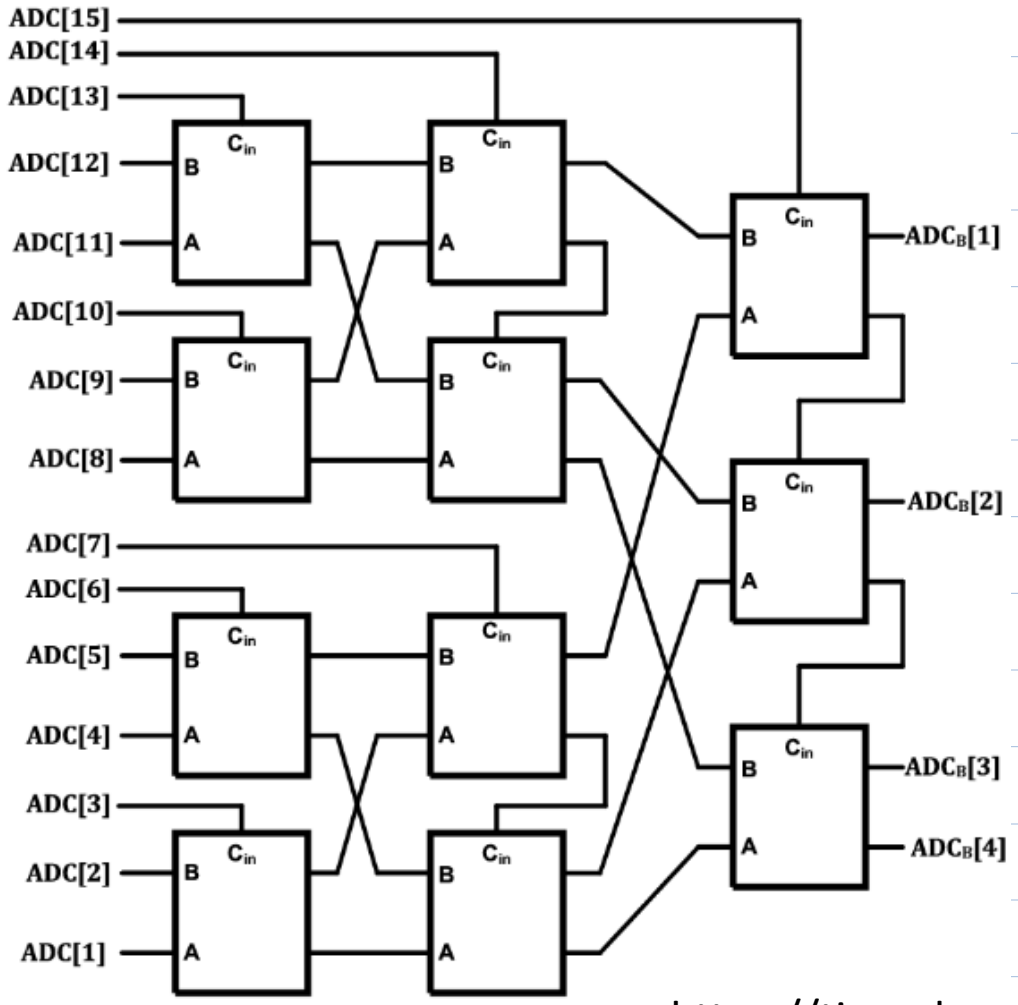
6.3.2 Codificadores de prioridad

- En la tabla de verdad podemos poner don't cares

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

- En el caso de codificación termómetro, los x deben ser 1
  - Si un x es 0, se llama burbuja… **¿por qué?**

6.3.3 Codificadores Wallace



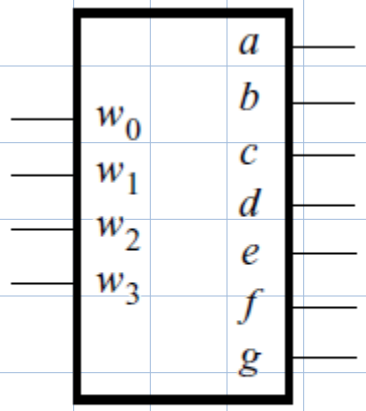
17: Block diagram of the Wallace-Tree thermometer-to-binary encoder

https://tinyurl.com/y2wuga74

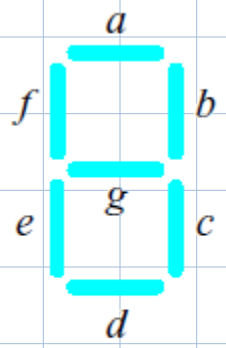
- Forma eficiente de implementar un codificador de prioridad
- La entrada **debe ser termómetro**
- **¿Alguien puede explicar su funcionamiento?**

