
Subsistemas combinacionales

DAPA
E.T.S.I. Informática
Universidad de Sevilla
Octubre, 2015

Jorge Juan <jjchico@dte.us.es> 2010-2015

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.



Departamento de Tecnología Electrónica - Universidad de Sevilla

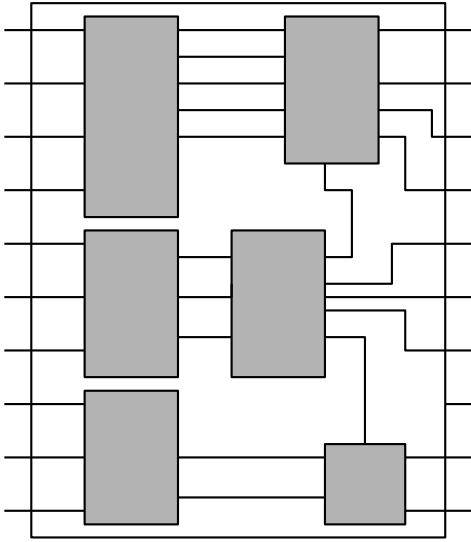
Contenido

- 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



Departamento de Tecnología Electrónica - Universidad de Sevilla

Perspectiva de sistemas



- Los subsistemas combinacionales son circuitos combinacionales que hacen funciones útiles de propósito general
- Muchos problemas prácticos son más fáciles de resolver dividiéndolos en problemas más sencillos y aplicando subsistemas
 - ¡Divide y vencerás!
- Necesario en problemas con muchas entradas y/o salidas.
 - No es posible aplicar algoritmos de optimización genéricos

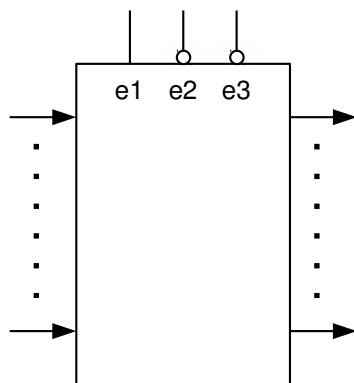
¿Dónde se encuentran?

- En dispositivos SSI/MSI (serie 74XX)
 - Formato clásico
 - Chips ya fabricados: pocos bloques con muchas opciones
 - Necesarios para implementar circuitos discretos complejos
- Bibliotecas de circuitos integrados (ASIC)
 - Gran variedad de opciones
 - Configurables durante el proceso de diseño
 - Gran cantidad de bloques, opciones a medida
- Primitivas de configuración en FPGA
 - Generados automáticamente durante la síntesis automática

Características generales de los subsistemas combinacionales

- Muchas entradas y/o salidas binarias
 - Entradas y salidas suelen formar señales multi-bit
- Modularidad
 - Funcionalidad similar con número de entradas/salidas variable.
 - Diseño modular: diseño para un bit y extensión a n bits.
- Funcionalidad expresada en términos de procesado de datos
 - Multiplexado, codificación, decodificación, comparación, etc.
- Dos tipos de puertos de entrada/salida:
 - Datos
 - Control

