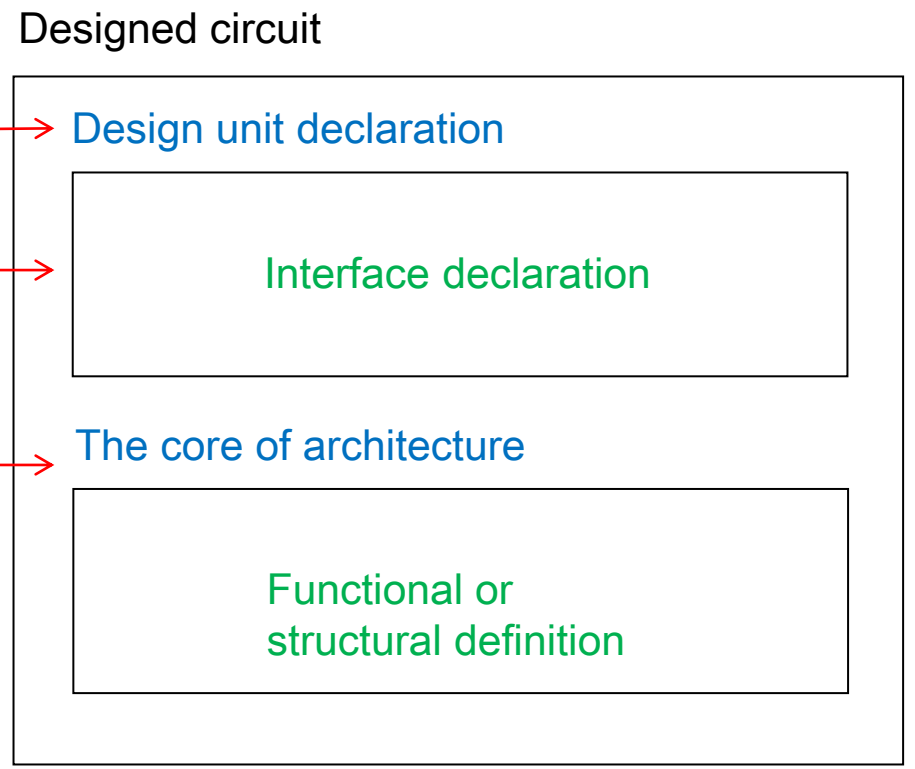
Digital Logic
Design with FPGA

Digital Design with VHDL

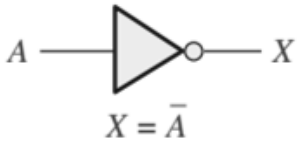
Inroduction

VHDL description

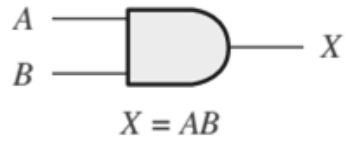
- Entity
  - generic
  - port
- Architecture
  - components
  - signals
  - concurrent statemens
    - processes



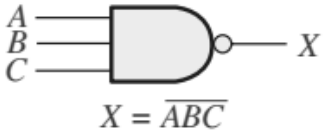
VHDL descriptions of basic logic



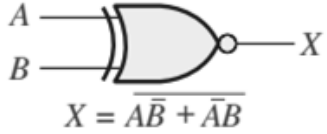

```
entity Inverter is
    port ( A: in bit;
           X: out bit );
end entity Inverter;
architecture NOTfunction of Inverter is
begin
    X <= not A;
end architecture NOTfunction;
```




```
entity ANDgate is
    port ( A, B: in bit;
           X: out bit );
end entity ANDgate;
architecture ANDfunction of ANDgate is
begin
    X <= A and B;
end architecture ANDfunction;
```