6.4 Conversores de código

- Convierten a otro tipo de codificación
- Pueden ser codificadores o decodificadores, u otros, por ejemplo
  - Decodificador BCD a 7 segmentos
  - Conversor binario a Código Gray
  - Conversor Código Gray a binario



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	1	1

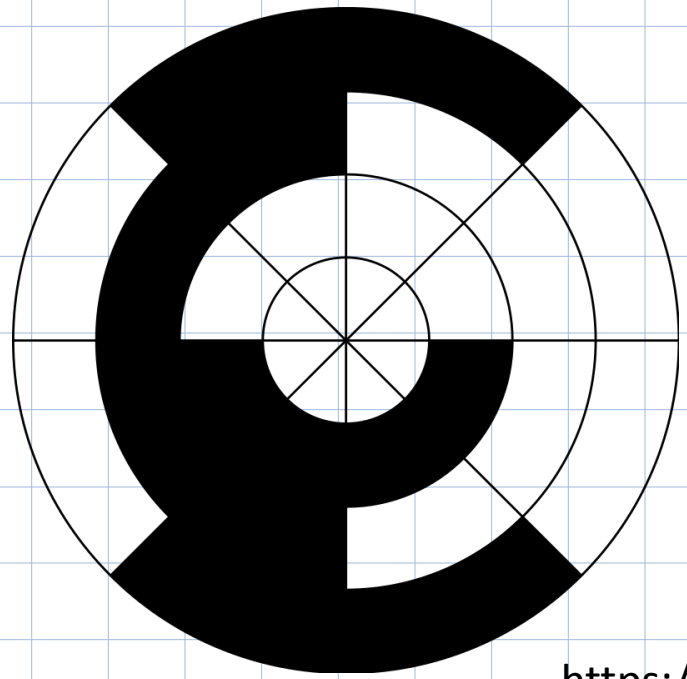
(c) Truth table

Código Gray

Números sucesivos están a distancia Hamming 1

¿Tabla de verdad?

Usos



https://tinyurl.com/jvsykwy

6.5 Circuitos de comparación aritmética

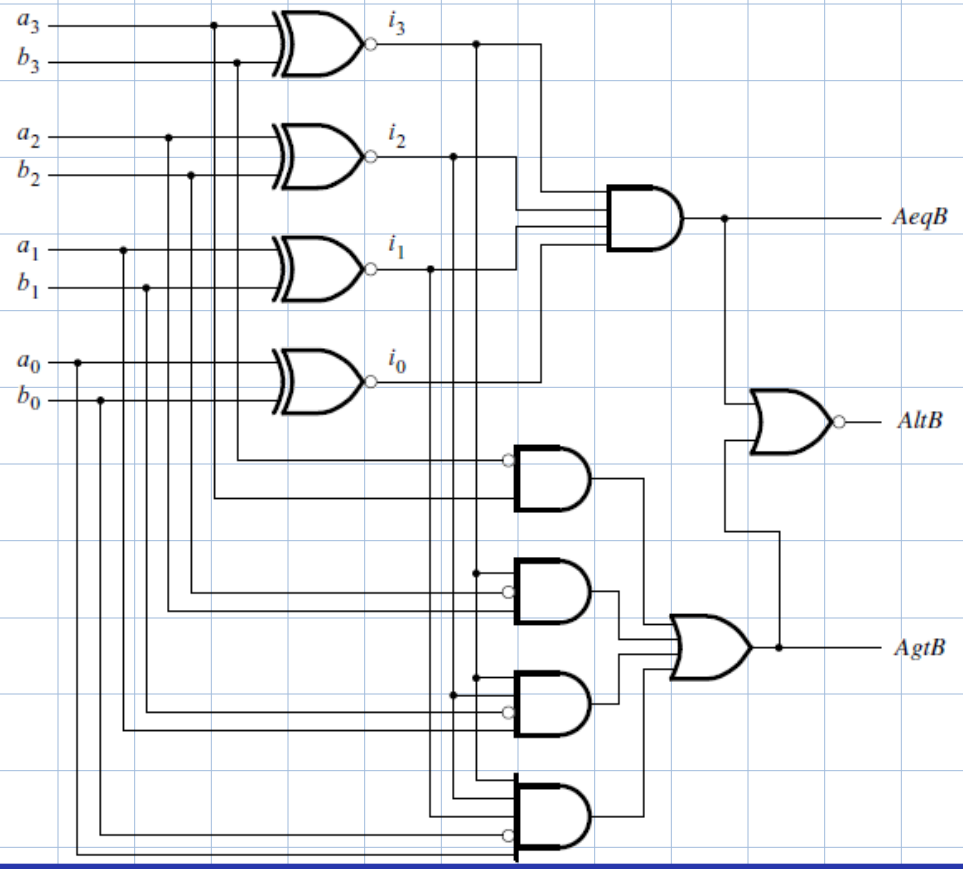
- Comparan tamaño relativo de números binarios
- Consideremos comparadores con entradas A y B:
  - AeqB
  - AgtB
  - AltB
- Sea $i_k = \overline{a_k \oplus b_k}$