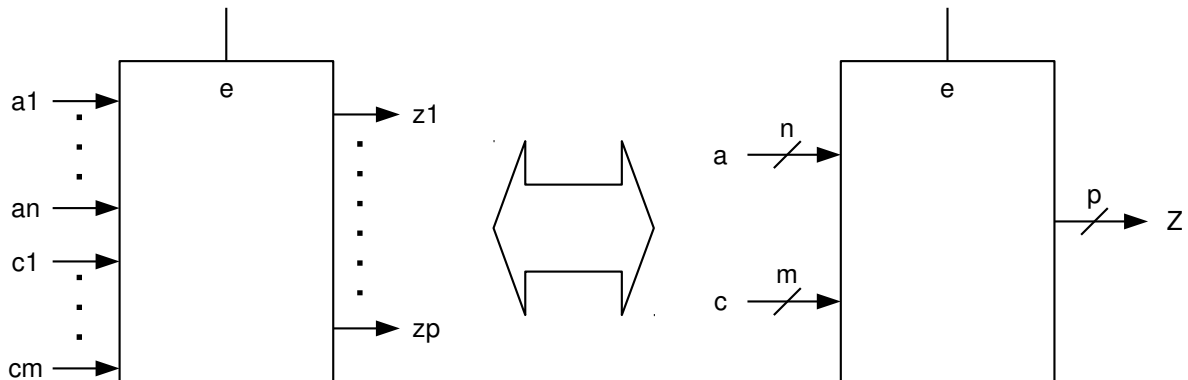
Señales de control



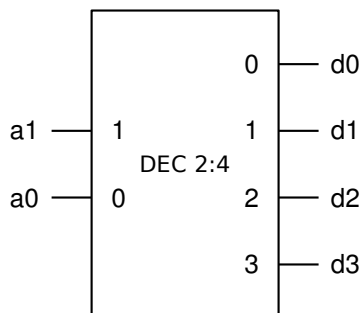
Activo si
 $e1=1$ y $e2=0$ y $e3=0$

- Condicionan la operación general del subsistema
 - Habilitación (enable)
 - Activación de salida (output enable)
 - Selección (select)
 - Etc.
- Tipos de activación
 - Activo en bajo: cuando la señal vale 0.
 - Activo en alto: cuando la señal vale 1.

Señales multi-bit (vectores/buses)



Decodificador



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
- 2^n salidas

Implementa todos los mintermos de las variables de entrada.

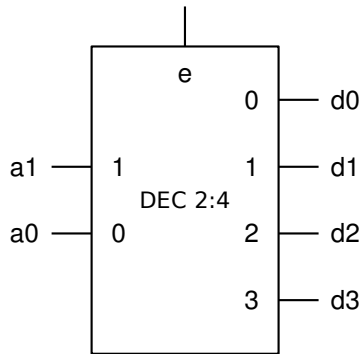
- $d_0 = m_0 = \bar{a}_1 \bar{a}_0$
- $d_1 = m_1 = \bar{a}_1 a_0$
- $d_2 = m_2 = a_1 \bar{a}_0$
- $d_3 = m_3 = a_1 a_0$

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 habilitación



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

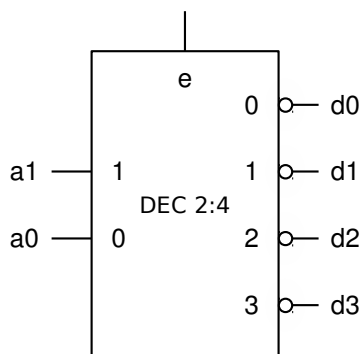
Si e (enable) no está activo, ninguna de las salidas se activa.

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

```

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. 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).

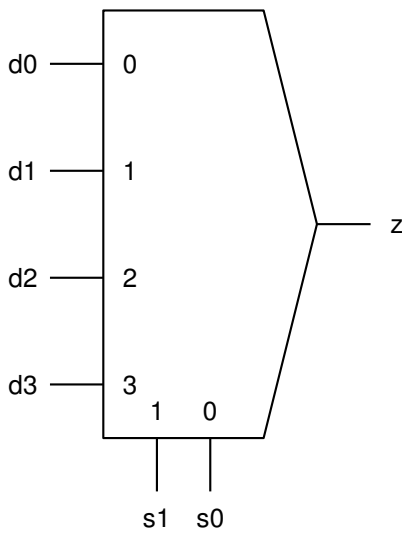
- $d0 = \overline{e} + M0 = \overline{e} + a1 + a0$
- $d1 = \overline{e} + M1 = \overline{e} + a1 + a0$
- $d2 = \overline{e} + M2 = \overline{e} + a1 + a0$
- $d3 = \overline{e} + M3 = \overline{e} + a1 + a0$

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
    
```

Multiplexor



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

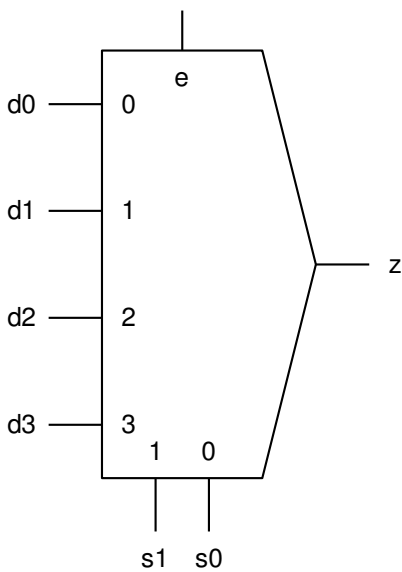
La salida z es igual a la entrada de datos dx seleccionada por las entradas de selección sx

```

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
    
```

$$z = \bar{s}_1 \bar{s}_0 d_0 + \bar{s}_1 s_0 d_1 + s_1 \bar{s}_0 d_2 + s_1 s_0 d_3$$

