Unidad 5. Subsistemas combinacionales

Circuitos Electrónicos Digitales
E.T.S.I. Informática
Universidad de Sevilla

Jorge Juan <jjchico@dte.us.es> 2010-2018

Esta obra esta sujeta a la Licencia Reconocimiento-CompartirIgual 4.0 Internacional de Creative Commons. Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-sa/4.0/ o envíe una carta Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

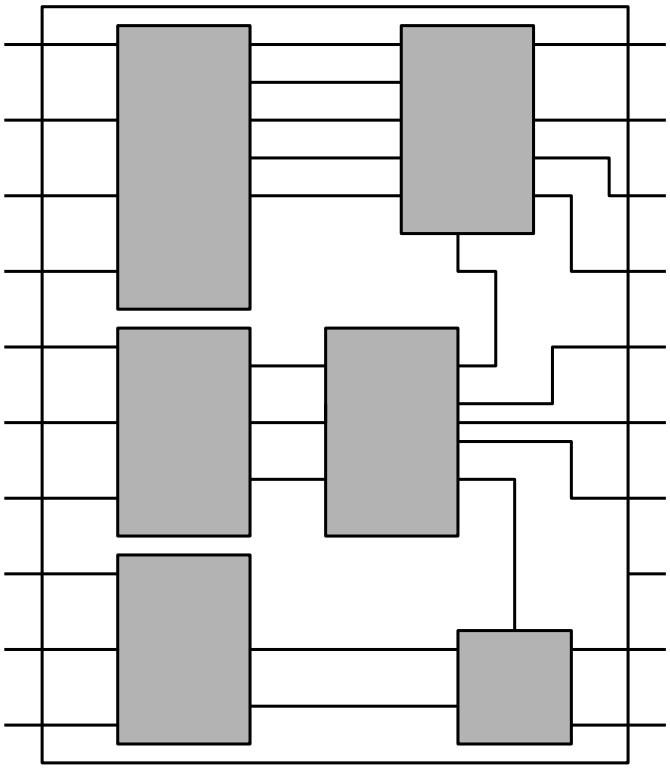
Contenido

- Perspectiva de sistemas
- Características generales de los subsistemas
- Decodificadores
- Multiplexores
- Demultiplexores
- Codificadores
- Matrices de puertas lógicas
- Convertidores de código
- Comparadores
- Detectores/generadores de paridad
- Metodología de diseño con subsistemas

Bibliografía

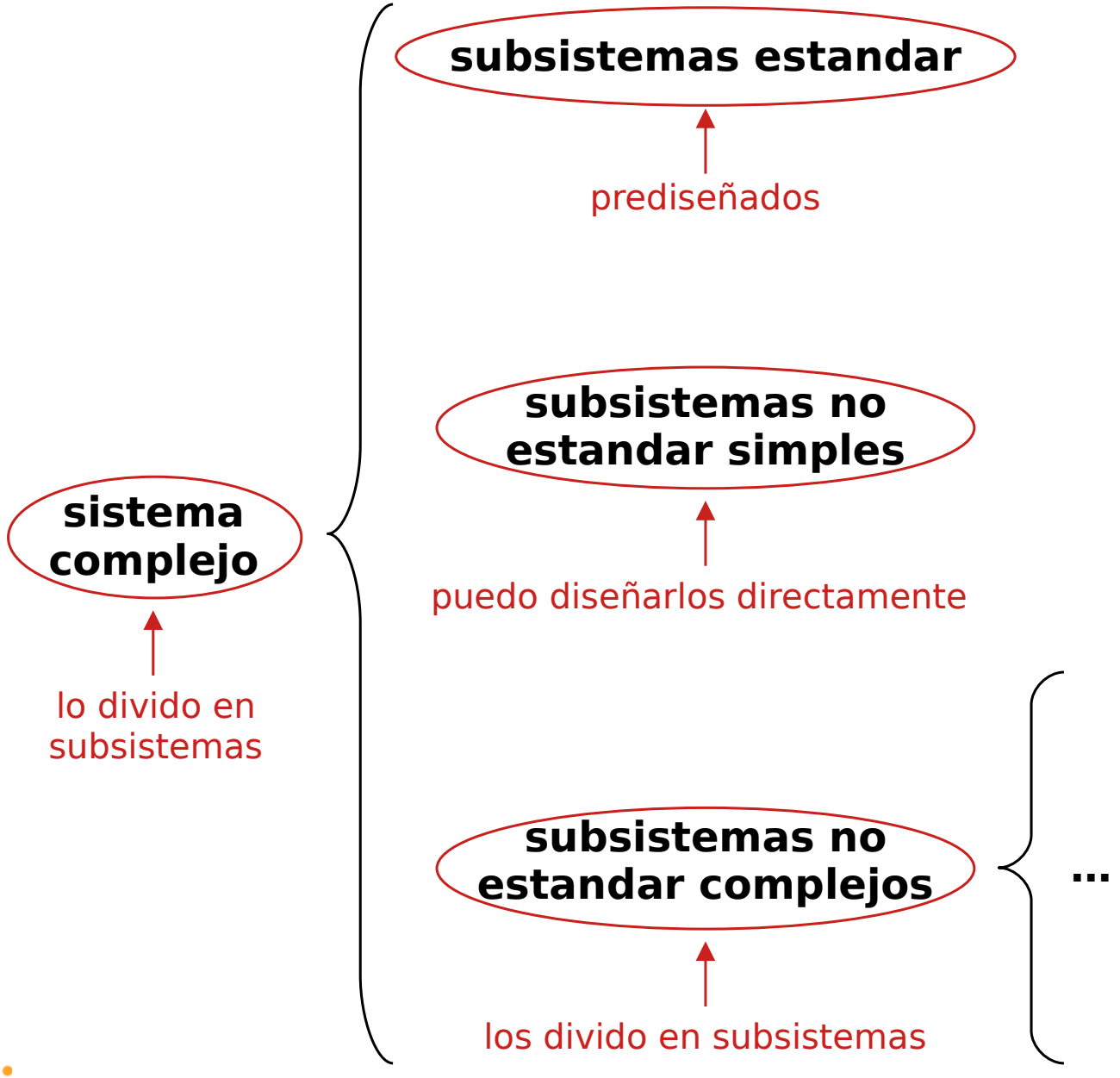
- Bibliografía de referencia. Para resolver dudas, etc.
  - LaMeres, capítulo 6
    - Trata parte del contenido del tema.
    - Ejemplos en Verilog empleando sólo descripciones funcionales (assign).
  - Floyd, 6.4 a 6.10
    - Trata la mayoría de los contenidos del tema.
    - Incluye ejemplos prácticos.
    - Orientado a diseño con dispositivo MSI (74XXX).
  - curso_verilog.v, unidad 4
    - Ejemplos de diseño de subsistemas y con subsistemas combinacionales con posibilidad de simularlos.

Perspectiva de sistemas



- El diseño de un circuito combinacional complejo (con muchas entradas) debe hacerse descomponiéndolo en subsistemas.
  - No es posible aplicar algoritmos de optimización genéricos
  - ¡Divide y vencerás!
- Los subsistemas combinacionales son circuitos combinacionales que hacen funciones estandar útiles para construir muchos circuitos más complejos.
  - Funcionalidad genérica estandar
  - Sirven para implementar muchos sistemas distintos

Metodología de diseño con subsistemas



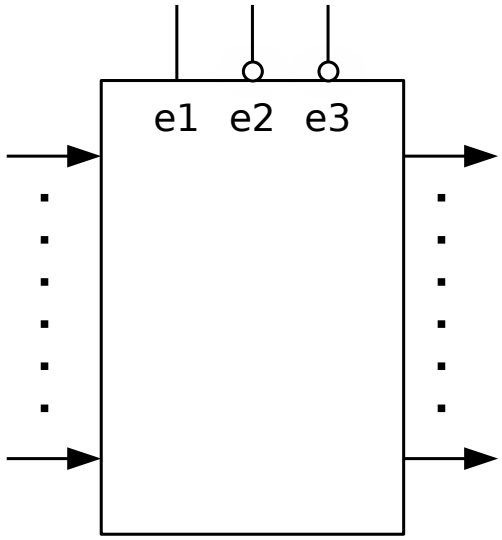
¿dónde podemos encontrarlos?

