Unit 4. Combinational Subsystems

Digital Electronic Circuits
E.T.S.I. Informática
Universidad de Sevilla

Jorge Juan-Chico <jjchico@dte.us.es> 2010-2020

Contents

- Subsystem perspective: blocks
- Subsystem general characteristics
- Decoders
- Multiplexers
- Demultiplexers
- Encoders
- Logic gates as pass through blocks
- Code converters
- Comparators
- Parity detectors/generators
- Design methodology with combinational subsystems
- Design examples
- Combinational subsystems and HDLs

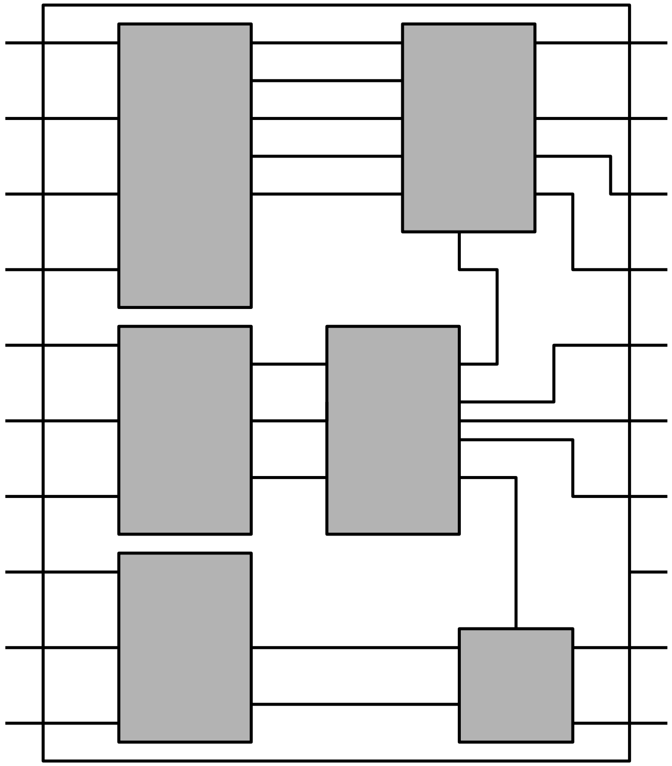
Bibliography

- Reference bibliography
  - LaMeres, chapter 6
    - Includes part of the unit's content.
    - Verilog examples using functional descriptions only (assign).
  - Floyd, 6.4 to 6.10
    - Includes most the unit's content.
    - Good practical examples.
    - Very much oriented to design with MSI devices (74xx).
  - verilog_course.v, unit 4
    - Combinational subsystems design examples and sample designs using subsystems.

Recommended extra exercises

- Exercises from the course's collection 3 (in Spanish)
  - Function design with decoders: 4, 7, 21
  - Function design with multiplexers: 8, 9, 10, 22
  - Comparators: 17

Subsystem perspective



- Combinational subsystems are combinational circuits that implement useful general-purpose functions.
- Many practical problems are solved more easily by splitting and mapping to subsystems
  - Divide and conker!
- Necessary when problems have many inputs or outputs
  - General purpose boolean minimization is not feasible

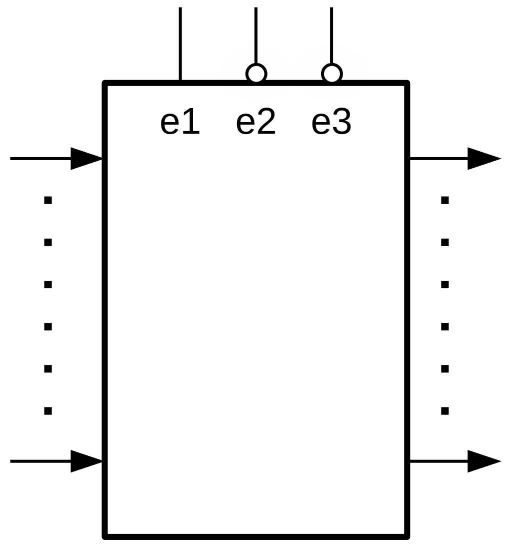
Where can I find them?

- In MSI devices (74XX series)
  - Traditional library of combinational subsystems.
  - Very flexible operation: chips are already fabricated but can be adapted to multiple applications.
  - Useful to design complex discrete digital circuits.
- Integrated circuits standard libraries (ASIC)
  - Same goal than MSI devices but for IC's.
  - Can be configured during the design process.
  - Great variety of options.
- FPGA configuration primitives
  - Standard configuration generated during the logic synthesis.

General characteristics of combinational subsystems

- 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: same functionality no matter the number of bits.
- Functionality expressed in terms of data processing:
  - multiplexing, decoding, encoding, comparing, …
- Two types of input/output ports:
  - Data
  - Control

Control signals

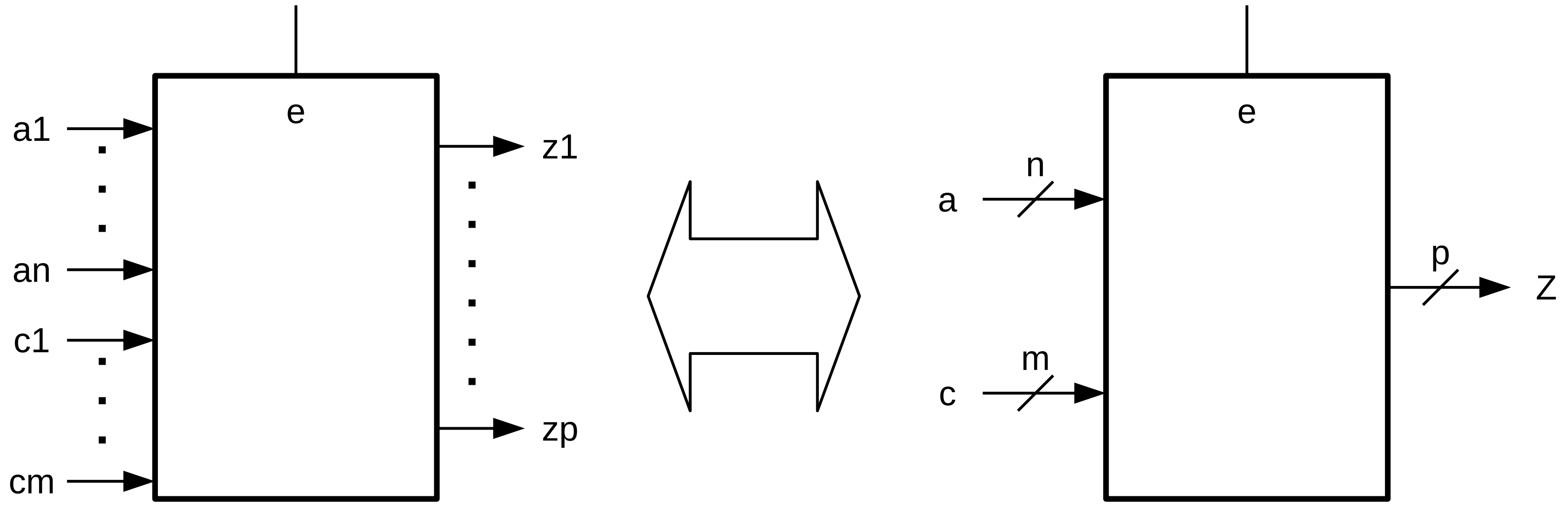


Enabled if

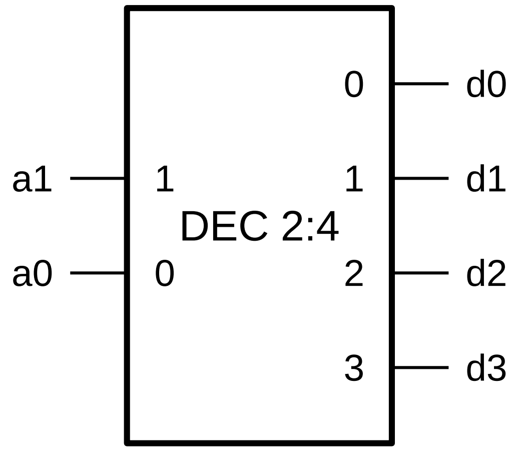
e1=1 & e2=0 & e3=0

- 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 signal notation: vector or buses