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
    
```

$$z = e \bar{s}_1 \bar{s}_0 d_0 + e \bar{s}_1 s_0 d_1 + e s_1 \bar{s}_0 d_2 + e s_1 s_0 d_3$$

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

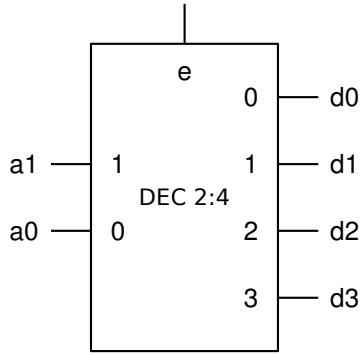
Diseño de funciones lógicas con multiplexores

- Ejemplo 1 (con MUX 8:1)
 - $f(x, y, z) = \Sigma(2, 3, 6, 7)$
- Ejemplo 2 (con MUX 8:1 y MUX 4:1)
 - $f(w, x, y, z) = \Sigma(0, 1, 2, 6, 7, 8, 12, 14, 15)$

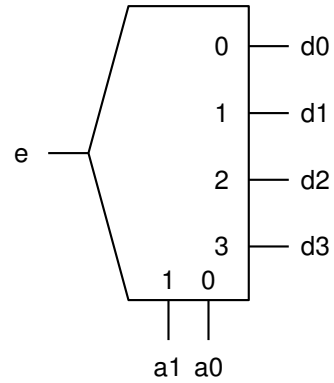
$$f(x_1, \dots, x_i, \dots, x_n) = \bar{x}_i f(x_1, \dots, 0, \dots, x_n) + x_i f(x_1, \dots, 1, \dots, x_n)$$

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= \\ \bar{x}_1 f(0, x_2, x_3, x_4) + x_1 f(1, x_2, x_3, x_4) &= \\ \bar{x}_1 \bar{x}_2 f(0, 0, x_3, x_4) + \bar{x}_1 x_2 f(0, 1, x_3, x_4) + x_1 \bar{x}_2 f(1, 0, x_3, x_4) + x_1 x_2 f(1, 1, x_3, x_4) &= \\ \dots & \end{aligned}$$

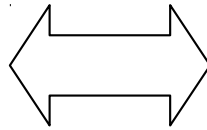
Demultiplexores



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

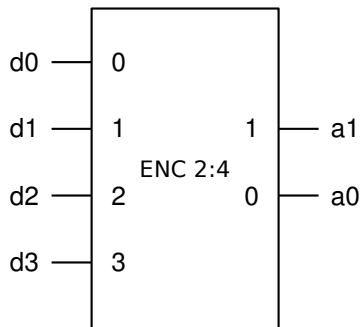


a1	a0	d0	d1	d2	d3
0	0	e	0	0	0
0	1	0	e	0	0
1	0	0	0	e	0
1	1	0	0	0	e



El decodificador con habilitación y el demultiplexor son el mismo circuito

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

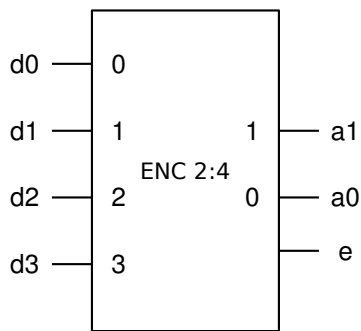
Diferentes formatos de codificación:

- 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
    
```

