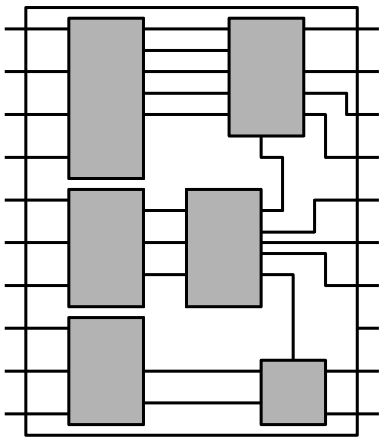
Digital Circuits II

DAPA
E.T.S.I. Informática
Universidad de Sevilla
10/2012

Contents

Subsystem perspective

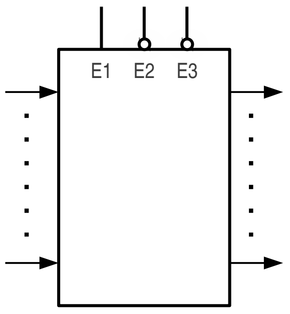


- Blocks make digital systems easier to design
  - Divide and conker!
- Combinational subsystems are combinational blocks that make general purpose useful combinational functions
- Many practical problems are solved more easily by splitting and mapping to subsystems
- Specially interesting when problems have many inputs or outputs
  - General purpose boolean minimization is not feasible

General characteristic

- Many binary inputs and/or outputs
  - Many inputs/outputs work together: multi-bit signals
- Modularity
  - Similar functionality, number of inputs/outputs may change
  - Modular design: subsystems are designed by thinking on one bit and extending to n bits.
- Functionality expressed in terms of data processing:
  - multiplexing, decoding, encoding, …
- Two types of ports:
  - Data
  - Control

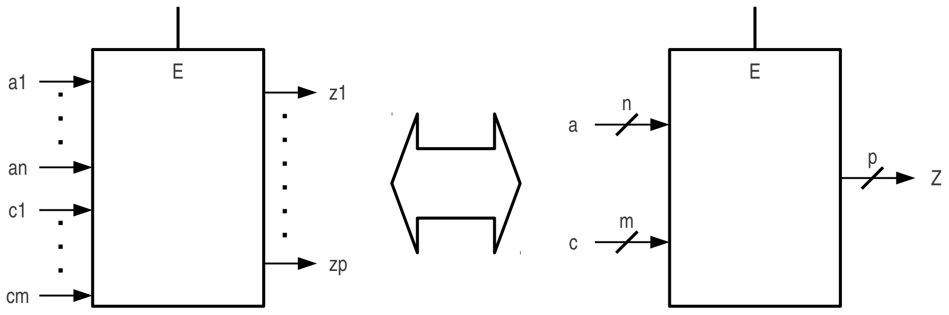
Control signals



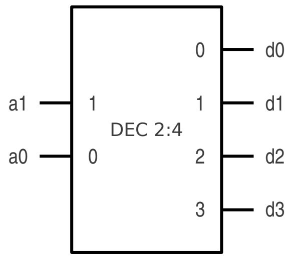
Enabled if
E1=1 & E2=0 & E3=0

- Put a condition on the overall operation of the subsystems
  - Enable
  - Output enable
  - Select
  - …
- Active low
  - signal is active when low (0)
- Active high
  - signal is active when high (1)

Multi-bit (vector) signals



Decoder



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

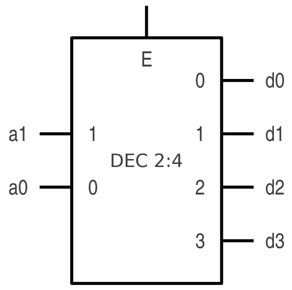
Only one active output for each input vector
- n inputs
- 2^n outputs

Implement all the minterms of the input variables
- $d0 = m0 = \overline{a1}\ \overline{a0}$
- $d1 = m1 = \overline{a1}\ a0$
- $d2 = m2 = a1\ \overline{a0}$
- $d3 = m3 = a1\ a0$

Natural binary to one-hot code converter

```
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
```

Decoder with enable

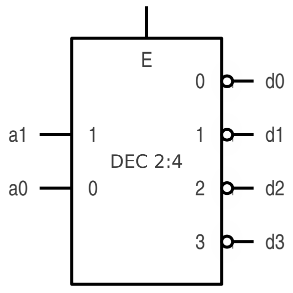


E	a1	a0	d0	d1	d2	d3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

```
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
```

If E (enable) is not active, none of the outputs is active.