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
- 2n outputs

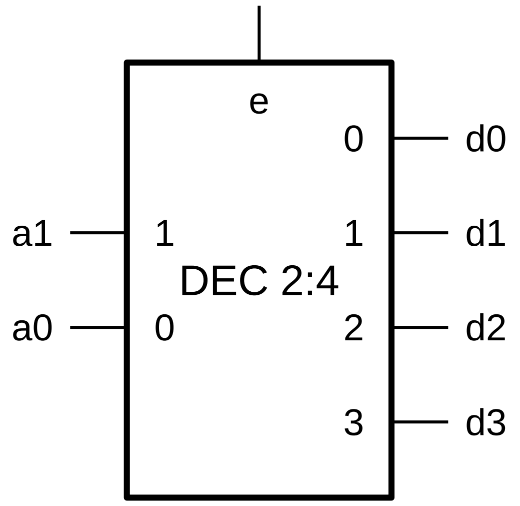
Implement all the minterms of the input variables

- 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



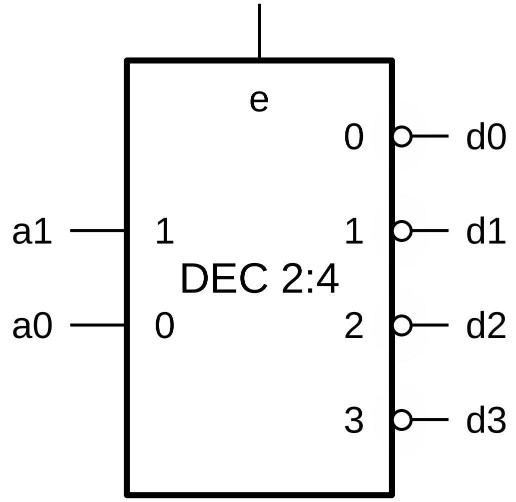
e	a1	a0	d0	d1	d2	d3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

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

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

```
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
```

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

Implement all the maxterms of the input variables (plus enable).

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

Natural binary to one-cold code converter.

```
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
```

Decoder design

- Just implement all the minterms/maxterms of the input variables.
- Enable signal:
  - Applied to all outputs.
  - Can be added post-design.

Example 1: design these decoders using logic gates

- DEC 2:4
- DEC 2:4, active low with active low enable

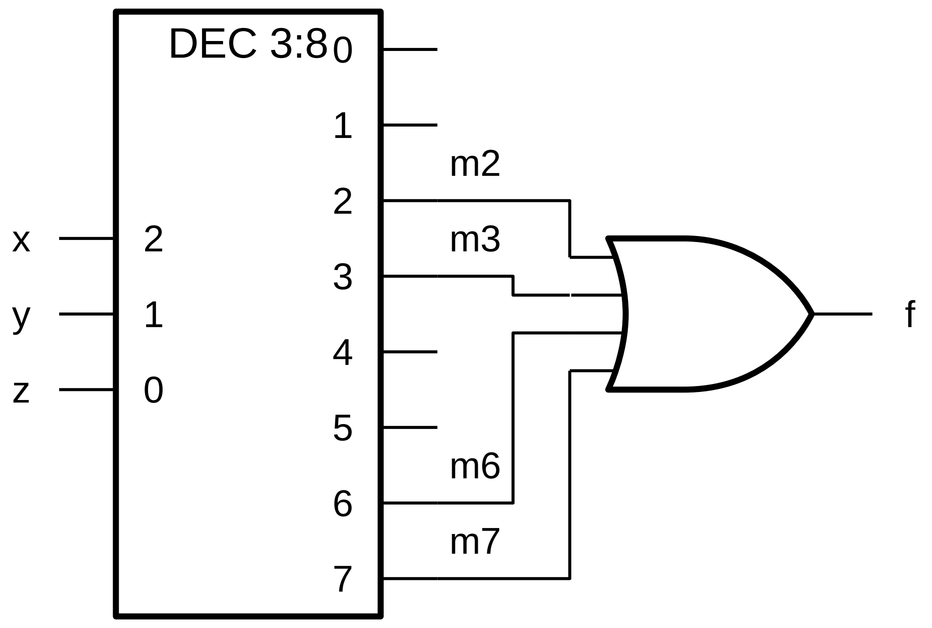
Logic function design with decoder + gates

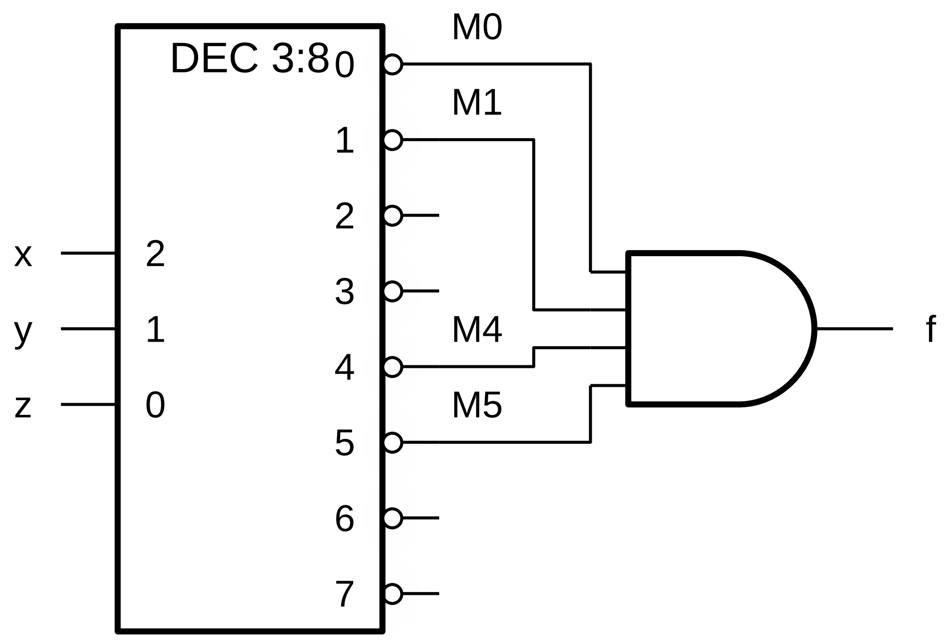
- Decoders provide a direct way to implement functions expressed as a sum of minterms or product of maxterms.

Example 2: f(x, y, z) = ∑(2, 3, 6, 7) = ∏(0, 1, 4, 5)

Design f using a DEC 3:8 (active-high output) and an OR gate

Design f using a DEC 3:8 (active-low output) and an AND gate.





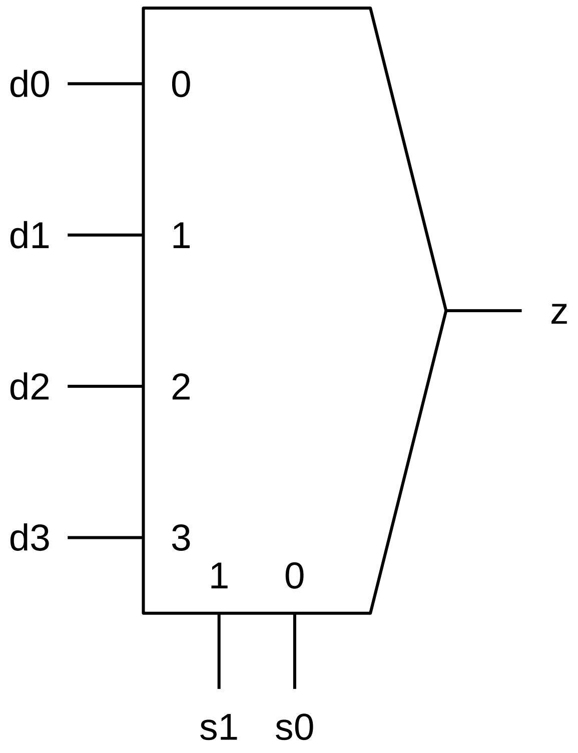
Decoder + gates: equivalent designs

- The structure DEC-OR is a particular case of the AND-OR
  - The AND part are the minterms generated by the decoder.
  - DEC-OR is equivalent to $\overline{\text{DEC}}$-NAND.
- The structure DEC-AND is particular case of the OR-AND
  - The OR part are the maxterms generated by the decoder.
  - $\overline{\text{DEC}}$-AND is equivalent to DEC-NOR.