Matrices de puertas como bloques combinacionales

Llave de paso AND

$$z = x y$$

$z = x$ si $y = 1$, si no
 $z = 0$

Llave de paso OR

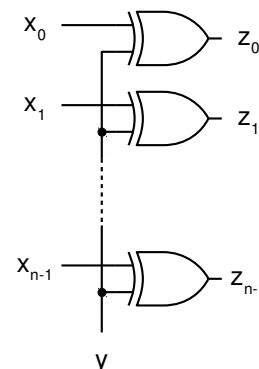
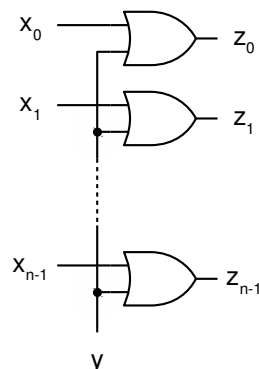
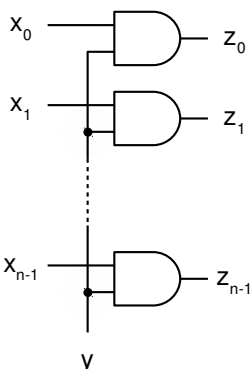
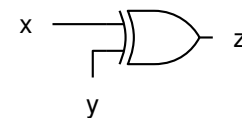
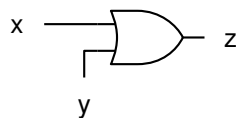
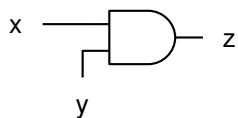
$$z = x + y$$

$z = x$ si $y = 0$, si no
 $z = 1$

Transfiere-complementa

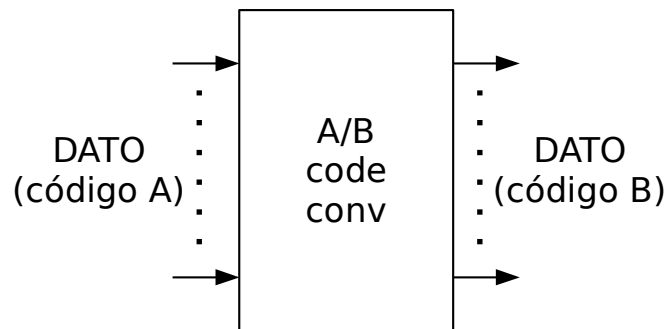
$$z = x \oplus y$$

$z = \overline{x}$ si $y = 0$, si no
 $z = \overline{\overline{x}}$

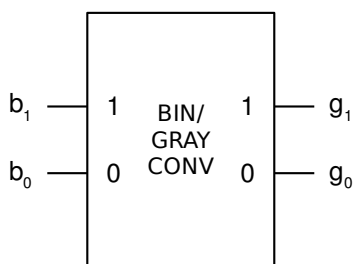


Convertidores de código

- 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
 - ...

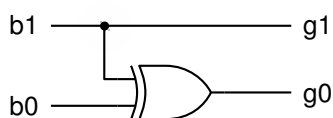


Ej: convertidor bin/gray de 2 bits



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

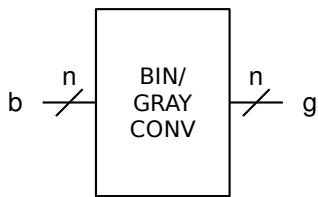
$$\begin{aligned}g_1 &= b_1 \\g_0 &= \bar{b}_1 b_0 + b_1 \bar{b}_0 \\g_0 &= b_1 \oplus b_0\end{aligned}$$



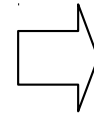
```
module bin_gray2 (  
    input wire [1:0] b,  
    output reg [1:0] g  
);  
  
always @(b)  
    case (b):  
        2'b00: g = 2'b00;  
        2'b01: g = 2'b01;  
        2'b10: g = 2'b11;  
        default: g = 2'10;  
    end  
endmodule // bin_gray_conv
```

```
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

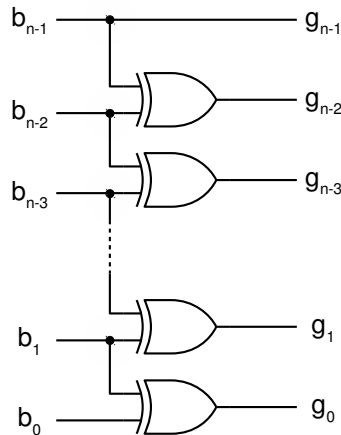


Para todo $i < n-1$:
 $g_i = b_i$ si $b_{i+1} = 0$, si no
 $g_i = \bar{b}_i$



$$\begin{aligned} g_{n-1} &= b_{n-1} \\ g_i &= b_i \oplus b_{i+1} \end{aligned}$$

$b_3 b_2 b_1 b_0$	$g_3 g_2 g_1 g_0$
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



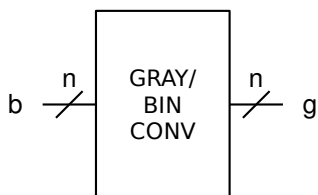
```