Decoder with enable. Active low



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

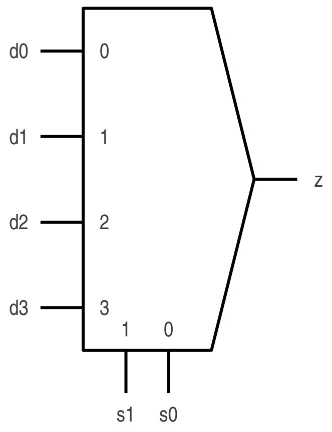
```
module dec4 (
    input wire [1:0] a,
    input wire e,
    output reg [3:0] d
    );
always @(a, E)
    if (E == 0)
        d = 4'b0000;
    else
        case (a)
            2'h0: d = 4'b1110;
            2'h1: d = 4'b1101;
            2'h2: d = 4'b1011;
            2'h3: d = 4'b0111;
        endcase
endmodule // dec4
```

Implement all the maxterms of the input variables

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

Natural binary to one-cold code converter

Multiplexer



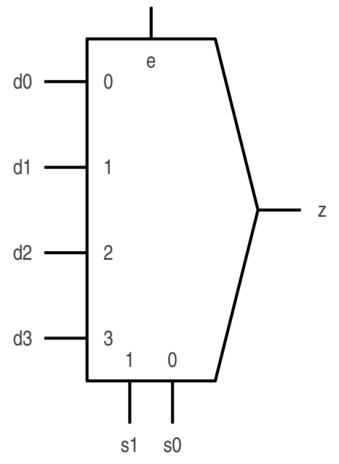
s1	s0	z
0	0	d0
0	1	d1
1	0	d2
1	1	d3

Output z is equal to the data input (dx) selected by the selection inputs (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
```

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

Multiplexer with enable

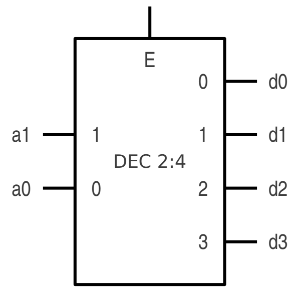


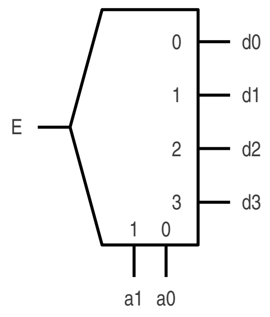
e	s1	s0	z
0	x	x	0
1	0	0	d0
1	0	1	d1
1	1	0	d2
1	1	1	d3

```
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
```

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

Demultiplexer





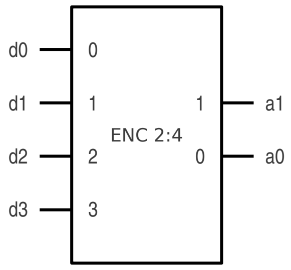
E	a1	a0	d0	d1	d2	d3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

$\Leftrightarrow$

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

The decoder (with enable) and the demultiplexer are the same circuit

Encoders



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

Other values are “don't cares”

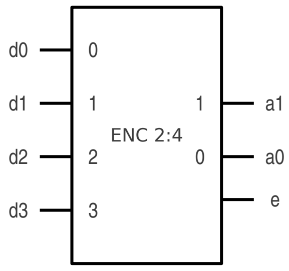
Encoders output the number of the input that is active.

Inputs can be active low or high.

Output value can be “encoded” in different forms:
- Natural binary
- 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
```

Priority encoders



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

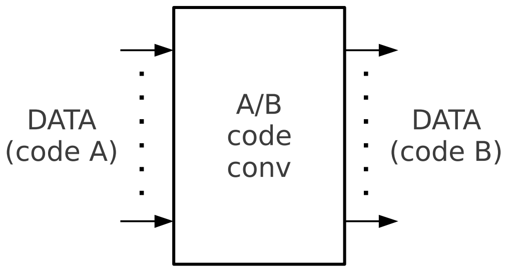
Priority encoders solve the problem of having “don't cares” by using different priorities for the inputs

Output “e” activates when no input is active: there is nothing to encode.

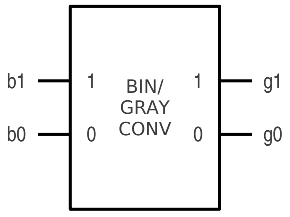
```
module pri_enc (
    input wire [3:0] d,
    output reg [1:0] a
    );

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
```

Code converters

- Convert information from one encoding to another
  - (Natural) binary to Gray
  - Gray to binary
  - BCD/7-segment
  - ...