Example 3: $f(x, y, z) = \sum(2, 3, 6, 7) = \prod(0, 1, 4, 5)$

a) Design f using a DEC 3:8 (active-low output) and a NAND gate
b) Design f using a DEC 3:8 (active-high output) and a NOR gate.

Multiplexer



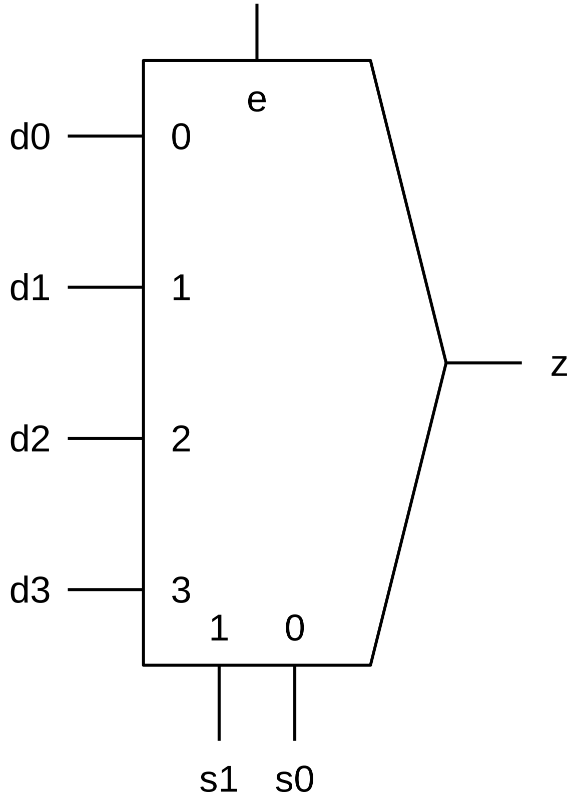
s1	s0	z
0	0	d0
0	1	d1
1	0	d2
1	1	d3

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
```

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

Multiplexer design

- Option 1: as a generic logic function (using a K-map)
  - Complex and unfeasible even for a few data inputs.
- Option 2: using a modular design and taking advantage of the regularity of the operation of the device for each data input.
  - Step 1: generate all the minterms of the selection inputs.
  - Step 2: AND each minterm with the corresponding data input.
  - Step 3: OR all the obtained terms

Example 4:

a) Design a MUX 8:1

b) Design a MUX 4:1 with an active-low enable input

Logic function design with multiplexers

- Based on the Shannon's expansion theorem.

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

$$f(x_1,x_2,x_3) = \overline{x}_1\, f(0,x_2,x_3) + x_1\, f(1,x_2,x_3) =$$

$$\overline{x}_1\overline{x}_2\, f(0,0,x_3) + \overline{x}_1 x_2 f(0,1,x_3) + x_1\overline{x}_2 f(1,0,x_3) + x_1 x_2 f(1,1,x_3) =$$

$$\overline{x}_1\overline{x}_2\,\overline{x}_3 f(0,0,0) + \overline{x}_1\overline{x}_2\, x_3 f(0,0,1) + \overline{x}_1 x_2\overline{x}_3 f(0,1,0) + \overline{x}_1 x_2 x_3 f(0,1,1) +$$
$$x_1\overline{x}_2\overline{x}_3 f(1,0,0) + x_1\overline{x}_2 x_3 f(1,0,1) + x_1 x_2\overline{x}_3 f(1,1,0) + x_1 x_2 x_3 f(1,1,1)$$

$$f(x_1,x_2,x_3) = \overline{x}_1\overline{x}_2\,\overline{x}_3 f_0 + \overline{x}_1\overline{x}_2\, x_3 f_1 + \overline{x}_1 x_2\overline{x}_3 f_2 + \overline{x}_1 x_2 x_3 f_3 +$$
$$+ x_1\overline{x}_2\overline{x}_3 f_4 + x_1\overline{x}_2 x_3 f_5 + x_1 x_2\overline{x}_3 f_6 + x_1 x_2 x_3 f_7$$

Logic function design with multiplexers

Example 5

Design $f(x, y, z) = \sum(2, 3, 6, 7)$ with MUX 8:1

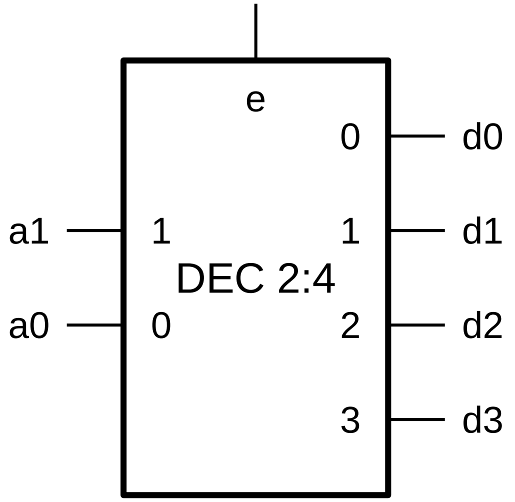
Example 6

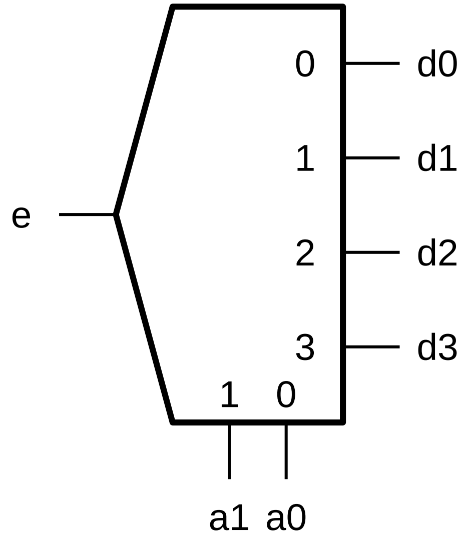
Design $f(w, x, y, z) = \sum(0, 1, 2, 6, 7, 8, 12, 14, 15)$

a) with MUX 8:1

b) with MUX 4:1

Demultiplexer





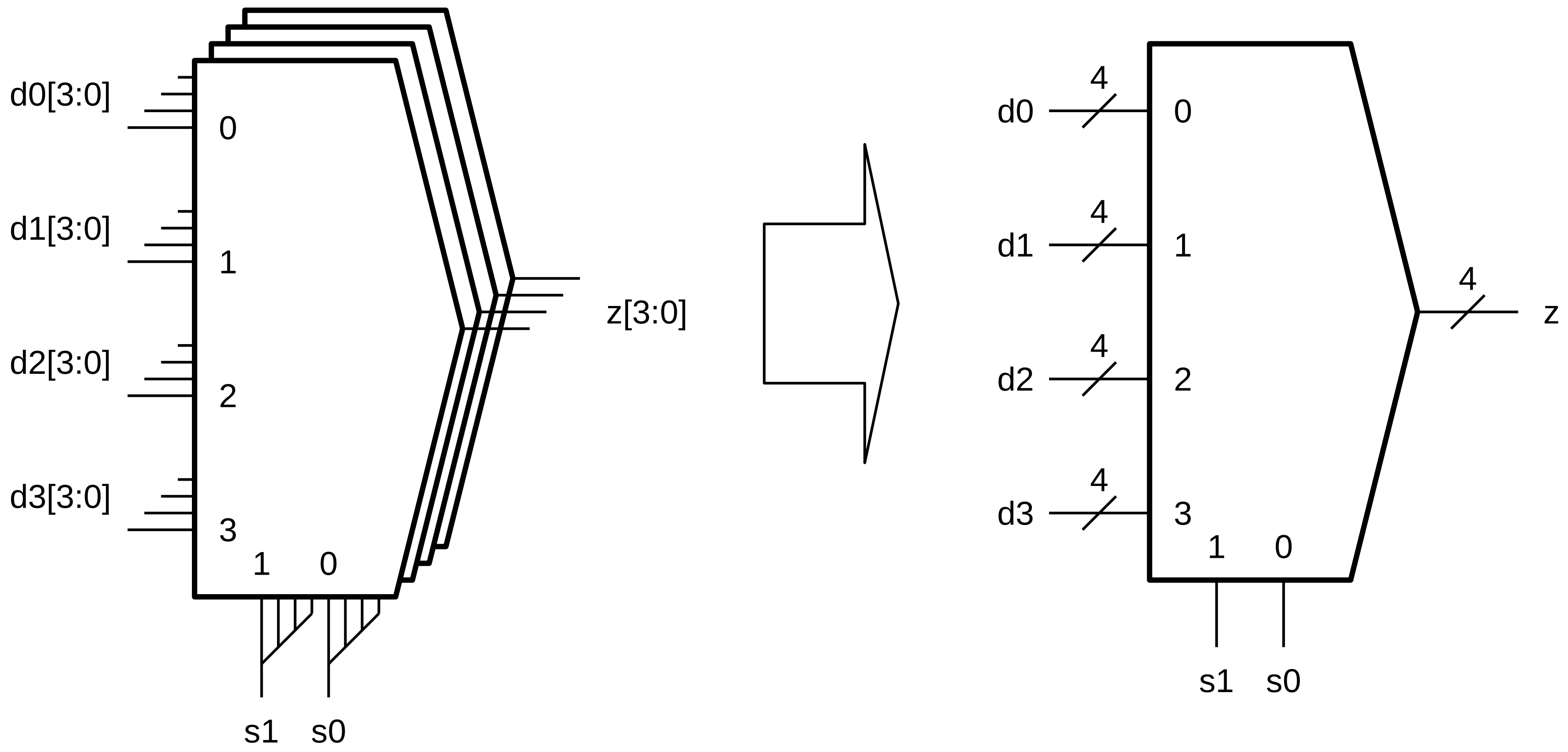
e	a1	a0	d0	d1	d2	d3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

$\Leftrightarrow$

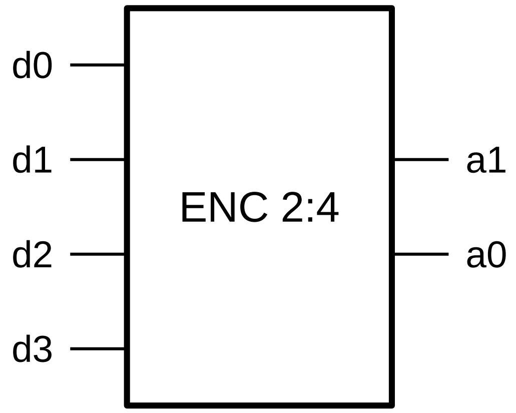
a1	a0	d0	d1	d2	d3
0	0	e	0	0	0
0	1	0	e	0	0
1	0	0	0	e	0
1	1	0	0	0	e

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

Multi-channel multiplexers



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.

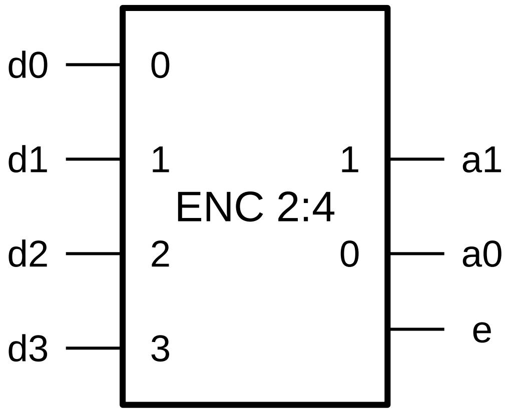
Inputs can be active low or high.

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;
        endcase
endmodule
```