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

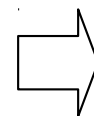
integer i;

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

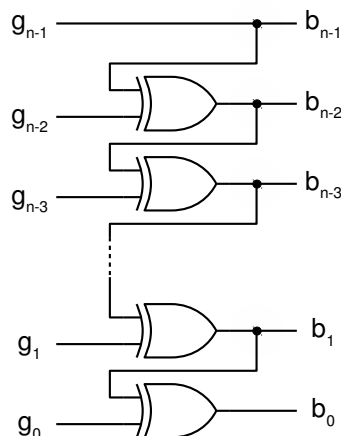


Para todo $i < n-1$:
 $b_i = g_i$ si $b_{i+1} = 0$, si no
 $b_i = \bar{g}_i$



$$\begin{aligned} b_{n-1} &= g_{n-1} \\ b_i &= g_i \oplus b_{i+1} \end{aligned}$$

$b_3 b_2 b_1 b_0$	$g_3 g_2 g_1 g_0$
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



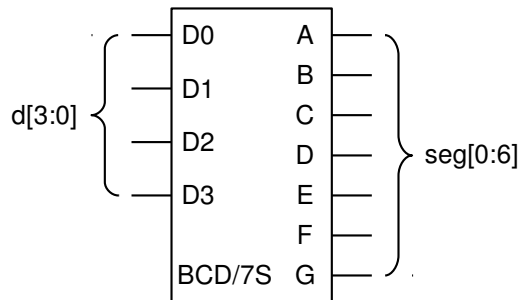
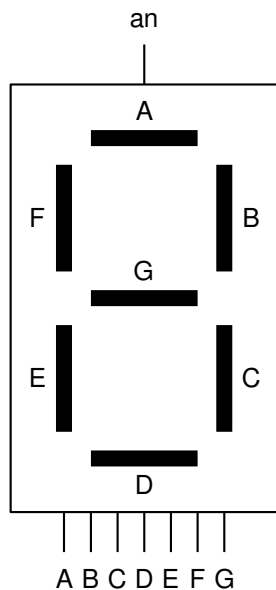
```

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
    
```

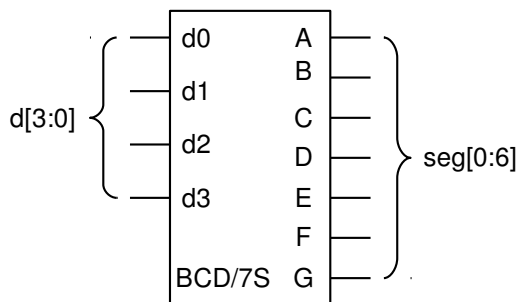
Convertidor BCD/7-segmentos



$d_3d_2d_1d_0$	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

an debe ser '1' para que funcione el *display*
segmentos activos en nivel bajo

Convertidor BCD/7-segmentos



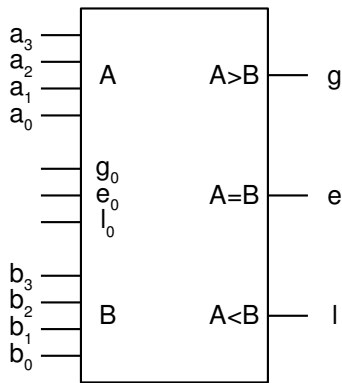
$d_3d_2d_1d_0$	d	seg[0:6] ABCDEFG
0000	0	0000001
0001	1	1001111
0010	2	0010010
0011	3	0000110
0100	4	1001100
0101	5	0100100
0110	6	0100000
0111	7	0001111
1000	8	0000000
1001	9	0001100

```

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