- Ya implementados en dispositivos MSI (serie 74XX)
  - Formato clásico
  - Chips ya fabricados: pocos bloques con muchas opciones
  - Necesarios para implementar circuitos discretos complejos
- Bibliotecas de diseño de circuitos integrados de aplicación específica (ASIC)
  - Gran variedad de opciones
  - Configurables durante el proceso de diseño
  - Gran cantidad de bloques, opciones a medida
- Primitivas de diseño en lógica reconfigurable (FPGA)
  - Empleados durante la síntesis automática

Características generales de los subsistemas combinacionales

- Muchas entradas y/o salidas binarias
  - Entradas y salidas suelen agruparse en señales multi-bit numeradas (buses)
- Funcionalidad expresada en términos de procesado de datos
  - Multiplexado, codificación, decodificación, comparación, suma, etc.
- Dos tipos de puertos de entrada:
  - Datos. Pueden ser de más de un bit (buses)
  - Control

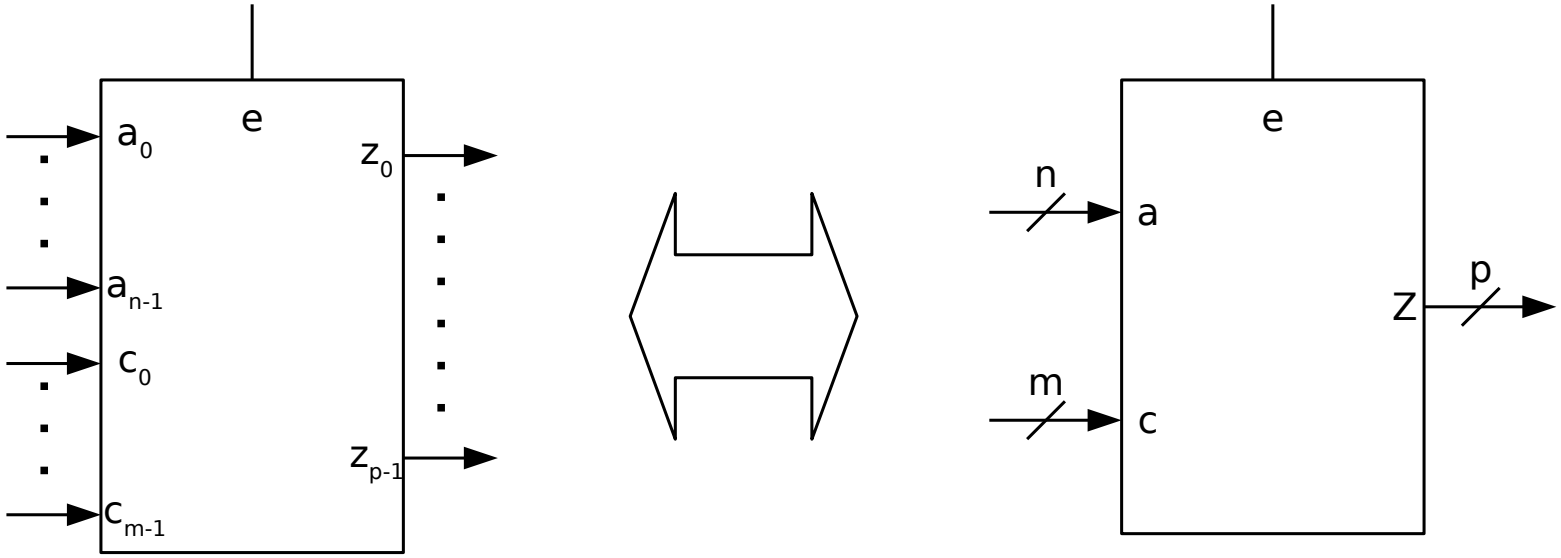
Entradas de control



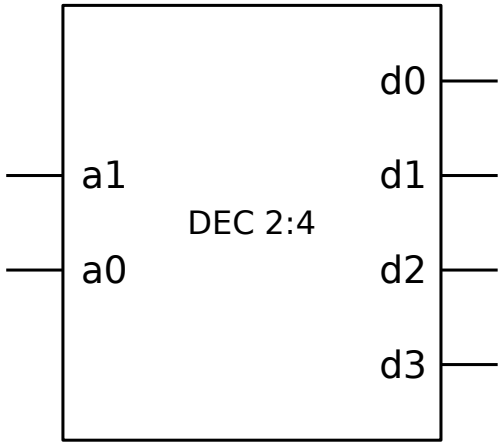
Activo si
e1=1 y e2=0 y e3=0

- Condicionan la operación general del subsistema
  - Habilitación (enable)/ selección (select)
  - Operación a realizar
  - Activación de salida (output enable)
- Tipos de activación/habilitación
  - Activo en bajo: cuando la señal vale 0.
  - Activo en alto: cuando la señal vale 1.

Señales multi-bit (vectores/buses)



Decodificador con salidas activas en alto



a1	a0	d0	d1	d2	d3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Sólo una salida activa para cada combinación de entrada

- n entradas numeradas de 0 a n-1
- 2^n salidas numeradas de 0 a 2^n-1

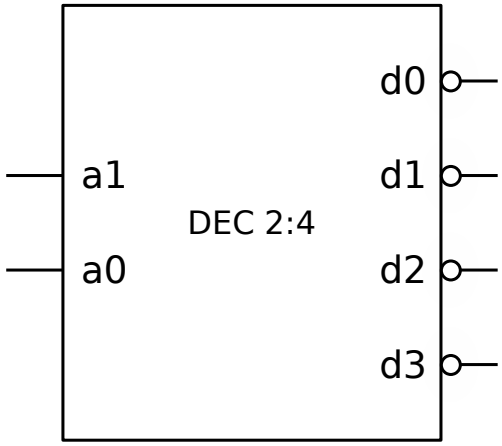
Implementa todos los mintérminos de n variables.

- d0 = m0 = $\overline{a1}\ \overline{a0}$
- d1 = m1 = $\overline{a1}$ a0
- d2 = m2 = a1 $\overline{a0}$
- d3 = m3 = a1 a0

Convertidor de binario natural a código one-hot.

```
module dec4 (
     input wire [1:0] a,
     output reg [3:0] d
     );
always @(a)
     case (a)
          2'h0: d = 4'b0001;
          2'h1: d = 4'b0010;
          2'h2: d = 4'b0100;
          2'h3: d = 4'b1000;
     endcase
endmodule // dec4
```

Decodificador con salidas activas en bajo



a1	a0	d0	d1	d2	d3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Sólo una salida activa para cada combinación de entrada

- n entradas numeradas de 0 a n-1
- 2^n salidas numeradas de 0 a 2^n-1

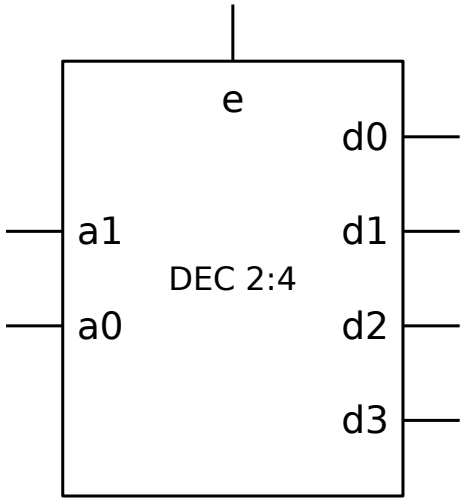
Implementa todos los maxtérminos de n variables.

- d0 = M0 = a1 + $\overline{a0}$
- d1 = M1 = $\overline{a1}$ + $\overline{a0}$
- d2 = M2 = $\overline{a1}$ + a0
- d3 = M3 = $\overline{a1}$ + $\overline{a0}$

Convertidor de binario natural a código “one-cold”.

```
module dec4 (
     input wire [1:0] a,
     output reg [3:0] d
     );
always @(a)
     case (a)
          2'h0: d = 4'b1110;
          2'h1: d = 4'b1101;
          2'h2: d = 4'b1011;
          2'h3: d = 4'b0111;
     endcase
endmodule // dec4
```

Decodificador con habilitación activa en alto y salidas activas en alto



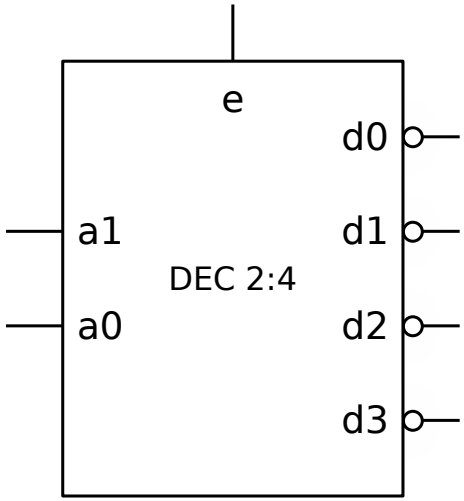
e	a1	a0	d0	d1	d2	d3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Si e (enable) no está activo, ninguna de las salidas se activa.