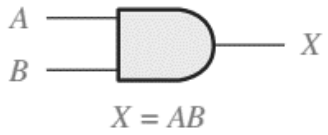



```
entity NANDgate is
    port ( A, B, C: in bit;
           X: out bit );
end entity NANDgate;
architecture NANDfunction of NANDgate is
begin
    X <= A nand B nand C;
end architecture NANDfunction;
```




```
entity XNORgate is
    port ( A, B: in bit;
           X: out bit );
end entity XNORgate;
architecture XNORfunction of XNORgate is
begin
    X <= A xnor B;
end architecture XNORfunction;
```

VHDL descriptions of basic logic



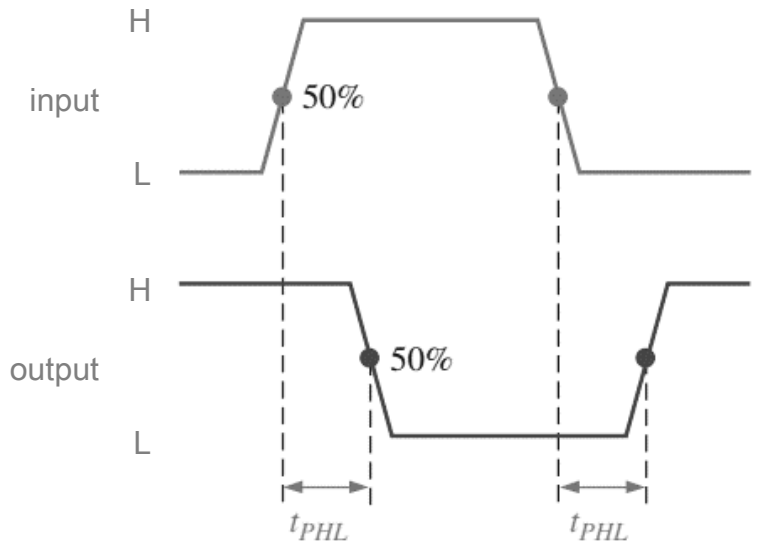

```
12  entity ANDgate is
13      port ( A, B: in bit;
14             X: out bit );
15  end entity ANDgate;
16  architecture ANDfunction of ANDgate is
17  begin
18      X <= A and B;
19  end architecture ANDfunction;
```

```
2   entity ANDgateN is
3       port ( A, B: in bit_vector(7 downto 0);
4              X: out bit_vector(7 downto 0) );
5   end entity ANDgateN;
6
7   architecture ANDfunction of ANDgateN is
8   begin
9       X <= A and B;
10  end architecture ANDfunction;
```

```
13  entity ANDgateG is
14      generic(N: natural := 8);
15      port ( A, B: in bit_vector(N-1 downto 0);
16             X: out bit_vector(N-1 downto 0) );
17  end entity ANDgateG;
18
19  architecture ANDfunction of ANDgateG is
20  begin
21      X <= A and B;
22  end architecture ANDfunction;
```

Basic parameters of digital logic

- Propagation delay time (t_p)
  the time interval between the transition of an input pulse and the occurrence of the resulting transition of the output pulse
  - t_{PHL}: the time between a specified reference point on the input pulse and a corresponding reference point on the resulting output pulse, with the output changing from the HIGH level to the LOW level (HL)
  - t_{PLH}: The time between a specified reference point on the input pulse and a corresponding reference point on the resulting output pulse, with the output changing from the LOW level to the HIGH level (LH)



source: Thomas Floyd: Digital Fundamentals

Basic parameters of digital logic

- DC supply voltage (V_{cc})
  - the typical dc supply voltage for CMOS logic is either 5V, 3.3V, 2.5V, or 1.8V, depending on the category
  - 5V CMOS fixed-logic can tolerate supply from 2V to 6V and still operate properly
  - 3.3V CMOS can operate with supply voltages from 2V to 3.6V