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

```
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
```

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.

Encoder design

- Option 1: as a generic logic function (using a K-map)
  - Complex and unfeasible even for a few data inputs.
- Option 2: using a modular design and taking advantage of the regularity of the operation of the device for each data input.
- E.g.: modular design of a priority encoder.
  - $(a_1,a_0) = \overline{d_3}$ (value of (a_1,a_0) when $d_3=0$) + d3 (1,1)
  - Obtain “value of (a_1,a_0) when $d_3=0$” in the same way and substitute.

Encoder design

Example 6

Design the following encoders using K-maps:

a) 4-bit binary encoder

b) 4-bit Gray encoder

Add an enable input signal after completing the design.

Example 7

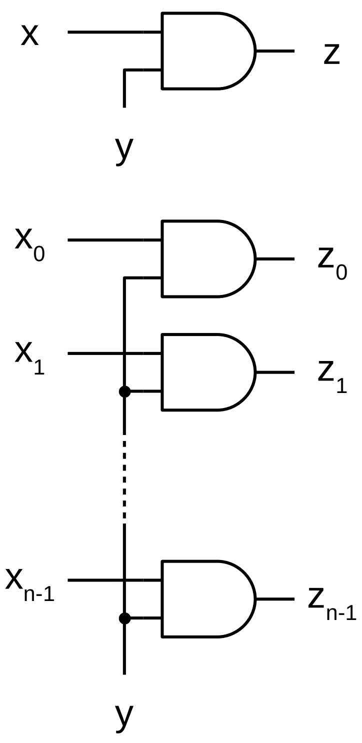
Design a 4-bit priority encoder using modular design: i.e. derive the logic expression from the encoder behavior.

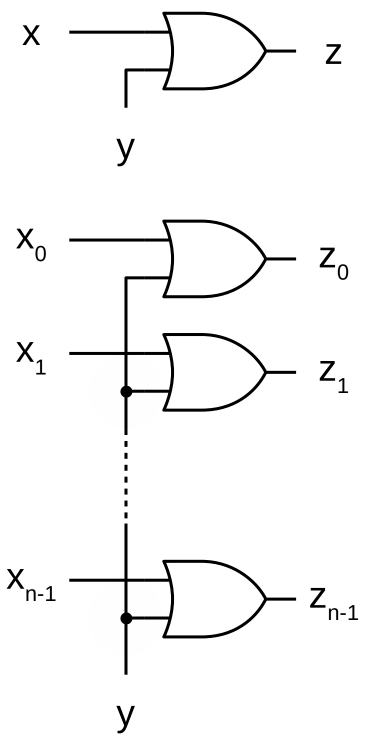
Logic gates as pass through blocks

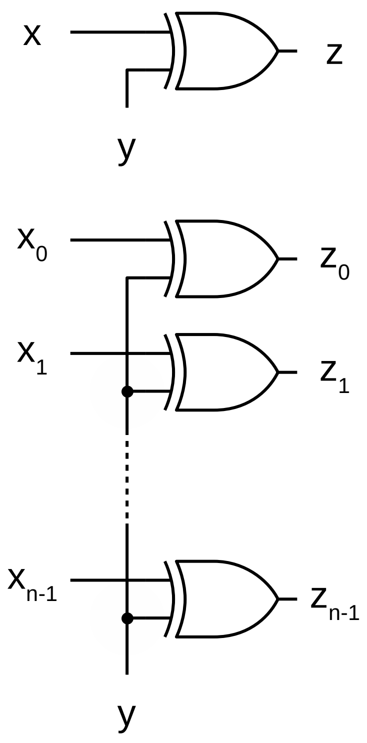
AND pass through
if $y = 1$, $z = x$
else, $z = 0$