  Entonces

$$AeqB = i_3 i_2 i_1 i_0$$

$$AgtB = a_3\bar{b}_3 + i_3 a_2 \bar{b}_2 + i_3 i_2 a_1 \bar{b}_1 + i_3 i_2 i_1 a_0 \bar{b}_0$$

$$AltB = \overline{AeqB + AgtB}$$



¿Conclusión? Es un poco caro comparar dos números. Tener esto en cuenta al escribir código.



6.6 – 6.7

Principales bloques combinacionales en Verilog

angel@uc.cl

IEE2713 Sistemas Digitales

6.6 Verilog para circuitos combinacionales

- Ya aprendimos varios circuitos lógicos útiles
- Ahora veamos cómo describirlos en Verilog
- Dada su complejidad, en lugar de especificarlos como conexiones circuitales, los especificaremos según su comportamiento
- Aprenderemos formas más rigurosas de hacer tal descripción

6.6.1 El operador condicional

- Es un operador que toma decisiones según condición
- Podemos agregar paréntesis para mejorar lectura
- Puede usar en asignación continua y en declaraciones de procedimiento (always block)
- Ejemplo: multiplexor

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;

    assign f = s ? w1 : w0;

endmodule
```

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output reg f;

    always @(w0, w1, s)
        f = s ? w1 : w0;

endmodule
```

6.6.1 El operador condicional

- Podemos anidar operadores condicionales
  - No abusar de esto, difícil leerlo

```
module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output f;

    assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

6.6.2 Declaración if-else

- Es otra declaración que permite tomar decisiones
- Caso más general que el operador condicional
- Si la expresión evaluada es verdadera, entonces se ejecuta la primera declaración o bloque entre begin y end
- Si es falsa, se ejecuta la segunda declaración
- Debe ir en un bloque always porque es un procedimiento que se ejecuta

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output reg f;

always @(w0, w1, s)
    if (s==0)
        f = w0;
    else
        f = w1;

endmodule
```

6.6.2 Declaración if-else

- Es posible anidar declaraciones else-if
- Siempre es bueno terminar con un else (este ejemplo es malo!!!)

```
module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output reg f;

    always @(*)
        if (S == 2'b00)
            f = w0;
        else if (S == 2'b01)
            f = w1;
        else if (S == 2'b10)
            f = w2;
        else if (S == 2'b11)
            f = w3;

endmodule
```

6.6.2 Declaración if-else

- Aquí escribimos el multiplexor en forma más compacta usando vectores
- Es más fácil leer el código de esta forma que con escalares

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        if (S == 0)
            f = W[0];
        else if (S == 1)
            f = W[1];
        else if (S == 2)
            f = W[2];
        else if (S == 3)
            f = W[3];

endmodule
```

6.6.2 Declaración if-else

- Por supuesto que también podemos trabajar con jerarquía
- En este caso instanciamos 4 multiplexores 4:1 para crear un multiplexor 16:1