- d0 = e m0 = e $\overline{a1}$ $\overline{a0}$
- d1 = e m1 = e $\overline{a1}$ a0
- d2 = e m2 = e a1 $\overline{a0}$
- d3 = e m3 = e a1 a0

```
module dec4 (
     input wire [1:0] a,
     input wire e,
     output reg [3:0] d
     );
always @(a, e)
     if (e == 0)
          d = 4'b0000;
     else
          case (a)
               2'h0: d = 4'b0001;
               2'h1: d = 4'b0010;
               2'h2: d = 4'b0100;
               2'h3: d = 4'b1000;
          endcase
endmodule // dec4
```

Decodificador con habilitación activa en alto y salidas activas en bajo



e	a1	a0	d0	d1	d2	d3
0	x	x	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	0	1	1
1	1	0	1	1	0	1
1	1	1	1	1	1	0

Implementa todos los maxtérminos de las variables de entrada (más habilitación).

- $d0 = \overline{e}+M0 = \overline{e}+a1+a0$
- $d1 = \overline{e}+M1 = \overline{e}+a1+\overline{a0}$
- $d2 = \overline{e}+M2 = \overline{e}+\overline{a1}+a0$
- $d3 = \overline{e}+M3 = \overline{e}+\overline{a1}+\overline{a0}$

Convertidor de binario natural a código “one-cold”

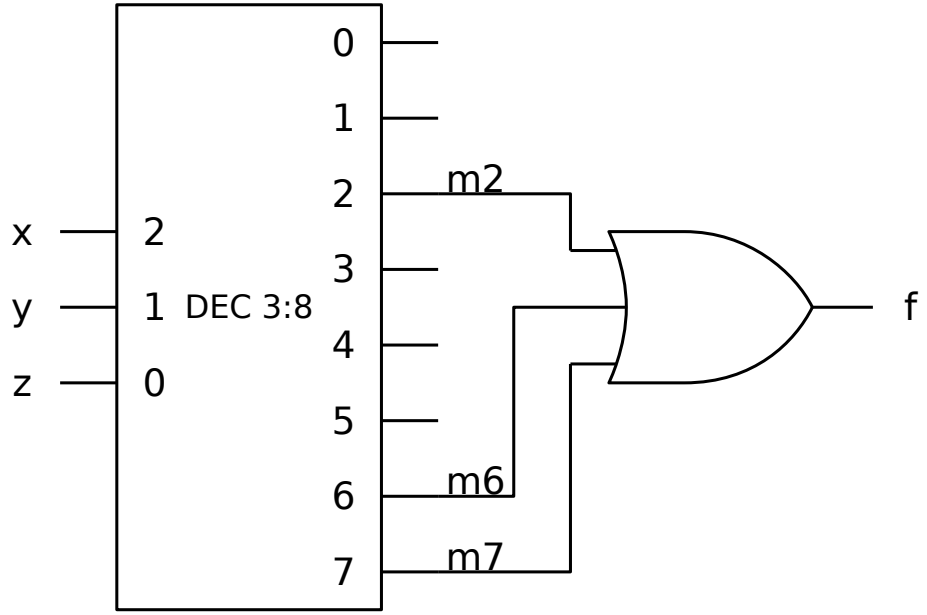
```
module dec4 (
     input wire [1:0] a,
     input wire e,
     output reg [3:0] d
     );
always @(a, e)
     if (e == 0)
          d = 4'b1111;
     else
          case (a)
               2'h0: d = 4'b1110;
               2'h1: d = 4'b1101;
               2'h2: d = 4'b1011;
               2'h3: d = 4'b0111;
          endcase
endmodule // dec4
```

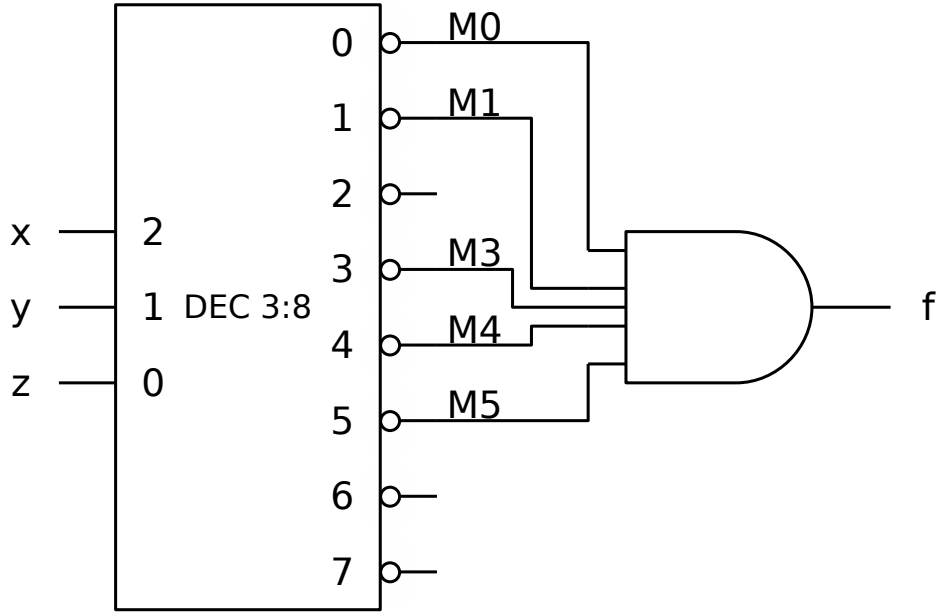
Diseño de decodificadores

- Basta implementar todos los mintérminos/maxtérminos de las variables de entrada
- Señal de habilitación
  - Afecta a todas las salidas por igual.
  - Puede añadirse como una opción post-diseño.
- Ejemplos:
  - Ejemplo 1: DEC 2:4
  - Ejemplo 2: DEC 2:4, activo en bajo con habilitación activa en bajo

Diseño de funciones lógicas con decodificador y puertas

- Los decodificadores permiten implementar de forma sencilla funciones expresadas como suma de mintérminos o producto de maxtérminos
  - Ej: f(x, y, z) = $\sum(2, 6, 7) = \prod(0, 1, 3, 4, 5)$

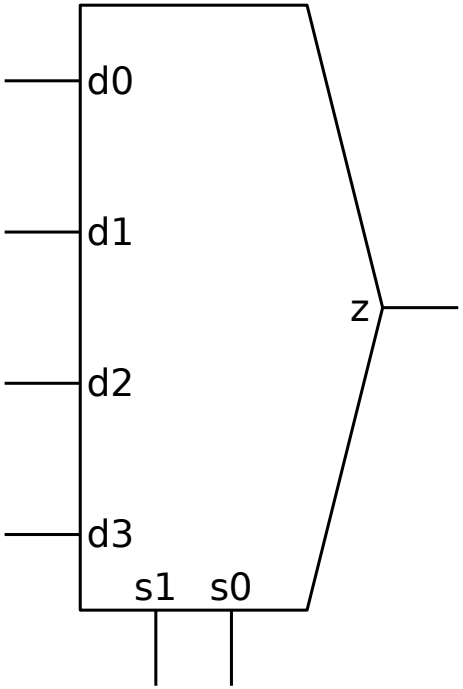




Diseño de funciones lógicas con decodificador: equivalencias

- La estructura DEC-OR es un caso particular de AND-OR
  - DEC-OR es equivalente a $\overline{\text{DEC}}$-NAND igual que AND-OR lo es a NAND-NAND
- La estructura $\overline{\text{DEC}}$-AND es un caso particular de OR-AND
  - $\overline{\text{DEC}}$-AND es equivalente a DEC-NOR igual que OR-AND lo es a NOR-NOR