- Power dissipation (P_D) - product of the dc supply voltage and the average supply current; supply current when the gate output is LOW is greater than when the gate output is HIGH

$$P_D = V_{CC}\left(\frac{I_{CCH} + I_{CCL}}{2}\right)$$

  - power dissipation of CMOS is dependent on the frequency of operation
  - at zero frequency the quiescent power is typically in the microwatt/gate range (fixed-logic HC: 2.75 uW/gate at 0 Hz (quiescent) and 600 uW/gate at 1 MHz)
  - power dissipation for bipolar gates is independent of frequency (fixed-logic ALS: 1.4 mW/gate regardless of the frequency; F: 6 mW/gate)

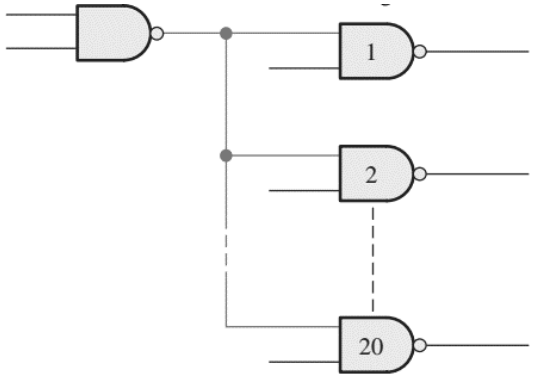
source: Thomas Floyd: Digital Fundamentals

Basic parameters of digital logic

- Speed-Power Product (SPP)
  can be used as a measure of the performance of a logic circuit
  taking into account the propagation delay time and the power dissipation

$$SPP = t_p P_D$$

- Fan-Out
  the maximum number of inputs of the next stage that can be connected to one output of a gate made in the same technology (gate still maintain the output voltage levels within specified limits)
  - Fan-out is specified in terms of **unit loads**
  - Example:
    fixed-logic 74LS00 NAND,
    current from a LOW input (I_{IL}) of a gate is 0,4mA
    and the current that a LOW output (I_{OL})
    can accept is 8,0mA



$$\text{unit loads} = \frac{I_{OL}}{I_{IL}} = \frac{8{,}0\ mA}{0{,}4\ mA} = 20$$

source: Thomas Floyd: Digital Fundamentals

Digital Logic
Design with FPGA

Digital Design with VHDL

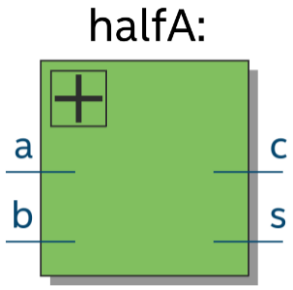
Combinational Logic

Outline

- Half and Full Adders
- Comparators
- Decoders
- Encoders
- Code Converters
- Multiplexers (Data Selectors)
- Demultiplexers
- Parity Generators/Checkers

Half-Adder

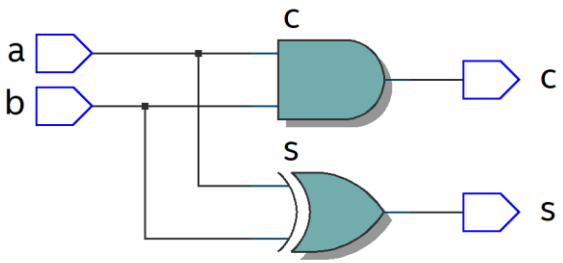
- The half-adder accepts two binary digits on its inputs and produces two binary digits on its outputs - a sum bit and a carry bit.



Half-adder truth table.

A	B	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

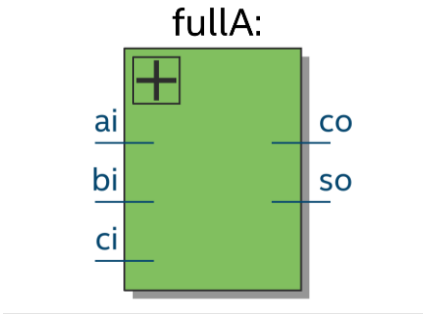
Σ = sum
C_{out} = output carry
A and B = input variables (operands)




```
10  entity halfA is port(
11      a,b:    in std_logic;
12      s,c:    out std_logic);
13  end halfA;
14  ----------------------------------
15  architecture df of halfA is
16  begin
17   s <= a xor b after 2 ns;
18   c <= a and b after 1 ns;
19  end df;
```