OR pass through
if $y = 0$, $z = x$
else, $z = 1$

XOR Transfer/complement
if $y = 0$, $z = x$
else, $z = \overline{x}$

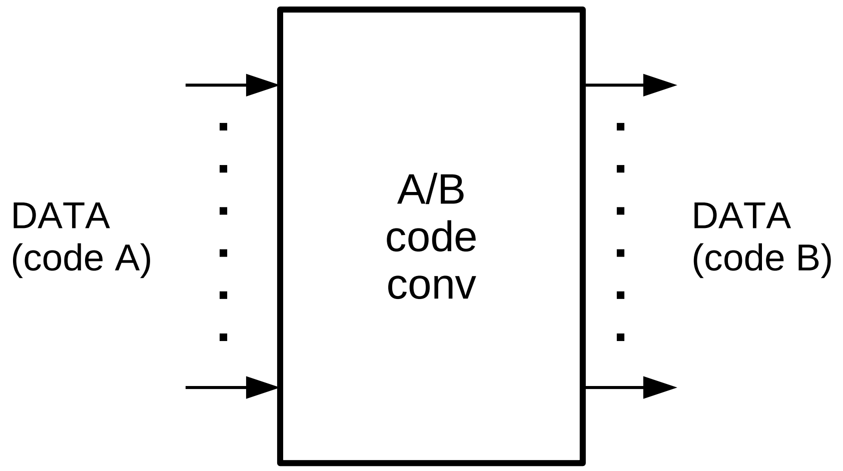




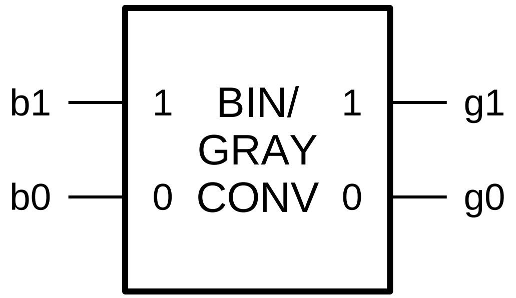


Code converters

- Convert information from one encoding to another (information does not change, only its representation)
  - (Natural) binary to Gray
  - Gray to binary
  - BCD/7-segment
  - ...



Example: 2-bit Bin/Gray converter

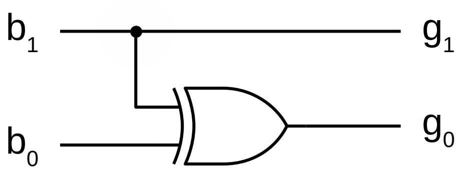


b_1	b_0	g_1	g_0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

$g1 = b_1$

$g_0 = b_1\overline{b}_0+\overline{b}_1b_0$

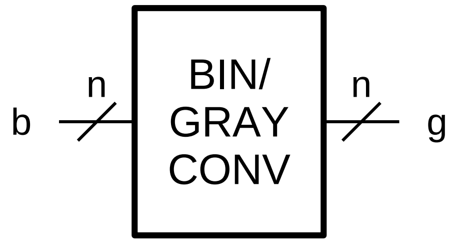
$g_0 = b_1 \oplus b_0$



```
// Procedural description
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
```

```
// Functional description
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
```

Binary to Gray generic code converter



For every $i < n-1$:

  if $b_{i+1} = 0$, $g_i = b_i$

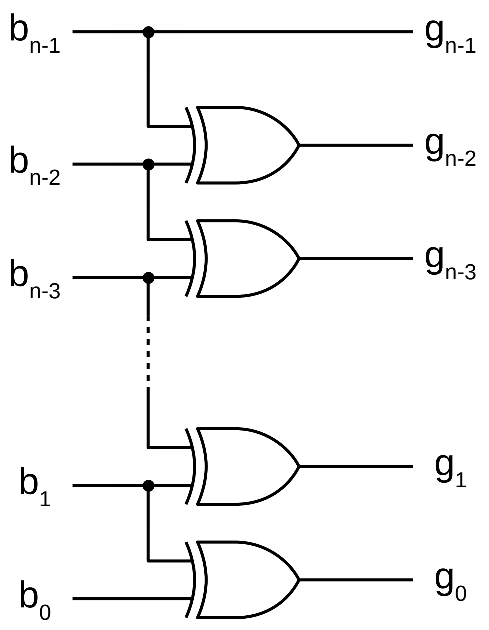
  else, $g_i = \overline{b_i}$

⇒

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

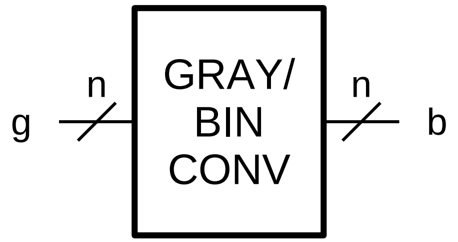
$g_i = b_i \oplus b_{i+1}$

$b_3b_2b_1b_0$	$g_3g_2g_1g_0$
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
```

Gray to binary generic code converter



For every i < n-1:

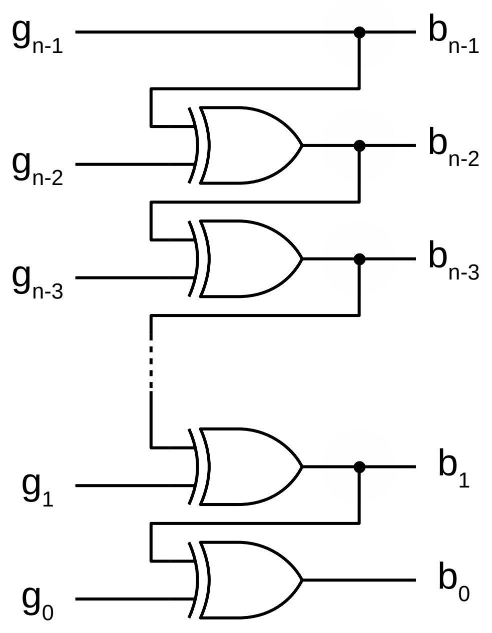
  if $b_{i+1} = 0$, $b_i = g_i$

  else, $b_i = \overline{g_i}$

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

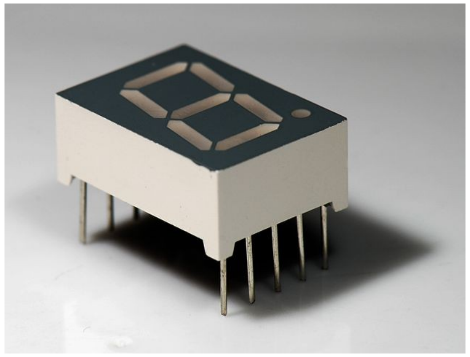
$bi = g_i \oplus b_{i+1}$

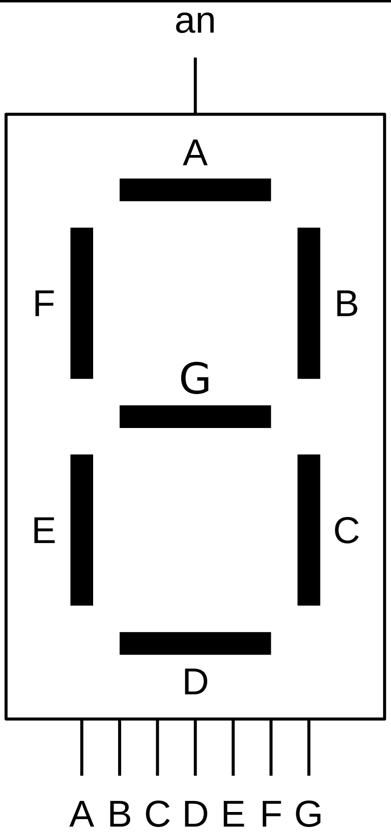
$g_3g_2g_1g_0$	$b_3b_2b_1b_0$
0000	0000
0001	0001
0011	0010
0010	0011
0110	0100
0111	0101
0101	0110
0100	0111
1100	1000
1101	1001
1111	1010
1110	1011
1010	1100
1011	1101
1001	1110
1000	1111

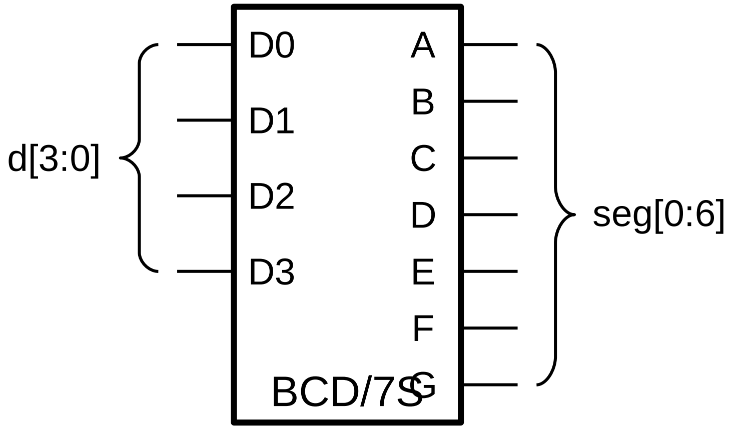