```

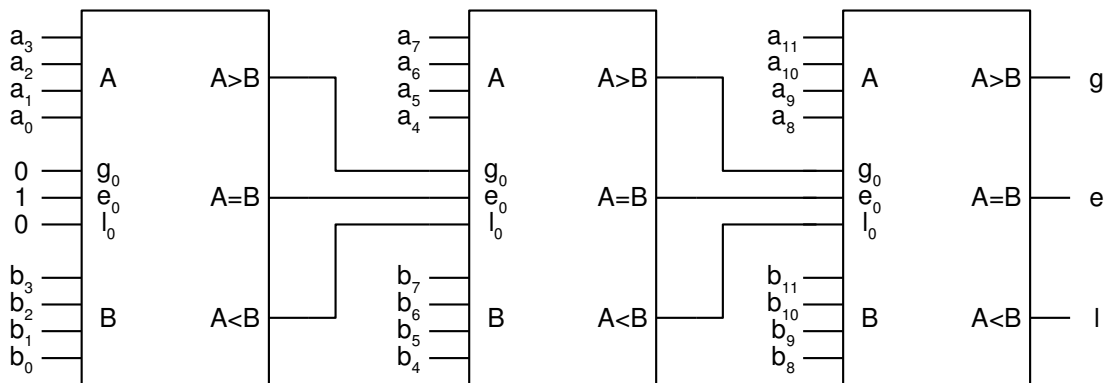
module comp4(
    input [3:0] a,
    input [3:0] b,
    input g0, e0, l0,
    output reg g, e, l
);