```
module mux16to1 (W, S, f);
    input [0:15] W;
    input [3:0] S;
    output f;
    wire [0:3] M;

    mux4to1 Mux1 (W[0:3], S[1:0], M[0]);
    mux4to1 Mux2 (W[4:7], S[1:0], M[1]);
    mux4to1 Mux3 (W[8:11], S[1:0], M[2]);
    mux4to1 Mux4 (W[12:15], S[1:0], M[3]);
    mux4to1 Mux5 (M[0:3], S[3:2], f);

endmodule
```

6.6.3 Declaración case

- Permite hacer lo mismo que un condicional anidado
- **Muy útil** para describir tablas de verdad
- Debe ir en bloque always porque es un procedimiento
- Siempre debiera terminar con default (ejemplo malo)

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        case (S)
            0: f = W[0];
            1: f = W[1];
            2: f = W[2];
            3: f = W[3];
        endcase

endmodule
```

Ejemplo - Declaración case

- Este ejemplo es un decodificador de binario a one-hot con enable
- Aquí concatenamos en el argumento del case
- Al expresar la entrada y la salida en binario, la tabla de verdad queda a la vista

```
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;

    always @(W, En)
        case ({En, W})
            3'b100: Y = 4'b1000;
            3'b101: Y = 4'b0100;
            3'b110: Y = 4'b0010;
            3'b111: Y = 4'b0001;
            default: Y = 4'b0000;
        endcase

endmodule
```

Ejemplo - Declaración case

- Otra posible implementación con un if para el enable
- Otra opción sería hacer un AND entre las salidas y el enable

```
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;

    always @(W, En)
    begin
        if (En == 0)
          Y = 4'b0000;
        else
          case (W)
              0: Y = 4'b1000;
              1: Y = 4'b0100;
              2: Y = 4'b0010;
              3: Y = 4'b0001;
          endcase
    end
endmodule
```

Ejemplo - Declaración case

- Por supuesto que siempre es posible usar jerarquía para implementar bloques más complicados
- En este caso, decodificador de 4 a 16

```
module dec4to16 (W, En, Y);
    input [3:0] W;
    input En;
    output [0:15] Y;
    wire [0:3] M;

    dec2to4 Dec1 (W[3:2], M[0:3], En);
    dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
    dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
    dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
    dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

Ejemplo case: una ALU sencilla

- ALU: Unidad aritmético-lógica
- Realiza operaciones aritméticas y lógicas sobre operandos de n bits
- El chip 74381 tiene una ALU
- El ejemplo muestra tabla de verdad y código Verilog para esta ALU

```
// 74381 ALU
module alu(s, A, B, F);
    input [2:0] S;
    input [3:0] A, B;
    output reg [3:0] F;

    always @(S, A, B)
        case (S)
            0: F = 4'b0000;
            1: F = B - A;
            2: F = A - B;
            3: F = A + B;
            4: F = A ^ B;
            5: F = A | B;
            6: F = A & B;
            7: F = 4'b1111;
        endcase
endmodule
```

Operation	Inputs $s_2\ s_1\ s_0$	Outputs F
Clear	0 0 0	0 0 0 0
B−A	0 0 1	$B - A$
A−B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

Declaraciones casex y casez

```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;

    always @(W)
    begin
        z = 1;
        casex (W)
            4'b1xxx: Y = 3;
            4'b01xx: Y = 2;
            4'b001x: Y = 1;
            4'b0001: Y = 0;
            default: begin
                        z = 0;
                        Y = 2'bx;
                     end
        endcase
    end
endmodule
```

- Son variantes del case
- Permiten usar x's y z's como comodines
  - De esta forma no tenemos que poner una tabla tan larga
  - casez trata todas las z's en las alternativas (ítems) y en la expresión case, como don't cares
  - casex trata todas las x's y z's como don't cares
- Es posible que haya más de un ítem case que iguale la expresión
  - En ese caso, el primer ítem es el que vale
  - En ese caso, mejor usar if-else para tener explícita la prioridad
- USARLAS CON PRECAUCIÓN o mejor no usarlas

6.6.4 Loop for

- Estructura que permite expresar cierta regularidad
- Debe ir en bloque always porque es un procedimiento
- La variable de control debe ser tipo entero
- A diferencia de los lenguajes de programación en que se especifican cambios en cada iteración del loop, en Verilog el loop for especifica diferentes subcircuitos en cada iteración