```
module gray_bin #(
  parameter n = 4
  )(
  input wire [n-1:0] g,
  output reg [n-1:0] b
  );

  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
```

BCD/7-segment converter





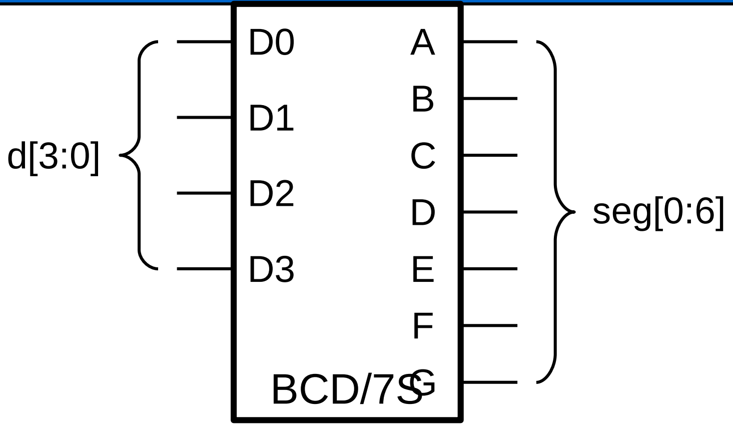


$d_3d_2d_1d_0$	d	seg[0:6] ABCDEFG
0000	0	0000001
0001	1	1001111
0010	2	0010010
0011	3	0000110
0100	4	1001100
0101	5	0100100
0110	6	0100000
0111	7	0001111
1000	8	0000000
1001	9	0001100

Common anode:
- an must be '1' for the device to work.
- Segments are active at a low level

https://en.wikipedia.org/wiki/Seven-segment_display#/media/File:Seven_segment_02_Pengo.jpg

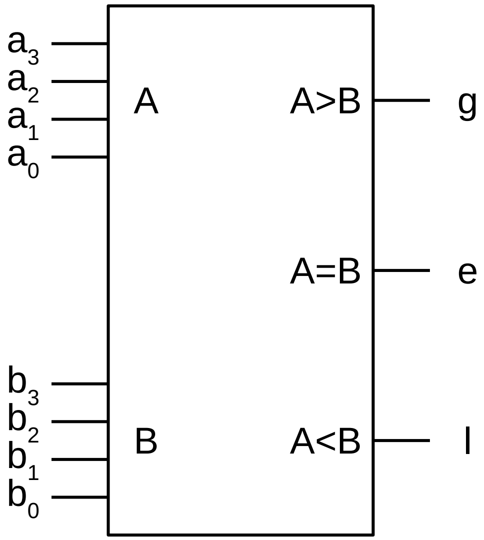
BCD/7-segment converter



$d_3d_2d_1d_0$	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
```

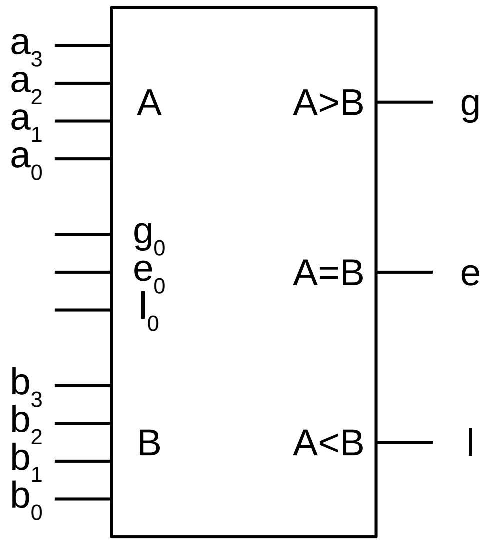
Comparators: simple comparator



A B	g	e	l
A>B	1	0	0
A=B	0	1	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} = {0,1,0};
  end

endmodule
```

Comparators: with cascade inputs

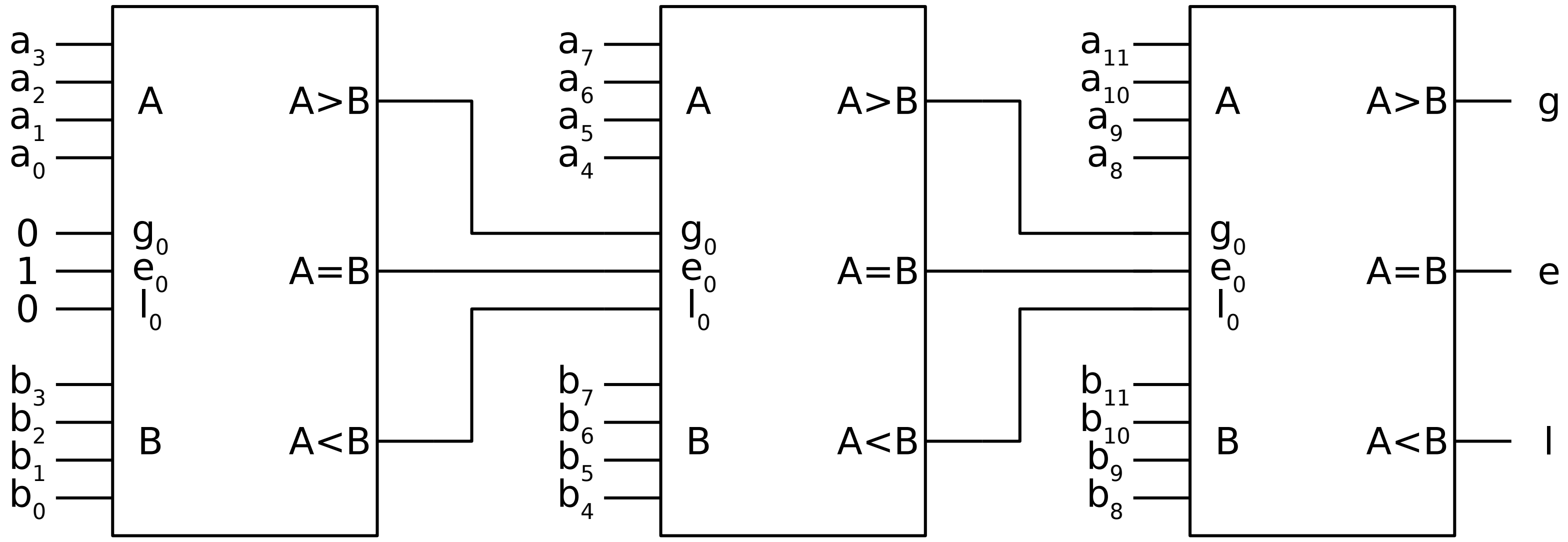


A B	g	e	l
A>B	1	0	0
A=B	g_0	e_0	l_0
A<B	0	0	1

```
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: cascade inputs usage

12-bit comparator out of 4-bit comparators



Parity detectors/generators

Definition: Given a word x of n bits from x_0 to x_{n-1}, we define the parity of the word up to bit i in x, p_i, so that:

$p_i = 0$ if the number of bits set to 1 from x_0 to x_{i-1} is an even number.
$p_i = 1$ if the number of bits set to 1 from x_0 to x_{i-1} is an odd number.

Theorem: $p_0 = x_0$.

Theorem: $p_i = p_{i-1}$ iif $x_i = 0$; $p_i = \overline{p}_{i-1}$ iif $x_i = 1$.