Full-Adder

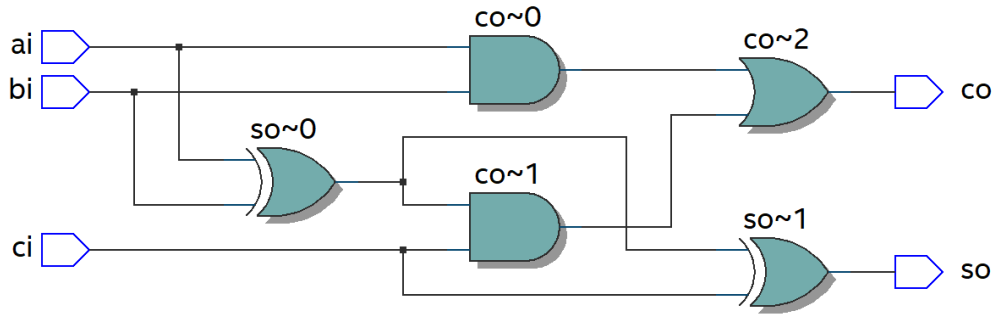
- The full-adder accepts two input bits and an input carry and generates a sum output and an output carry.



Full-adder truth table.

A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

C_{in} = input carry, sometimes designated as CI
C_{out} = output carry, sometimes designated as CO
Σ = sum
A and B = input variables (operands)

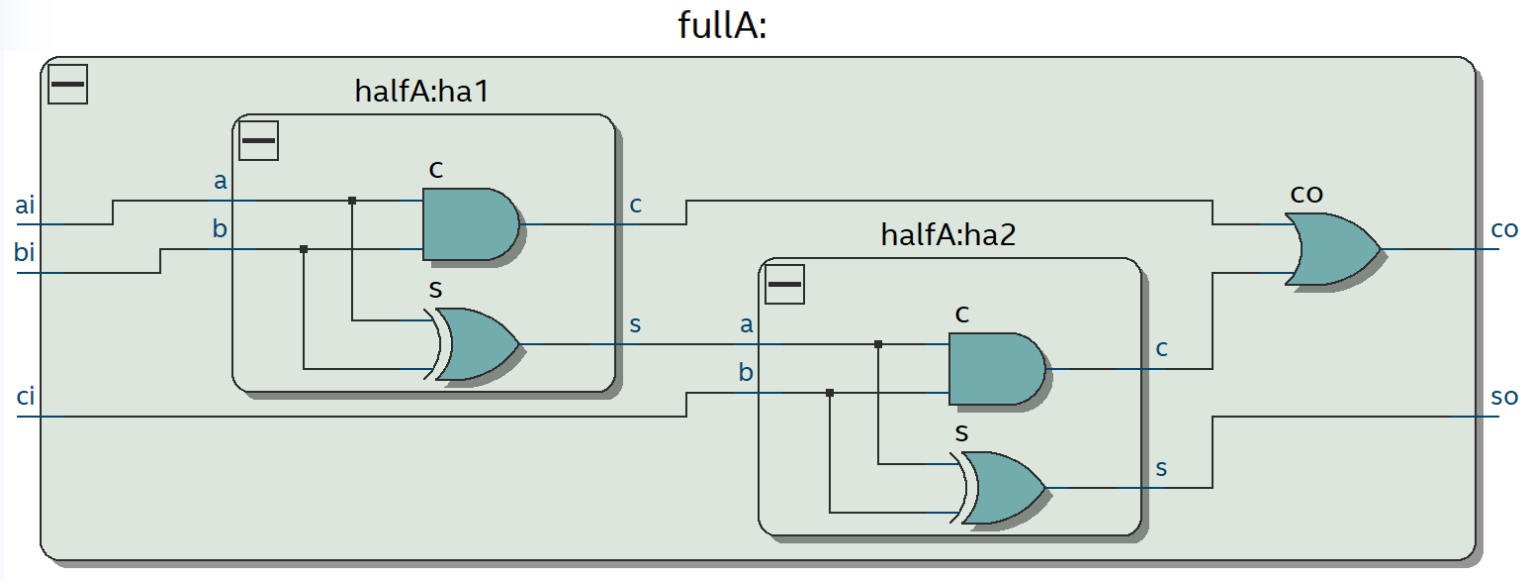



```
5  entity fullA is port(
6      ai,bi,ci:   in std_logic;
7      so,co:  out std_logic);
8  end fullA;
9  ----------------------------------------------
10 architecture gates of fullA is
11
12 begin
13   so <= (ai xor bi) xor ci;
14   co <= (ai and bi) or ((ai xor bi) and ci);
15 end gates;
```