Multiplexor



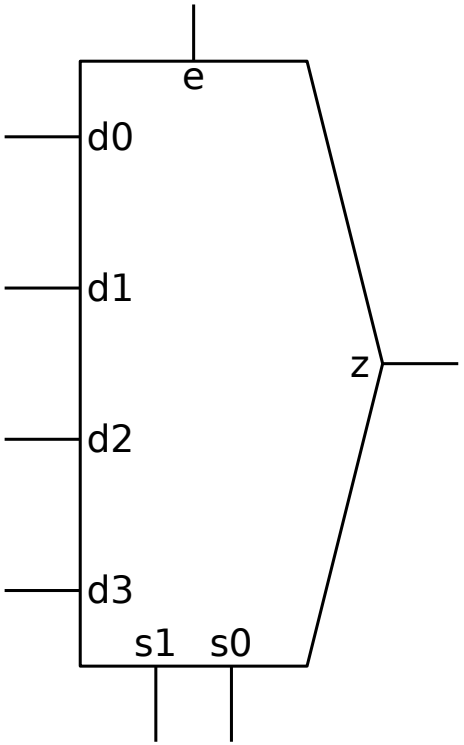
s1	s0	z
0	0	d0
0	1	d1
1	0	d2
1	1	d3

La salida z es igual a la entrada de datos dx seleccionada por las entradas de selección s. $x=s_{(2}$

```
module mux4 (
     input wire [3:0] d,
     input wire [1:0] s,
     output reg z
     );
always @(d, s)
     case (s)
          2'h0: z = d[0];
          2'h1: z = d[1];
          2'h2: z = d[2];
          2'h3: z = d[3];
     endcase
endmodule // mux4
```

$$z = \overline{s1}\ \overline{s0}\ d0 + \overline{s1}\ s0\ d1 + s1\ \overline{s0}\ d2 + s1\ s0\ d3$$

Multiplexor con habilitación



e	s1	s0	z
0	x	x	0
1	0	0	d0
1	0	1	d1
1	1	0	d2
1	1	1	d3

```
module mux4 (
     input wire [3:0] d,
     input wire [1:0] s,
     input wire e,
     output reg z
     );
always @(d, s)
     if (e == 0)
          z = 1'b0;
     else
          case (s)
               2'h0: z = d[0];
               2'h1: z = d[1];
               2'h2: z = d[2];
               2'h3: z = d[3];
          endcase
endmodule // mux4
```

$$z = e\,\overline{s1}\,\overline{s0}\,d0 + e\,\overline{s1}\,s0\,d1 + e\,s1\,\overline{s0}\,d2 + e\,s1\,s0\,d3$$

Diseño de multiplexores

- Opciones de diseño:
  - Como función lógica genérica (K-mapa, etc.): costoso y prohibitivo incluso para pocas entradas.
  - Diseño modular como extensión del decodificador.
- Ejemplo: MUX 4:1 con/sin habilitación

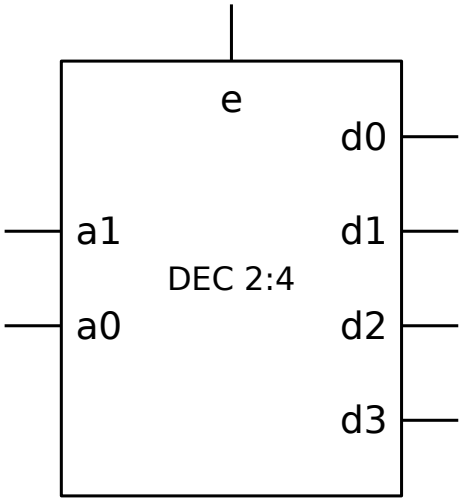
Diseño de funciones lógicas con multiplexores

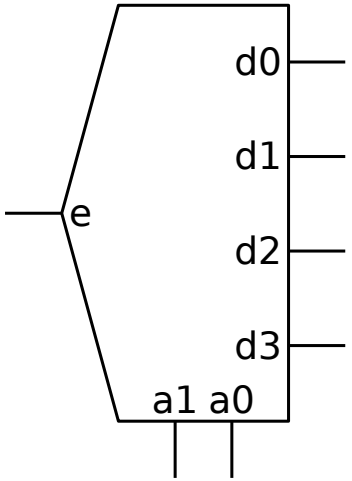
- Ejemplo 1 (con MUX 8:1)
  - $g(x, y, z) = \sum(1, 2, 3, 5, 6)$
- Ejemplo 2 (con MUX 8:1 y MUX 4:1)
  - $f(w, x, y, z) = \sum(0, 1, 2, 6, 7, 8, 12, 14, 15)$
- Se utiliza el Teorema de expansión de Shannon:

$$f(x_{n-1}, \ldots, x_{i+1}, x_i, x_{i-1}, \ldots, x_0) =$$
$$\bar{x}_i\, f(x_{n-1}, \ldots, x_{i+1}, 0, x_{i-1}, \ldots, x_0) + x_i\, f(x_{n-1}, \ldots, x_{i+1}, 1, x_{i-1}, \ldots, x_0)$$

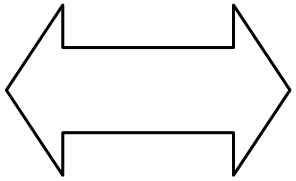
$$f(x_3,x_2,x_1,x_0) =$$
$$\bar{x}_3\, f(0,x_2,x_1,x_0) + x_3\, f(1,x_2,x_1,x_0) =$$
$$\bar{x}_3\bar{x}_2\, f(0,0,x_1,x_0) + \bar{x}_3 x_2 f(0,1,x_1,x_0) + x_3\bar{x}_2 f(1,0,x_1,x_0) + x_3 x_2 f(1,1,x_1,x_0) =$$
$$\ldots$$

Demultiplexores





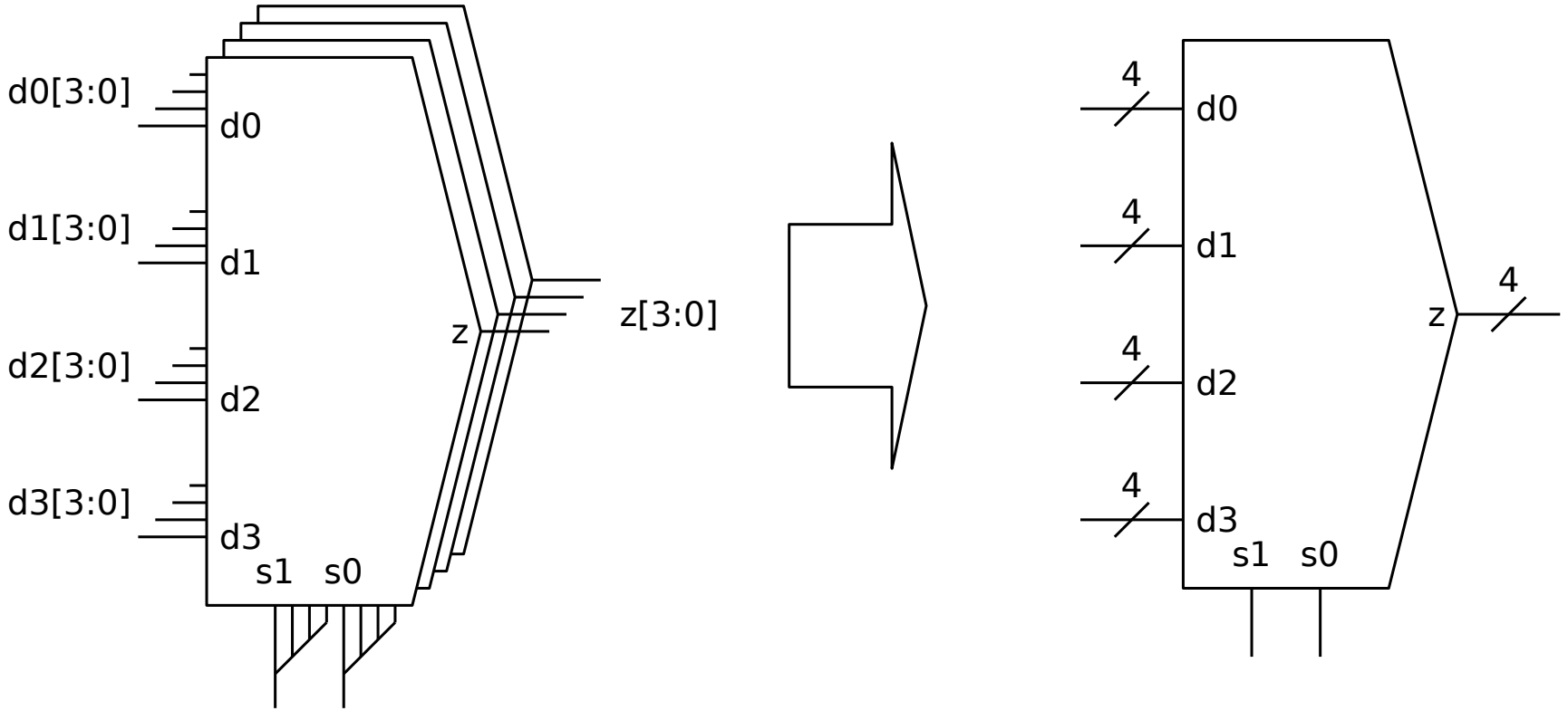
e	a1	a0	d0	d1	d2	d3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