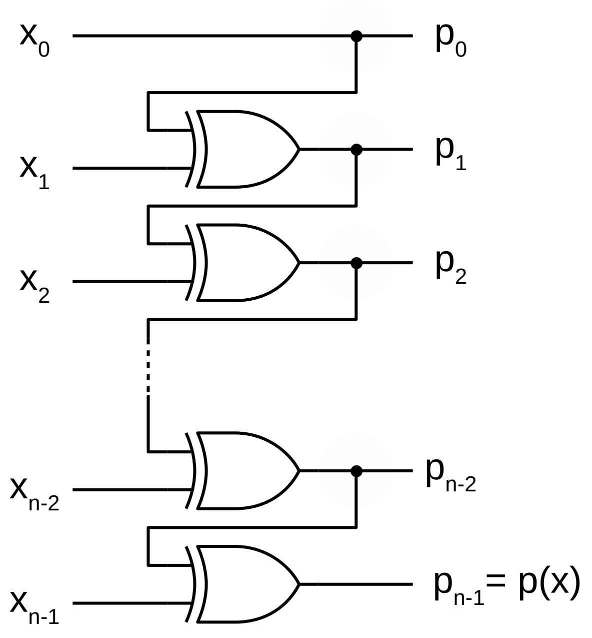
Corolary: $p_i = x_i \oplus p_{i-1}$.

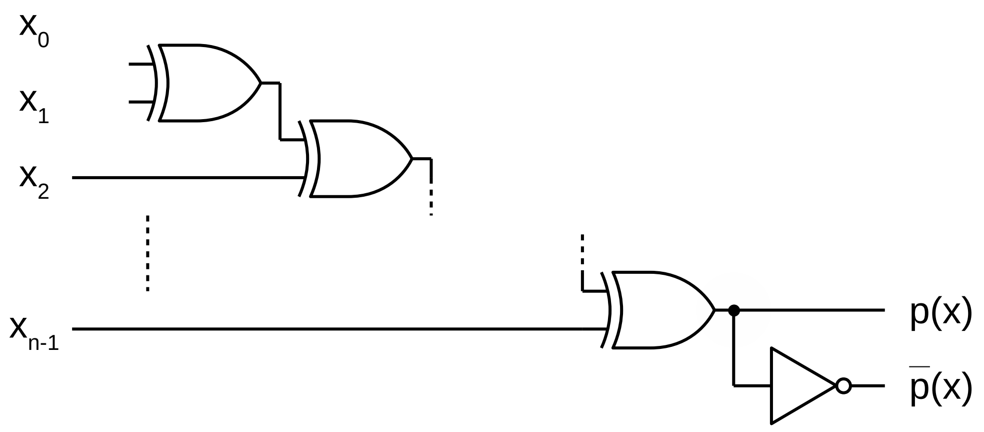
Definition: Given a word x of n bits from x_0 to x_{n-1}, we define the parity of word x, p(x), as the parity up to bit n-1 of x.

Theorem: A word x of n bits augmented with its parity bit produces a n+1 bits word of even parity.

Theorem: A word x of n bits augmented with the complement of its parity bit produces a n+1 bits word of odd parity.

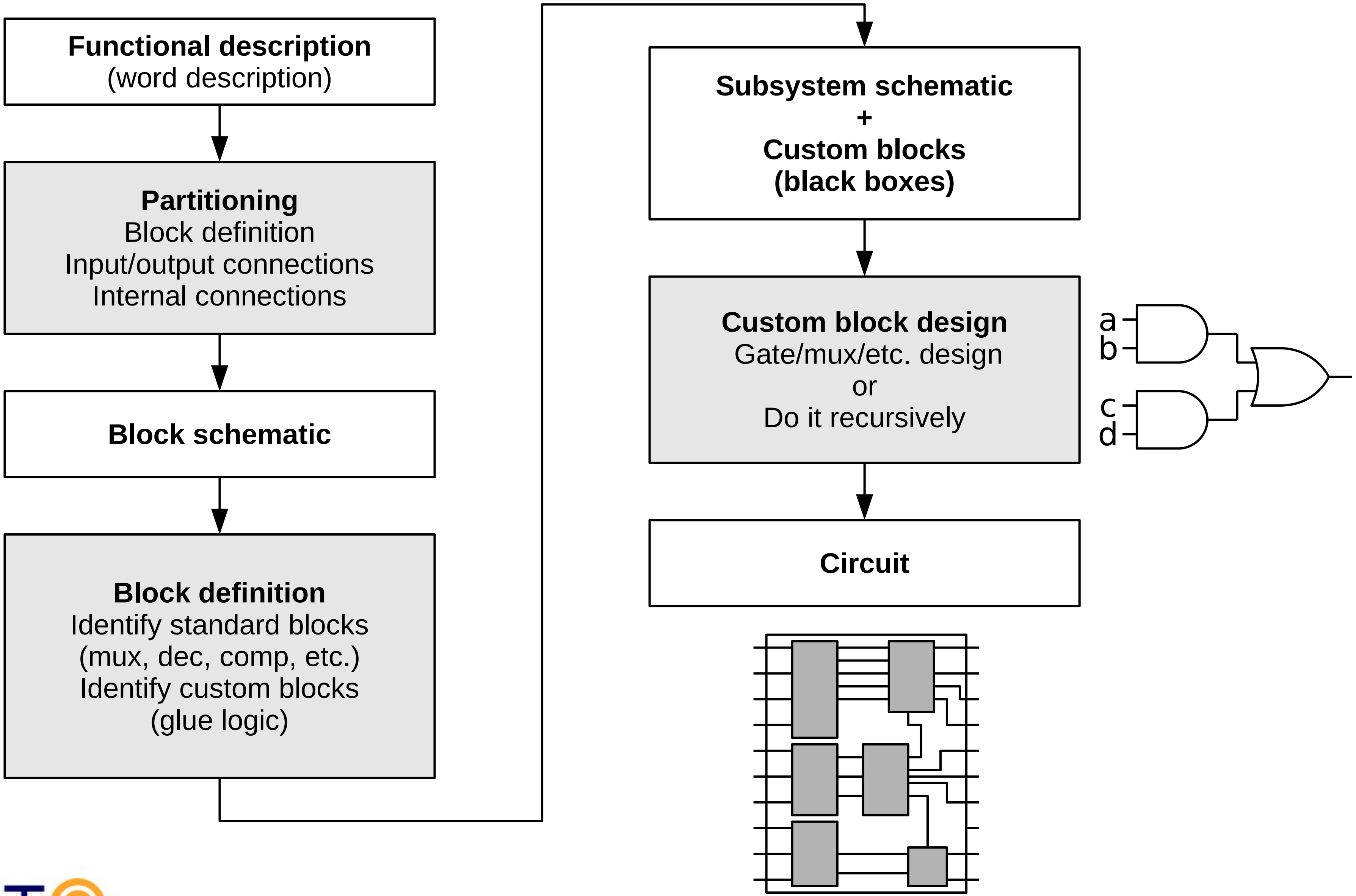
Parity detectors/generators





- $p(x)$
  - Odd parity detection (p=1).
  - Even parity generation.
- $\overline{p}(x)$
  - Even parity detection (p=1).
  - Odd parity generation.

Design methodology with subsystems



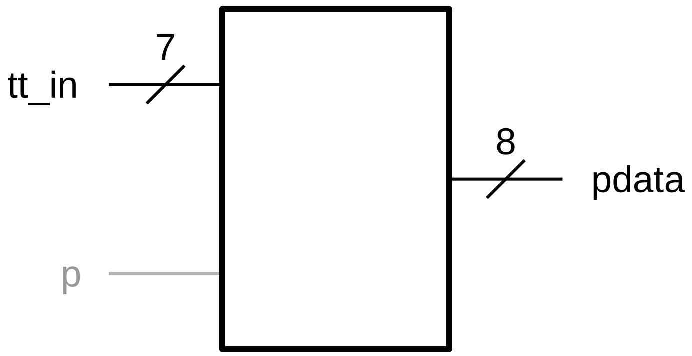
Design examples

Example 8

We need to design an interface module for an old teletype system in order to connect it to a standard serial computer port. The teletype generates 7-bit ASCII character codes while our computer can only receive 8-bit words (bytes) that must have even parity.

a) Design a module that reads 7-bit words through a 'tt_in' input signal and generates 8-bit words with leading even parity bit in output signal 'pdata' (bit 7 in pdata is the parity bit and the rest of the bits are copied from tt_in)

b) Add a control signal 'p' to the system to select the parity of the generated output so that the output will be even when p=0 and odd when p=1.