```
module dec2to4 (W, En, Y);
    input [1:0] W;
    input En;
    output reg [0:3] Y;
    integer k;

    always @(W, En)
        for (k = 0; k <= 3; k = k+1)
        if ((W == k) && (En == 1))
            Y[k] = 1;
        else
            Y[k] = 0;

endmodule
```

Loop for en un codificador de prioridad

- En Verilog, si una señal es especificada varias veces durante una ejecución de bloque always, retiene el valor de la última asignación
- Este ejemplo muestra justamente este comportamiento
- Quiero ser honesto, yo (Angel Abusleme) **no recomiendo** el bloque for en Verilog
  - Al menos no es para principiantes
  - No es como el loop for en programación
  - Puede resultar en circuitos subóptimos
  - Piensen que cada iteración usa un circuito y una lógica nueva…

```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;
    integer k;

    always @(W)
    begin
        Y = 2'bx;
        z = 0;
        for (k = 0; k < 4; k = k+1)
            if (W[k])
            begin
                Y = k;
                z = 1;
            end
    end
endmodule
```

6.6.5 Operadores en Verilog

Table 4.2. Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1

- Hay distintos tipos:
  - operadores bit a bit (bitwise)
  - Lógicos
  - De reducción
  - Aritméticos
  - Relacionales o de comparación
  - De igualdad
  - De desplazamiento
  - De concatenación
  - De replicación
  - Condicional (ya lo vimos)

6.6.5 Operadores en Verilog

Arithmetic	+	Addition	2
	−	Subtraction	2
	−	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal to	2
	<=	Less than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

- Hay distintos tipos:
  - operadores bit a bit (bitwise)
  - Lógicos
  - De reducción
  - Aritméticos
  - Relacionales o de comparación
  - De igualdad
  - De desplazamiento
  - De concatenación
  - De replicación
  - Condicional (ya lo vimos)

Operadores bit a bit: !, &, |, ^, ~^

Tablas de verdad para operadores bit a bit

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~ ^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Operadores lógicos: !, &&, ||

- El operador ! tiene el mismo efecto que ~ en escalares
- Pero si el argumento es un vector A = $a_1a_2a_3$, entonces

$$!A = \overline{a_2 + a_1 + a_0}$$

- El operador && produce un AND lógico entre dos vectores:

$$A\&\&B = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)$$

- El operador || produce un OR lógico entre dos vectores:

$$A||B = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)$$

Operadores de reducción: &, ~&, |, ~|, ^, ~^

- Reducen un vector a un bit de salida
- Por ejemplo

$$\&A = a_2 \cdot a_1 \cdot a_0$$

- Mientras que

$$\hat{}A = a_2 \oplus a_1 \oplus a_0$$

- Y sirve para calcular paridad

Operadores aritméticos +, -, *, /

- Producen operaciones aritméticas:

  A+B es la suma de A y B

  A-B es la diferencia entre A y B

  -A es el complemento-2 de A
- La operación / no siempre es soportada por sistemas de CAD
  - Requiere alguna biblioteca
- **Eviten multiplicar y dividir**, a menos que sepan exactamente lo que están haciendo
  - En lo posible, usen multiplicación o división por potencias de 2

Operadores relacionales y de igualdad >, <, >=, <=, ==, !=

```
module compare (A, B, AeqB, AgtB, AltB);
    input [3:0] A, B;
    output reg AeqB, AgtB, AltB;

    always @(A, B)
    begin
        AeqB = 0;
        AgtB = 0;
        AltB = 0;
        if(A == B)
          AeqB = 1;
        else if (A > B)
          AgtB = 1;
        else
          AltB = 1;
    end
endmodule
```

- Típicamente usados en expresiones condicionales y en loops for
- Misma sintaxis que en C
- El resultado del operador igualdad == o desigualdad != es ambiguo (x) si algún operando contiene x o z

Operadores de desplazamiento a la derecha >> y a la izquierda <<

- Llenan con ceros el resto de los bits
  - Si A = $a_2a_1a_0$, B = A << 1 produce B = a_1a_00 (multiplica por 2)
  - Si A = $a_2a_1a_0$, B = A >> 2 produce B = $00a_2$ (divide por 4)
- Si uno quiere hacer desplazamiento circular o rotación, debe implementarlo usando concatenación
- Si uno quiere hacer desplazamiento a la derecha con extensión de signo, también hay que usar concatenación

Operadores de concatenación y replicación

- Concatenación:

  D = {A, B} produce D = $a_2a_1a_0b_2b_1b_0$

  E={3'b111, A, 2'b00} produce E = $111a_2a_1a_000$
- El operador replicación puede ser usado junto al de concatenación

  {3{A}} es equivalente a {A, A, A}

  {4{2'b10}} produce 8'b10101010

Precedencia de operadores

Operator type	Operator symbols	Precedence
Complement	! ~ –	Highest procedence
Arithmetic	* / + –	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& ^ ~^ | ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

6.6.6 El constructo generate

- Es un loop capaz de producir múltiples instancias de un circuito
  - El resultado es más hardware
- El subcircuito es definido en un bloque delineado por generate y endgenerate
- El índice es del tipo genvar, entero > 0