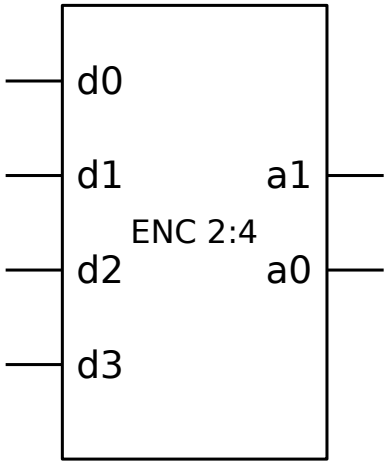
a1	a0	d0	d1	d2	d3
0	0	e	0	0	0
0	1	0	e	0	0
1	0	0	0	e	0
1	1	0	0	0	e

El decodificador con habilitación y el demultiplexor son el mismo circuito

Asociación de MUX en paralelo



Codificadores



d0	d1	d2	d3	a1	a0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Otros valores son inespecificaciones

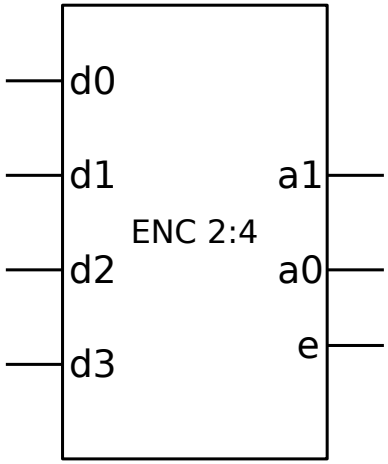
Generan un código binario que identifica la entrada activa.

Las entradas pueden ser activas en alto o bajo

Diferentes formatos de codificación:
- Binario natural
- Código Gray
- Etc.

```
module enc (
     input wire [3:0] d,
     output reg [1:0] a
     );
always @(d)
     case (d)
          4'b0001: a = 2'b00;
          4'b0010: a = 2'b01;
          4'b0100: a = 2'b10;
          4'b1000: a = 2'b11;
          default: a = 2'bxx;
     end
endmodule // enc
```

Codificadores de prioridad



d0	d1	d2	d3	a1	a0	e
0	0	0	0	0	0	1
1	0	0	0	0	0	0
x	1	0	0	0	1	0
x	x	1	0	1	0	0
x	x	x	1	1	1	0

Resuelven el problema de la inespecificaciones de los codificadores simples asignando prioridades a las entradas.
La salida “e” se activa cuando ninguna entrada está activa: no hay nada que codificar.

```
module pri_enc (
     input wire [3:0] d,
     output reg [1:0] a
     );

always @(d)
     if      (d[3]) a = 2'b11;
     else if (d[2]) a = 2'b10;
     else if (d[1]) a = 2'b01;
     else           a = 2'b00;

assign e = ~|d;

endmodule // pri_enc
```

Diseño de codificadores

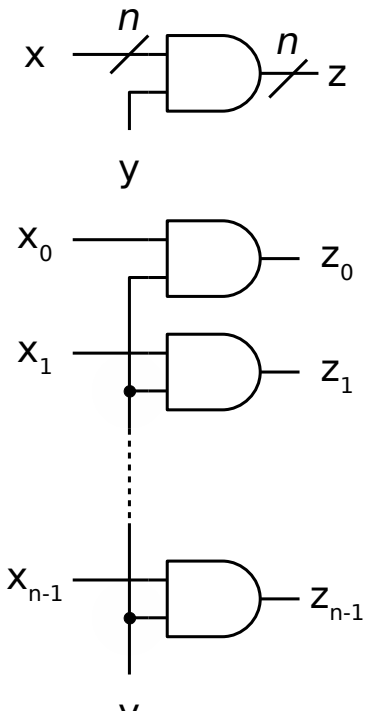
- Opciones de diseño:
  - Como función lógica genérica (K-mapa, etc.): costoso y prohibitivo incluso para pocas entradas.
  - Método específico para codificadores aprovechando la redundancia de su operación (codificadores de prioridad)
- Ejemplos:
  - Ejemplo 1: codificador binario de 4 bits.
  - Ejemplo 2: codificador Gray de 4 bits.
  - Ejemplo 3: codificador de prioridad de 4 bits.

Matrices de puertas como bloques combinacionales

- Linea de control de un bit *Y*
- Buses de datos de *n* bits *X* y *Z*.

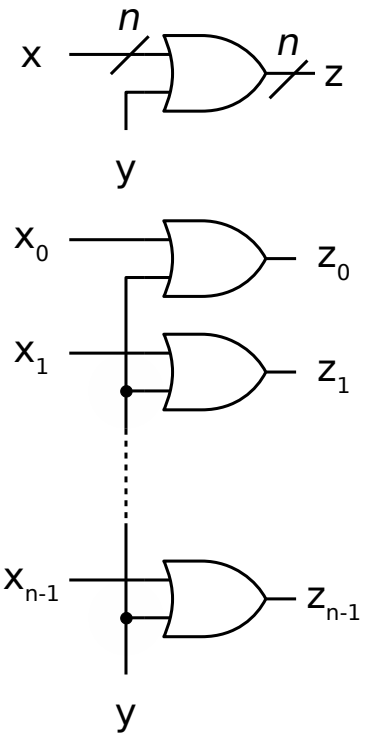
Llave de paso AND

$Z = X\ Y$
$Z = X$ si Y = 1, si no
$Z = 0$



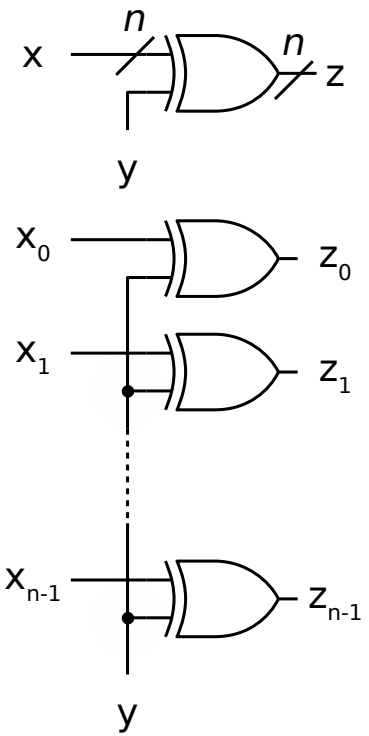
Llave de paso OR

$Z = X + Y$
$Z = X$ si Y = 0, si no
Z = 11..1



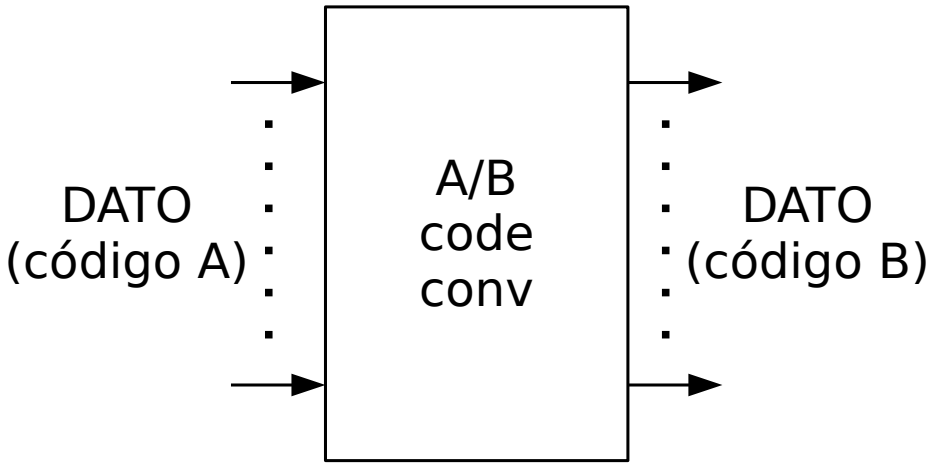
Transfiere-complementa

$Z = X \oplus Y$
$Z = X$ si Y = 0, si no
$Z = \overline{X}$

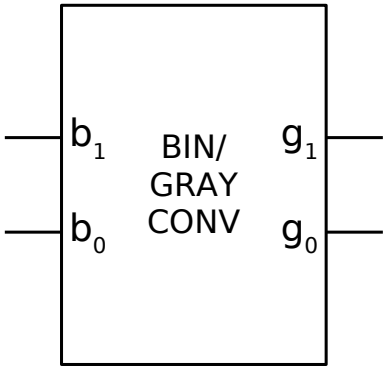


Convertidores de código

- Convierten un dato de un código a otro
- No cambian el dato (información), sólo la representación
  - Binario (natural) a Gray
  - Gray a binario
  - BCD a 7-segmentos
  - ...