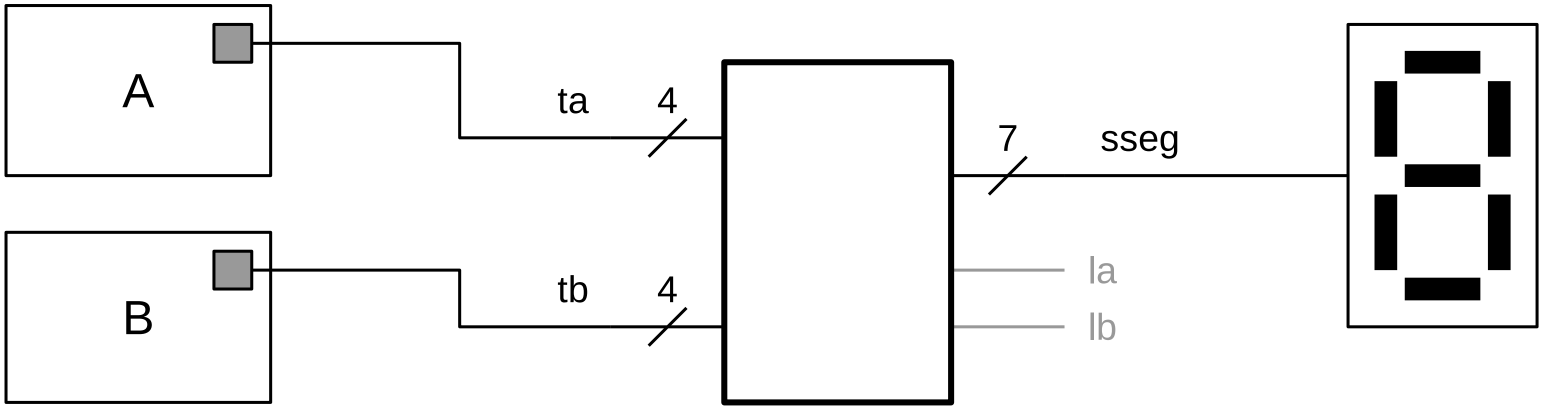
Design examples

Example 9

Two experiments are carried out in two rooms A and B. It is important to know the maximum temperature achieved in both rooms. Temperature sensors provide digital reading of the temperature through 4-bit signals 'ta' and 'tb' that range from 0 to 9. We need a circuit that displays the temperature achieved in the room with the highest temperature.

a) Design a digital circuit using standard combinational subsystems that generates the 7-segment code 'sseg' for the temperature in the room with the highest temperature.

b) Add two additional output to the system 'la' and 'lb' to control two LEDs. 'la' should be '1' when ta>=tb, and 'lb should be '1' when tb>=ta.



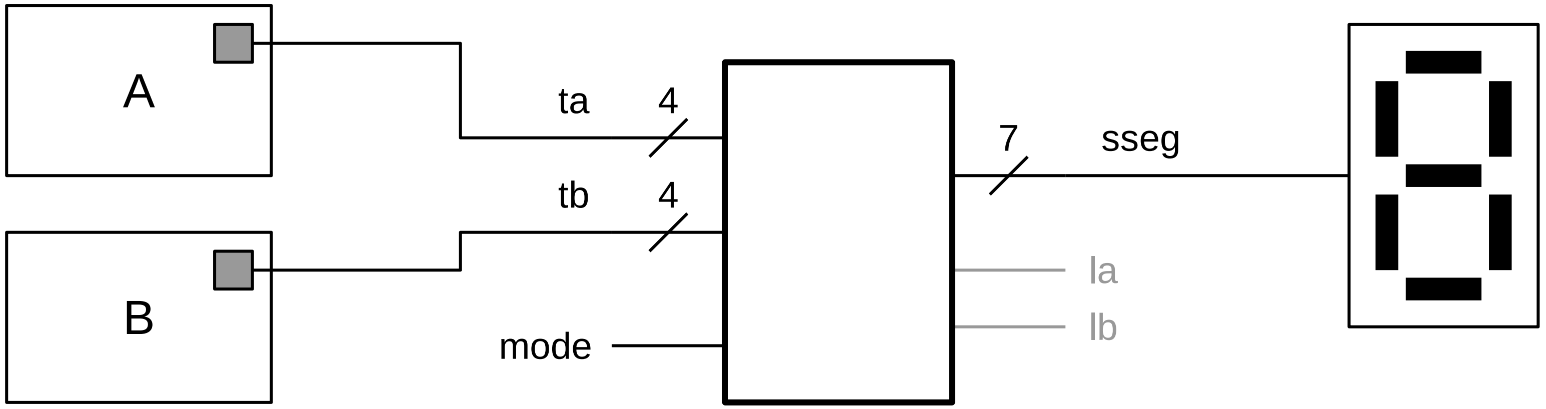
Design examples

Example 10

Add a control signal 'mode' to the circuit in the previous example so that:

- When mode=0, the temperature shown and the active LED correspond to the maximum (as in the previous example).
- When mode=1, the temperature shown and the active LED corresponds to the minimum.

In both cases, both LEDs should be on when the temperatures in A and B are the same.



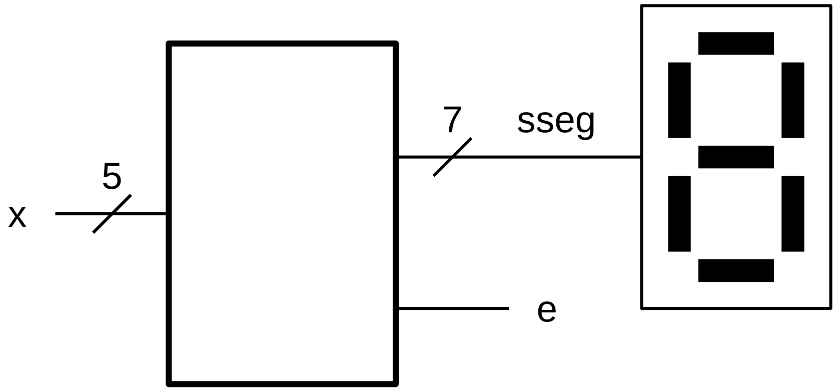
Design examples

Example 11

A computation system receives BCD digits through an input 'x' of 5 bits, where the most significant bit is an even parity bit.

a) Design a circuit that tests the parity of the input number and displays the number in a 7-segment display. An error output 'e' will be activated when the parity is not correct or the input number is not a BCD digit.

b) Modify the design so that when there is an error, the displays shows the symbol corresponding to number 14 ($1110_{(2} = E_{(16}$).



Combinational Subsystems and HDLs

- Design with combinational subsystems and HDL similarities:
  - Circuit partitioning: divide the problem in blocks.
  - Work with data words (not only bits).
  - Complex operations are easy to express: compare, select, convert, etc.
- Some HDL statements are almost equivalent to some comb. sub.
  - case(x) … → MUX
  - out[a] = 1'b1 → Decoder
  - if (a < B) … → Comparator
- A circuit built using comb. sub. can be easily translated to HDL
  - But comb. sub. blocks do not need to match exactly HDL modules.
- HDL synthesis tools work by mapping HDL statements to standard combination subsystems.

Combinational Subsystems and HDLs
E.g.: Verilog code for example 9

```
module max_temp(
    input wire [3:0] ta, tb,
    output reg [0:6] sseg,
    output reg la, lb
    );

    reg [3:0] max_temp;                 // internal signal to hold the
                                        // maximum temperature

    always @*
        if (ta > tb) begin              // comparator
            max_temp = ta;              // multplexing: max_temp, ta and tb are
            la = 1'b1; lb = 1'b0;       // assigned depending on the comparison
        end else if (ta < tb) begin     // results
            max_temp = tb;
            la = 1'b0; lb = 1'b1;
        end else begin
            max_temp = ta;
            la = 1'b1; lb = 1'b1;
        end

    bin-7s bcd_7s(   .d(max_temp),      // modularity: use bin to 7 segment
                     .seg(sseg));       // converter designed elsewhere

endmodule
```