Full-Adder

- Arrangement of two half-adders to form a full-adder

```
10  entity halfA is port(
11      a,b:    in std_logic;
12      s,c:    out std_logic);
13  end halfA;
```



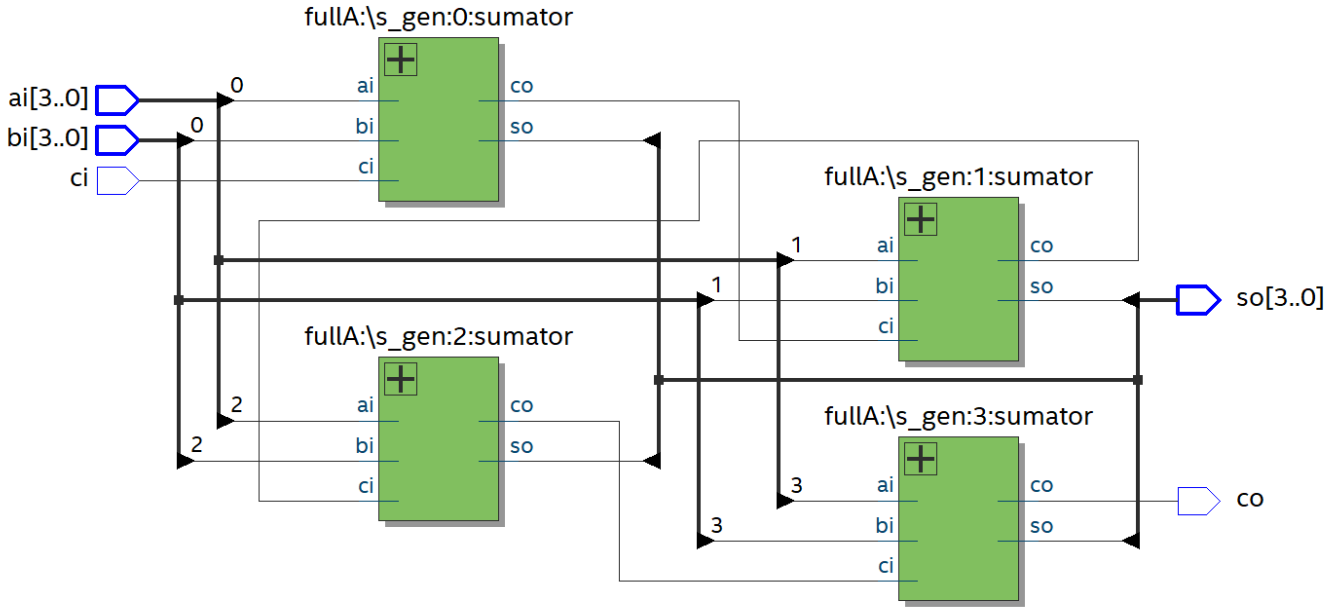

```
26  entity fullA is port(
27      ai,bi,ci:   in std_logic;
28      so,co:  out std_logic);
29  end fullA;
30  ---------------------------------------
31  architecture mix of fullA is
32  signal ha1s,ha1c,ha2c: std_logic;
33  begin
34  ha1: entity work.halfA(df)
35      port map(ai,bi,ha1s,ha1c);
36  ha2: entity work.halfA(df)
37      port map(ha1s,ci,so,ha2c);
38  co <= ha1c or ha2c after 1 ns;
39  end mix;
```