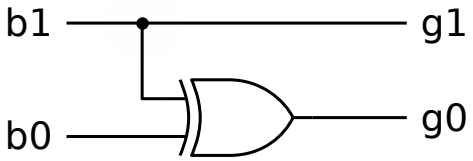
Ej: convertidor bin/gray de 2 bits



b_1	b_0	g_1	g_0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

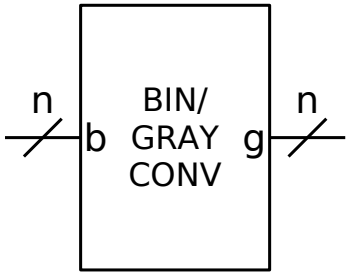
```
module bin_gray2 (
     input wire [1:0] b,
     output reg [1:0] g
     );

always @(b)
     case (b):
          2'b00: g = 2'b00;
          2'b01: g = 2'b01;
          2'b10: g = 2'b11;
          default: g = 2'10;
     end
endmodule // bin_gray_conv
```

$$g_1 = b_1$$
$$g_0 = \overline{b}_1 b_0 + b_1 \overline{b}_0$$
$$g_0 = b_1 \oplus b_0$$




```
module bin_gray2 (
     input wire [1:0] b,
     output wire [1:0] g
     );

assign g[1] = b[1];
assign g[0] = b[1] ^ b[0];

endmodule // bin_gray_conv
```

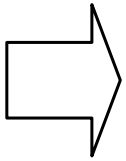
Ej: convertidor bin/gray genérico



Para todo i < n-1:

$g_i = b_i$ si $b_{i+1} = 0$, si no

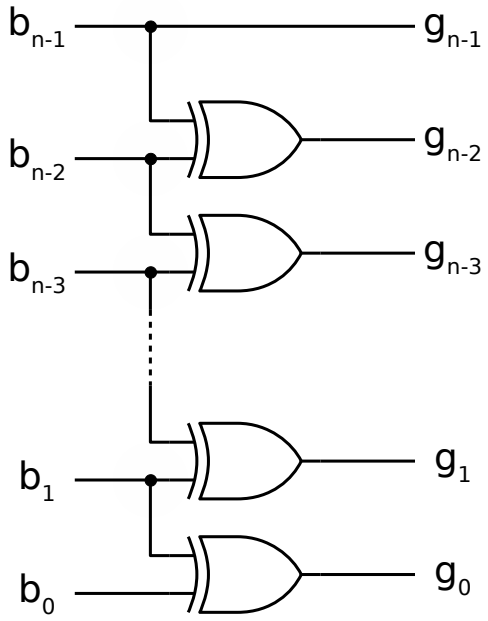
$g_i = \overline{b_i}$



$g_{n-1} = b_{n-1}$

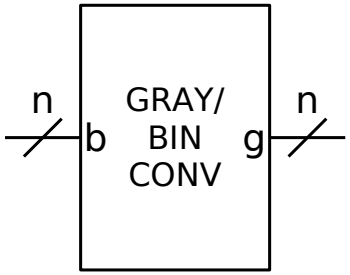
$g_i = b_i \oplus b_{i+1}$

$b_3b_2b_1b_0$	$g_3g_2g_1g_0$
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000




```
module bin_gray #(
  parameter n = 4
)(
  input wire [n-1:0] b,
  output reg [n-1:0] g
);

  integer i;

  always @(*) begin
    g[n-1] = b[n-1];
    for (i=n-2; i>=0; i=i-1)
      g[i] = b[i] ^ b[i+1];
  end
endmodule
```

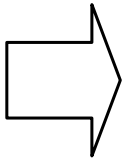
Ej: convertidor gray/bin genérico



Para todo i < n-1:

$b_i = g_i$ si $b_{i+1} = 0$, si no

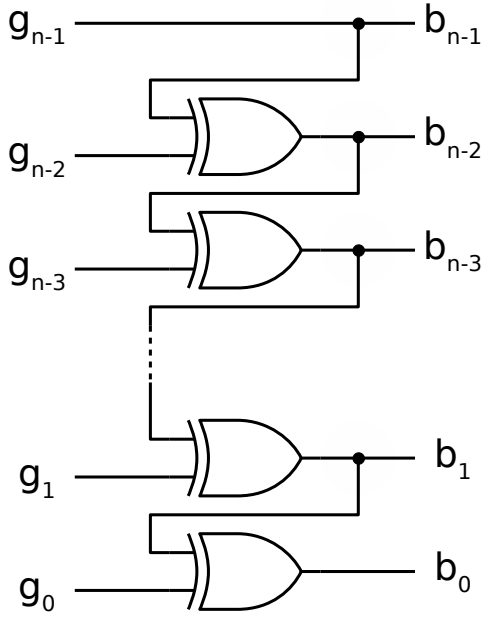
$b_i = \overline{g_i}$



$b_{n-1} = g_{n-1}$

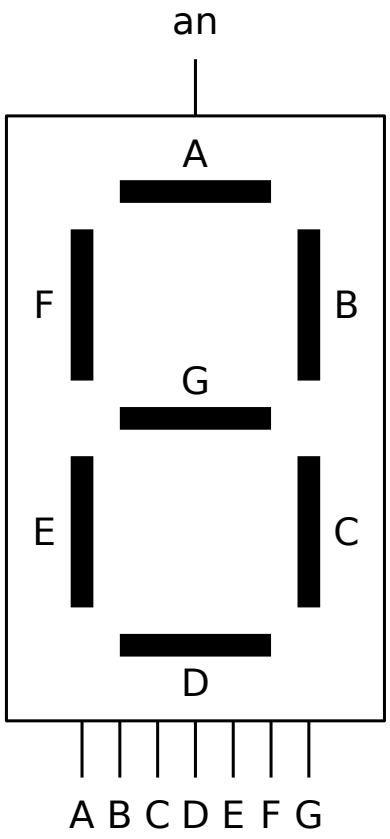
$b_i = g_i \oplus b_{i+1}$

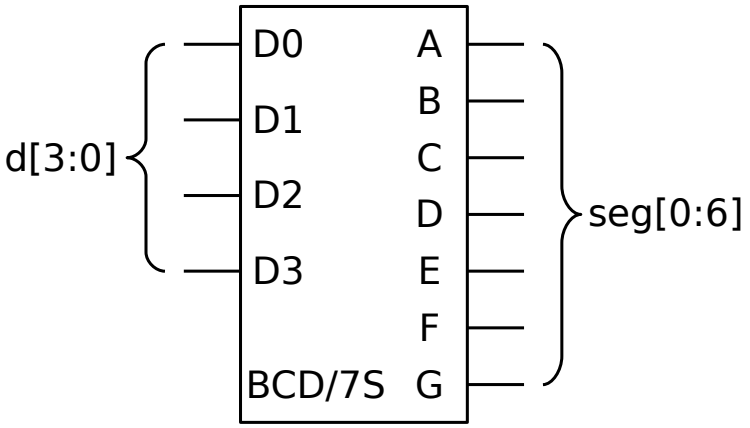
$b_3b_2b_1b_0$	$g_3g_2g_1g_0$
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000




```
module gray_bin #(
  parameter n = 4
)(
  input wire [n-1:0] g,
  output reg [n-1:0] b
);

  integer i;

  always @(*) begin
    b[n-1] = g[n-1];
    for (i=n-2; i>=0; i=i-1)
      b[i] = g[i] ^ b[i+1];
  end
endmodule
```