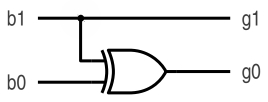


Example 2-bit Bin/Gray converter



b1	b0	g1	g0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

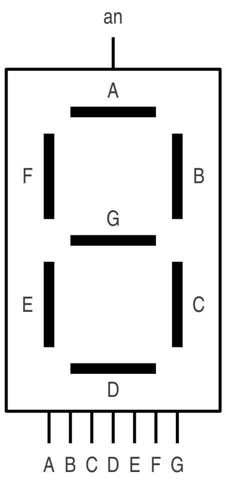
$$\boxed{\begin{array}{l} g1 = b1 \\ g0 = b1 \oplus b0 \end{array}}$$

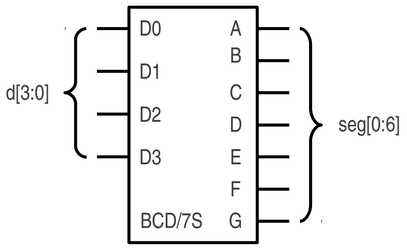


```
module bin_gray_conv (
    input wire [1:0] b,
    output reg [1:0] g
    );

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_gray_conv (
    input wire [1:0] b,
    output reg [1:0] g
    );

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

endmodule // bin_gray_conv
```

BCD/7-segment converter

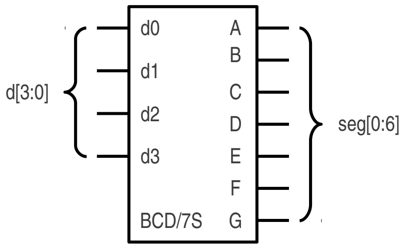




an should be '0' for the device to work.

$d_3d_2d_1d_0$	d	seg[0:6] ABCDEFG
0000	0	0000001
0001	1	1001111
0011	2	0010010
0010	3	0000110
0110	4	1001100
0111	5	0100100
0101	6	0100000
0100	7	0001111
1100	8	0000000
1101	9	0001100

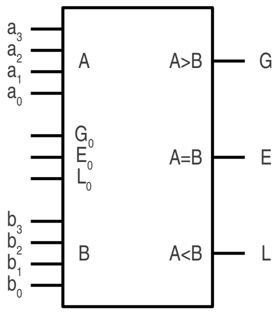
BCD/7-segment converter



$d_3d_2d_1d_0$	d	seg[0:6] ABCDEFG
0000	0	0000001
0001	1	1001111
0011	2	0010010
0010	3	0000110
0110	4	1001100
0111	5	0100100
0101	6	0100000
0100	7	0001111
1100	8	0000000
1101	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
```

Comparators

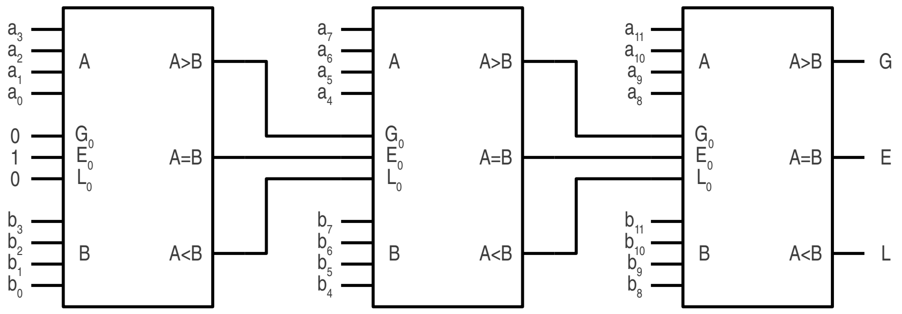


A B	G	E	L
A>B	1	0	0
A=B	G_0	E_0	L_0
A<B	0	0	1

```
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
```

Comparators

12-bit comparator out of 4-bit comparators



a_3
a_2
a_1
a_0
0
1
0
b_3
b_2
b_1
b_0
a_7
a_6
a_5
a_4
b_7
b_6
b_5
b_4
a_{11}
a_{10}
a_9
a_8
b_{11}
b_{10}
b_9
b_8
G_0
E_0
L_0
A
B
A>B
A=B
A<B
G
E
L