```
module addern (carryin, X, Y, S, carryout);
        parameter n=32;
        input carryin;
        input [n-1:0] X, Y;
        output [n-1:0] S;
        output carryout;
        wire [n:0] C;
        genvar k;
        assign C[0] = carryin;
        assign carryout = C[n];
        generate
            for (k = 0; k < n; k = k+1)
            begin: fulladd_stage
                wire z1, z2, z3; //wires within full-adder
                xor (S[k], X[k], Y[k], C[k]);
                and (z1, X[k], Y[k]);
                and (z2, X[k], C[k]);
                and (z3, Y[k], C[k]);
                or (C[k+1], z1, z2, z3);
            end
        endgenerate
endmodule
```

6.6.7 Tareas y funciones

- Sirven para modularizar funciones grandes, facilitando su descripción en Verilog
- Tareas (task) son especies de subrutinas en lenguajes de alto nivel
- Funciones (function) son similares a funciones en lenguajes de alto nivel y deben devolver un resultado
- Yo (Angel Abusleme) no recomiendo mucho el uso de tareas y funciones en Verilog, a menos que sepamos exactamente lo que estamos haciendo
  - Siempre es posible hacer lo mismo con un poco más de código
  - Yo no sé qué es lo que produce el compilador cuando realizo un task o un function

Ejemplo task

Implementación de Mux16:1 usando task

```
module  mux16to1 (W, S16, f);
    input  [0:15] W;
    input  [3:0] S16;
    output  reg  f;

    always @(W, S16)
        case (S16[3:2])
            0: mux4to1 (W[0:3], S16[1:0], f);
            1: mux4to1 (W[4:7], S16[1:0], f);
            2: mux4to1 (W[8:11], S16[1:0], f);
            3: mux4to1 (W[12:15], S16[1:0], f);
        endcase
```

```
// Task that specifies a 4-to-1 multiplexer
task  mux4to1;
    input  [0:3] X;
    input  [1:0] S4;
    output  reg  g;

    case (S4)
        0: g = X[0];
        1: g = X[1];
        2: g = X[2];
        3: g = X[3];
    endcase
endtask

endmodule
```

Ejemplo function

Implementación de un Mux16:1 usando function

```
module mux16to1 (W, S16, f);
    input [0:15] W;
    input [3:0] S16;
    output reg f;

    // Function that specifies a 4-to-1 multiplexer
    function mux4to1;
        input [0:3] X;
        input [1:0] S4;

        case (S4)
            0: mux4to1 = X[0];
            1: mux4to1 = X[1];
            2: mux4to1 = X[2];
            3: mux4to1 = X[3];
        endcase
    endfunction

    always @(W, S16)
        case (S16[3:2])
            0: f = mux4to1 (W[0:3], S16[1:0]);
            1: f = mux4to1 (W[4:7], S16[1:0]);
            2: f = mux4to1 (W[8:11], S16[1:0]);
            3: f = mux4to1 (W[12:15], S16[1:0]);
        endcase

endmodule
```

Diferencias entre tareas y funciones

Funciones

- Pueden llamar a otras funciones, pero no pueden llamar a otras tareas
- Son ejecutadas instantáneamente, 0 tiempo de simulación
- No se permiten declaraciones de control de tiempo
- Al menos una entrada
- Retornan un puro valor de salida

Tareas

- Pueden llamar a otras tareas y otras funciones
- Pueden ser ejecutadas con tiempo de simulación distinto de 0
- Pueden contener declaraciones de control de tiempo
- Pueden tener número arbitrario de entradas, salidas, o entrada-salida (inout)
- No retornan un valor