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
    
```

A B	g	e	l
A > B	1	0	0
A = B	g ₀	e ₀	l ₀
A < B	0	0	1

Comparadores

Comparador de 12 bits a partir de comparadores de 4 bits



Detectores/generadores de paridad

Definición: Dada una palabra x de n bits, x_0 hasta x_{n-1} , se define la paridad hasta el bit i -ésimo de x , p_i , tal que:

$p_i = 0$ si el número de bits a 1 desde x_0 a x_{i-1} es par.

$p_i = 1$ si el número de bits a 1 desde x_0 a x_{i-1} es impar.

Teorema: $p_0 = x_0$.

Teorema: $p_i = p_{i-1}$ si $x_i = 0$; $p_i = \bar{p}_{i-1}$ si $x_i = 1$.

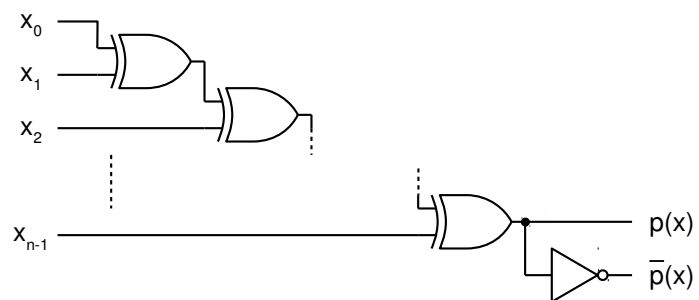
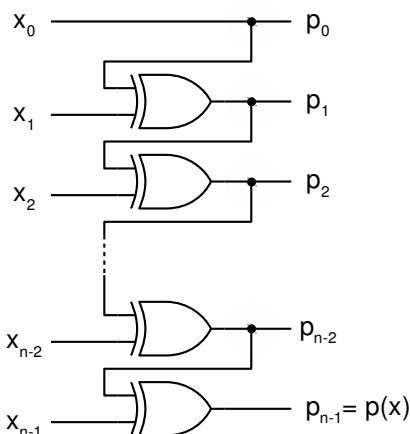
Corolario: $p_i = x_i \oplus p_{i-1}$.

Definición: Dada una palabra x de n bits, x_0 hasta x_{n-1} , se define la paridad de la palabra x , $p(x)$, como la paridad hasta el bit $n-1$ -ésimo de x .

Teorema: Una palabra x de n bits aumentada con su bit de paridad produce una palabra de $n+1$ bits de paridad par.

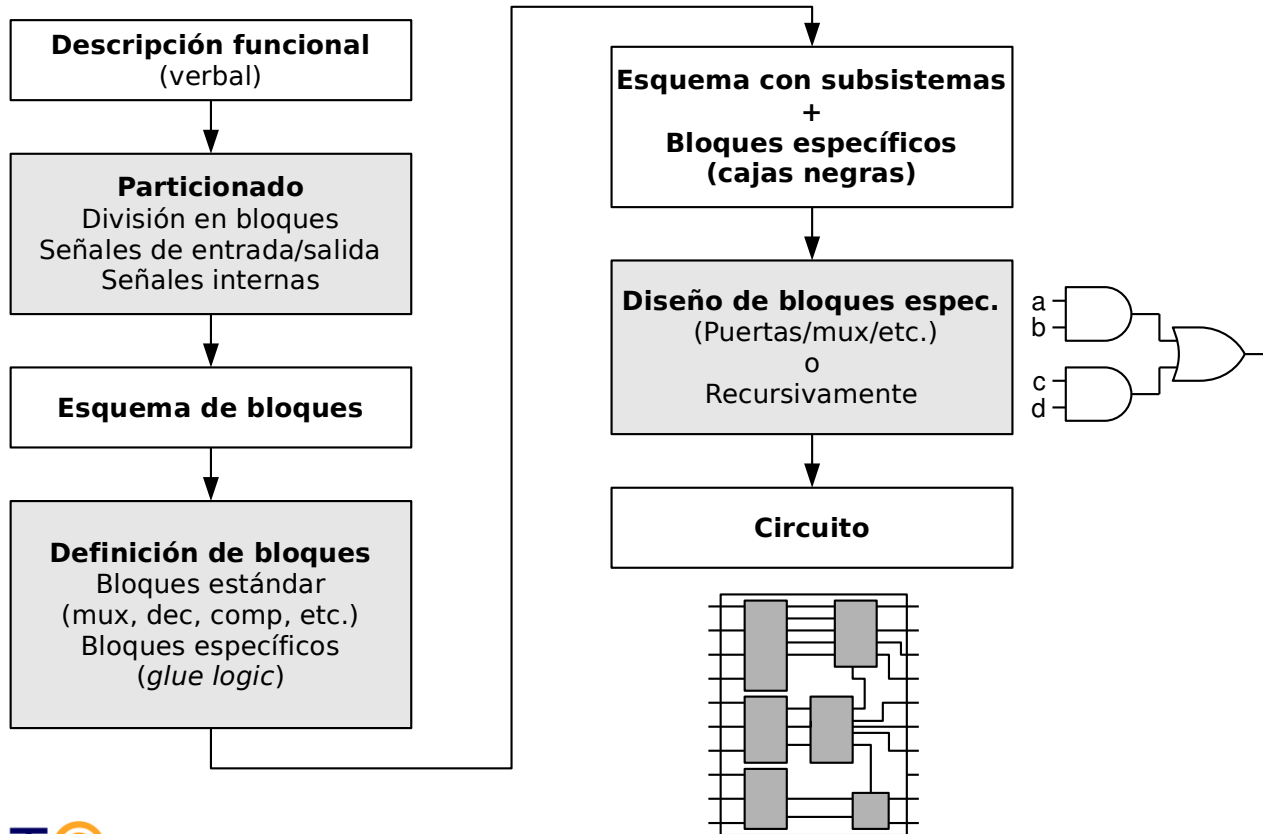
Teorema: Una palabra x de n bits aumentada con el complemento de su bit de paridad produce una palabra de $n+1$ bits de paridad impar.

Detectores/generadores de paridad



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

Metodología de diseño con subsistemas



Ejemplo de diseño

- Entradas:
 - a (4bits): temperatura en la habitación A (0 to 9).
 - b (4bits): temperatura en la habitación B (0 to 9).
 - m (1bit): selector mín/máx (0-mín, 1-máx)
- Salidas:
 - sseg (7bits): salida para visor de 7 segmentos que muestra la temperatura de la habitación seleccionada.
 - la - LED a (1bit): 1 cuando se muestra la temperatura de A
 - lb - LED b (1bit): 1 cuando se muestra la temperatura de B
- Descripción
 - Si $m = 0$, muestra la temperatura más baja de A y B. Se activa la si la temperatura más baja es en A y activa lb en caso contrario.
 - If $m = 1$, muestra la temperatura más alta de A y B. Se activa la si la temperatura más alta es en A y activa lb en caso contrario.