Convertidor BCD/7-segmentos



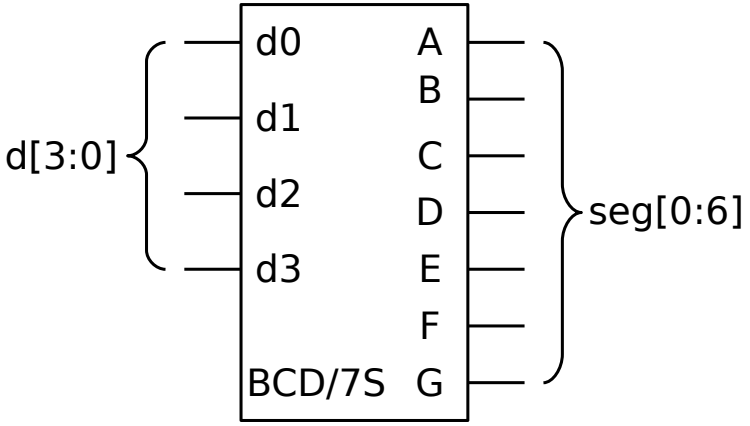


an debe ser '1' para que funcione el *display*

segmentos activos en nivel bajo

$d_3d_2d_1d_0$	d	seg[0:6] ABCDEFG
0000	0	0000001
0001	1	1001111
0010	2	0010010
0011	3	0000110
0100	4	1001100
0101	5	0100100
0110	6	0100000
0111	7	0001111
1000	8	0000000
1001	9	0001100

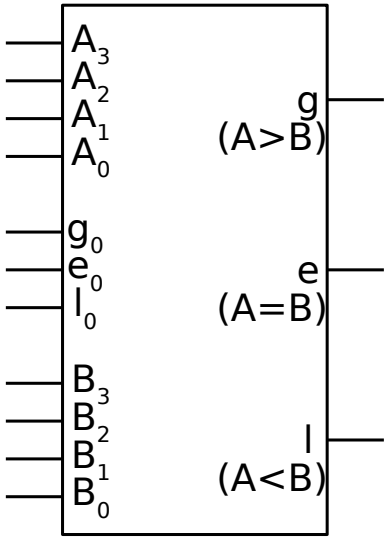
Convertidor BCD/7-segmentos



$d_3d_2d_1d_0$	d	seg[0:6] ABCDEFG
0000	0	0000001
0001	1	1001111
0010	2	0010010
0011	3	0000110
0100	4	1001100
0101	5	0100100
0110	6	0100000
0111	7	0001111
1000	8	0000000
1001	9	0001100

```
module bcd_7s (
     input wire [3:0] d,
     output reg [0:6] seg
     );

always @(b)
     case (d):
          4'h0:    seg = 7'b0000001;
          4'h1:    seg = 7'b1001111;
          4'h2:    seg = 7'b0010010;
          4'h3:    seg = 7'b0000110;
          4'h4:    seg = 7'b1001100;
          4'h5:    seg = 7'b0100100;
          4'h6:    seg = 7'b0100000;
          4'h7:    seg = 7'b0001111;
          4'h8:    seg = 7'b0000000;
          4'h9:    seg = 7'b0001100;
          default: seg = 7'b1111110;
     end
endmodule // bcd_7s
```

Comparadores

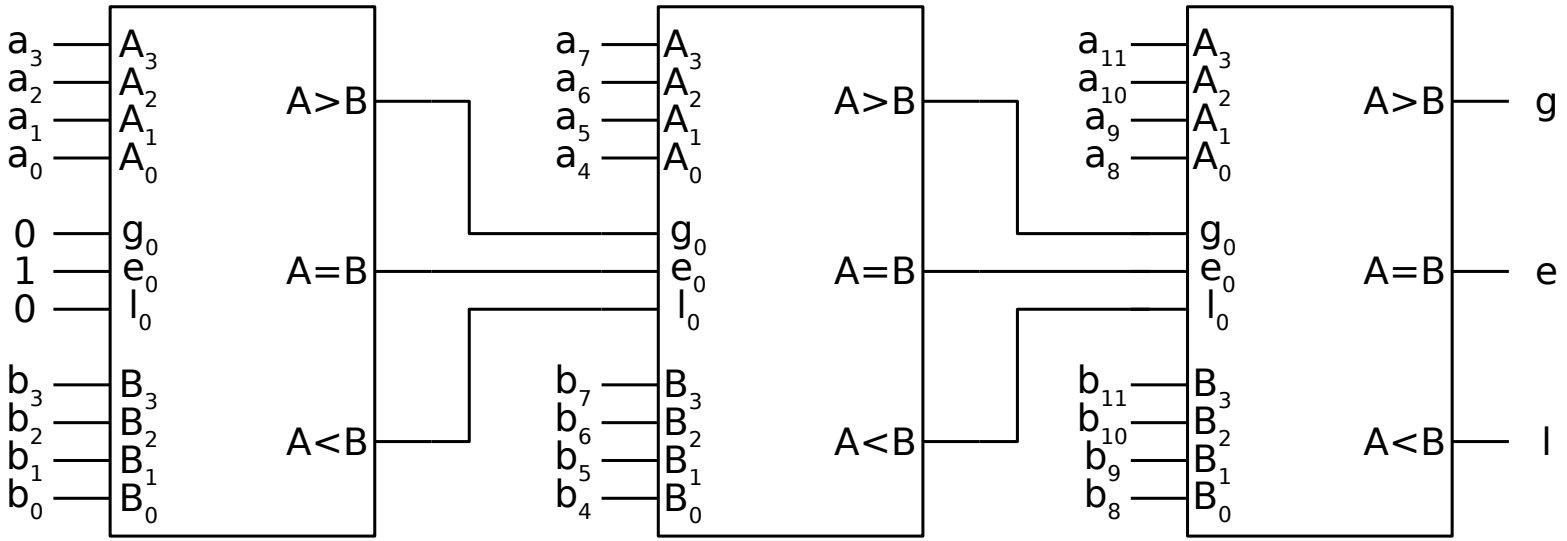


A B	g	e	l
$A_{(2}>B_{(2}$	1	0	0
A=B	g_0	e_0	l_0
$A_{(2}<B_{(2}$	0	0	1

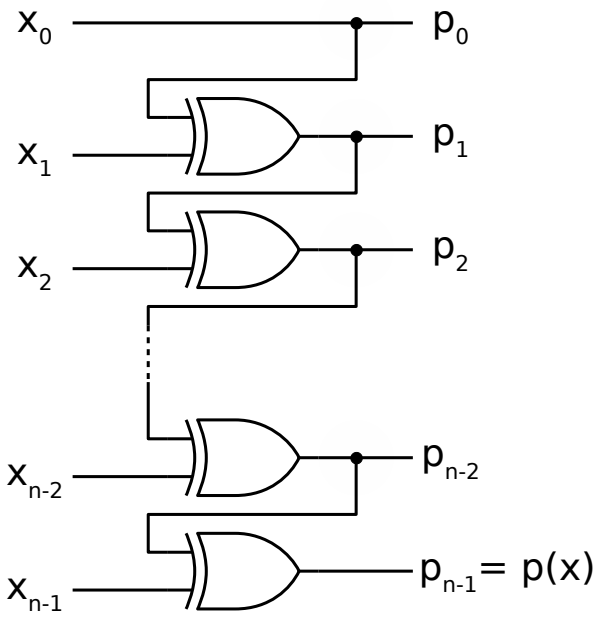
```
module comp4(
  input [3:0] a,
  input [3:0] b,
  input g0, e0, l0,
  output reg g, e, l
);

  always @(*) begin
    if (a > b)
      {g,e,l} = 3'b100;
    else if (a < b)
      {g,e,l} = 3'b001;
    else
      {g,e,l} = {g0,e0,l0};
  end

endmodule
```

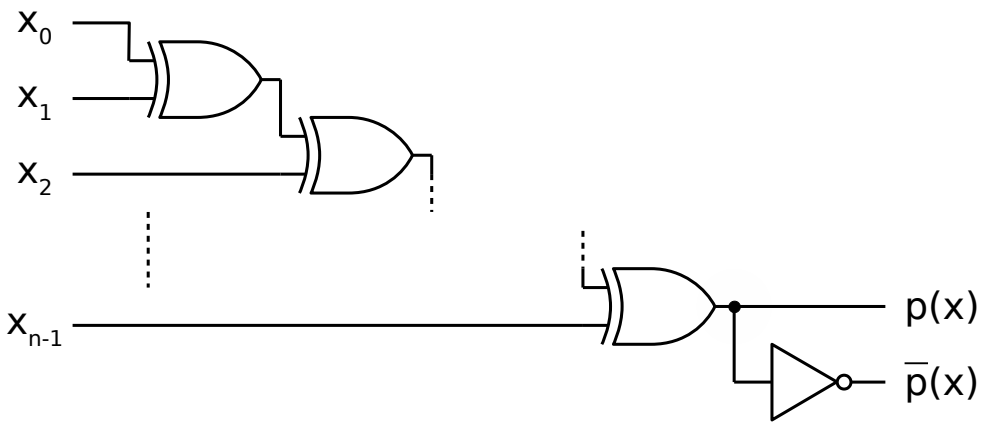
Comparadores

Comparador de 12 bits a partir de comparadores de 4 bits



Detectores/generadores de paridad

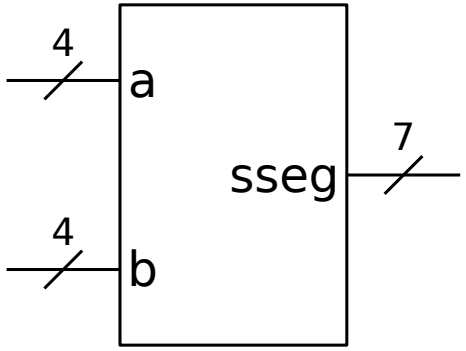




- p(x)
  - Detección de paridad impar (p(x)=1)
  - Generación de bit de paridad par
- $\overline{p}$(x)
  - Detección de paridad par ($\overline{p}$(x)=1)
  - Generación de bit de paridad impar

Ejemplo de diseño 1

- Entradas:
  - a (4bits): temperatura en la habitación A en base 2 (0 to 9).
  - b (4bits): temperatura en la habitación B en base 2 (0 to 9).
- Salidas:
  - sseg (7bits): salida para visor de 7 segmentos que muestra la temperatura más alta de las dos habitaciones.

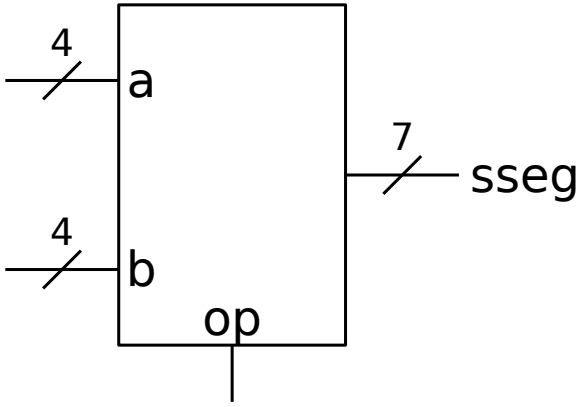


Ejemplo de diseño 6

- Descripción:
  - Diseñe un circuito que monitoriza la temperatura en dos instalaciones y muestra la temperatura mayor y la instalación en la que se produce.
- Entradas:
  - **a** (4 bits): temperatura en la habitación A (0 to 9).
  - **b** (4 bits): temperatura en la habitación B (0 to 9).
- Salidas:
  - **sseg** (7bits): salida para visor de 7 segmentos que muestra la temperatura mayor de ambas.
  - **la** - LED a (1bit): 1 cuando se muestra la temperatura de A
  - **lb** - LED b (1bit): 1 cuando se muestra la temperatura de B

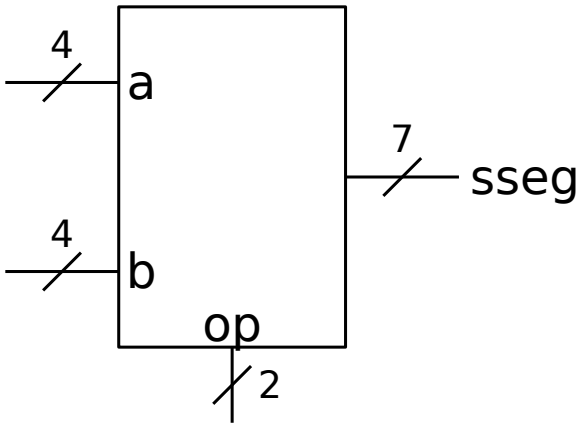
Ejemplo de diseño 3

- Entradas:
  - a (4bits): temperatura en la habitación A (0 to 9).
  - b (4bits): temperatura en la habitación B (0 to 9).
  - op (1bit): entrada de selección.
- Salidas:
  - sseg (7bits): salida para visor de 7 segmentos.
- Descripción
  - El circuito genera el código de 7 segmentos correspondiente a la temperatura más baja de las dos habitaciones si op=0, y a la temperatura más alta si op=1.



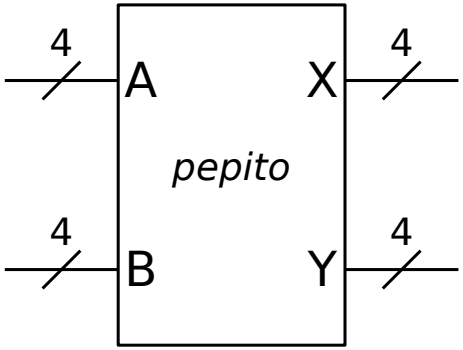
Ejemplo de diseño 4

- Entradas:
  - a (4 bits): temperatura en la habitación A (0 to 9).
  - b (4 bits): temperatura en la habitación B (0 to 9).
  - op (2 bits): entrada de selección.
- Salidas:
  - sseg (7bits): salida para visor de 7 segmentos.
- Descripción
  - El circuito genera el código de 7 segmentos correspondiente a la temperatura más baja de las dos habitaciones si $op_{(2}=0$, a la temperatura más alta si $op_{(2}=1$, a la temperatura A si $op_{(2}=2$ y a la temperatura B si $op_{(2}=3$.



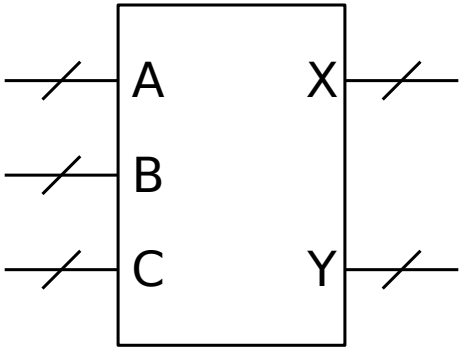
Ejemplo de diseño 2

- Un circuito *pepito* recibe por sus entradas A y B dos números. Su salida X debe representar el mayor de ambos, mientras que su salida Y debe representar el menor de ambos. Todas las entradas y salidas son de cuatro bits y representan los valores en binario natural. Diseñe el circuito usando subsistemas combinacionales y puertas lógicas.



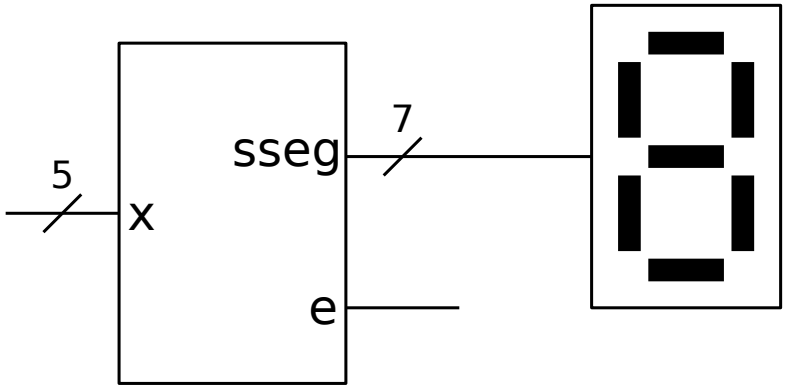
Ejemplo de diseño 5

- Diseñe un circuito que toma tres números de 4 bits A, B y C como entradas y proporciona como salida dos números de 4 bits X e Y tales que X es el mayor de A, B y C, e Y es el menor de A, B y C. Diseñe y dibuje el circuito usando el módulo *pepito* diseñado en el ejemplo 2 como bloque básico.



Ejemplo 7

- Un sistema recibe un dato en paridad par por una entrada X de 5 bits. El bit más significativo es el de paridad. El dato es un número entre 0 y 9 codificado en binario natural. Diseñe un circuito que compruebe la paridad y muestre el dato en un visor de 7 segmentos. Una salida de error “e” se activa si se detecta un error de paridad o si el número no está entre 0 y 9.
- Modifique el diseño para que en caso de error se muestre el número cero en el visor de 7 segmentos.



Ejercicios

https://www.dte.us.es/docencia/etsii/gii-ti/cedti/problemas/problemas5/view