Parallel Binary Adder


```
26  entity fullA is port(
27      ai,bi,ci:   in std_logic;
28      so,co:  out std_logic);
29  end fullA;
```

```
46  entity adderN is
47      generic (N: integer:=4);
48      port(
49      ai,bi:  in std_logic_vector(N-1 downto 0):=(others=>'0');
50      ci: in std_logic;
51      so: out std_logic_vector(N-1 downto 0);
52      co: out std_logic);
53  end adderN;
```

```
55  architecture generacja of adderN is
56  signal carry: std_logic_vector(N downto 0):=(others=>'0');
57  begin
58  carry(0) <= ci;
59  co <= carry(N);
60  s_gen: for i in (N-1) downto 0 generate
61      sumator:entity work.fullA(mix)
62          port map(ai(i),bi(i),carry(i),so(i),carry(i+1));
63  end generate;
64
65  end generacja;
```



Parallel Binary Adder – FPGA implementation

```
46  entity adderN is
47      generic (N: integer:=64);
48      port(
49      ai,bi:  in std_logic_vector(N-1 downto 0):=(others=>'0');
50      ci: in std_logic;
51      so: out std_logic_vector(N-1 downto 0);
52      co: out std_logic);
53  end adderN;
54  ------------------------------------------------------------------
55  architecture structural of adderN is
56      signal carry: std_logic_vector(N downto 0):=(others=>'0');
57  begin
58      carry(0) <= ci;
59      co <= carry(N);
60      s_gen: for i in (N-1) downto 0 generate
61         sumator:entity work.fullA(mix)
62                 port map(ai(i),bi(i),carry(i),so(i),carry(i+1));
63         end generate;
64  end structural;
```

```
17  entity adder64 is
18      Port ( a : in  STD_LOGIC_VECTOR (63 downto 0);
19             b : in  STD_LOGIC_VECTOR (63 downto 0);
20             s : out  STD_LOGIC_VECTOR (63 downto 0);
21             ci: in std_logic;
22             co : out std_logic);
23  end adder64;
24
25  architecture Behavioral of adder64 is
26  signal s_i : std_logic_vector (64 downto 0);
27  signal a_i, b_i : std_logic_vector (64 downto 0);
28
29  begin
30  -- ins
31  a_i <= ('0' & a);
32  b_i <= ('0' & b);
33  -- func
34  s_i <= a_i + b_i + ci;
35  -- outs
36  s <= s_i(63 downto 0);
37  co <= s_i(64);
38  end Behavioral;
```

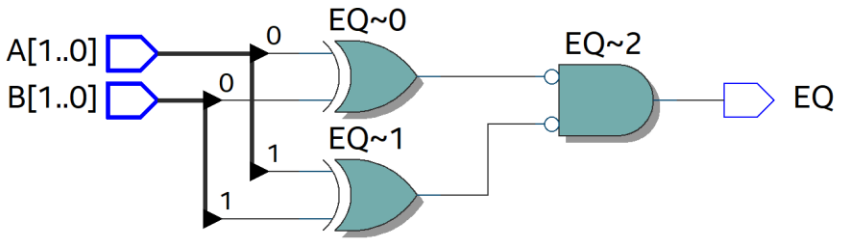
FPGA Device Utilization:

■ 4-input LUTs:	320	64
■ Maximum path delay:	160ns	11ns

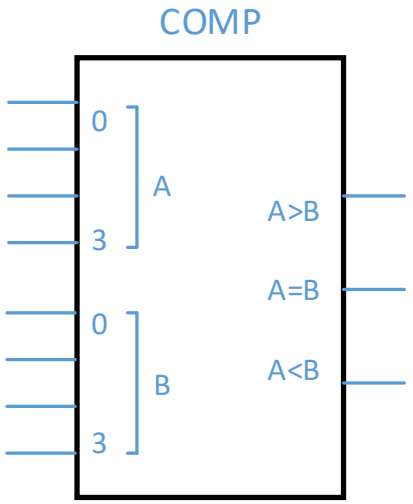
Comparator

- Basic function of a comparator is to compare the magnitudes of two binary quantities to determine the relationship of those quantities
- In simplest form, a comparator circuit determines whether two numbers are equal

- Equality



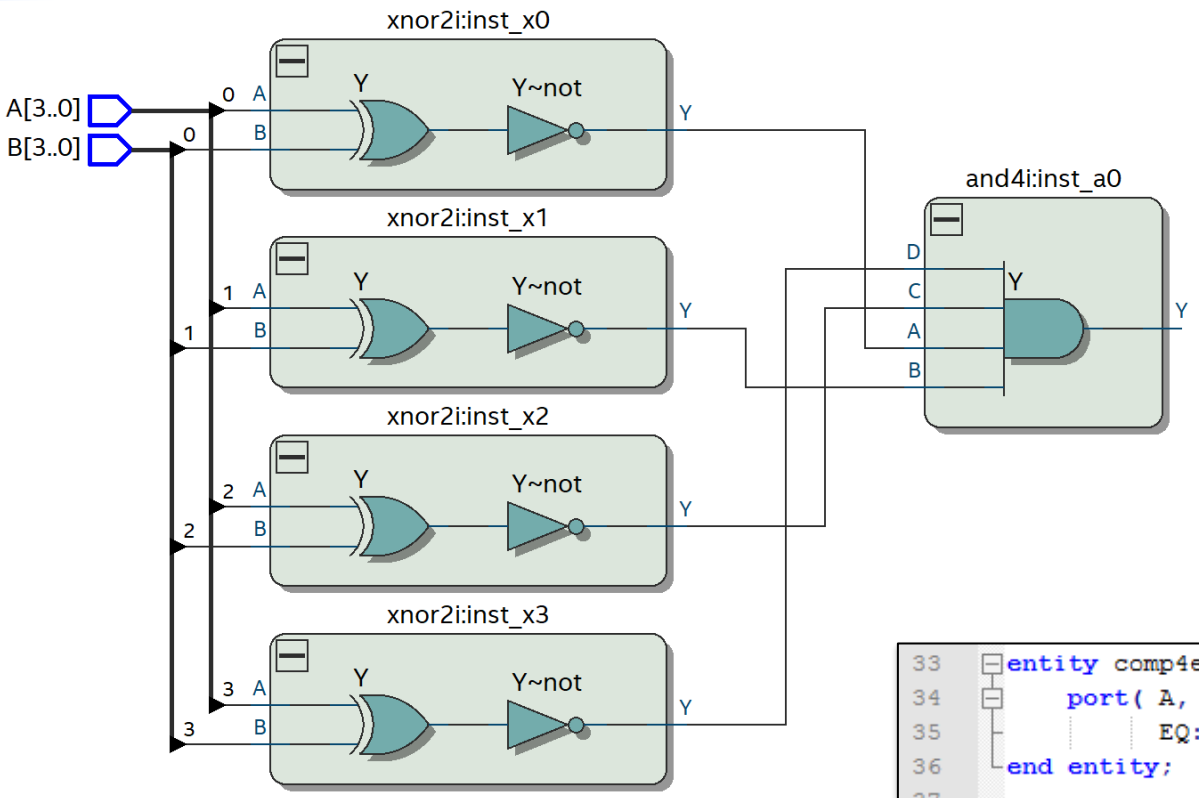
- Inequality



To determine an inequality of binary numbers A and B:

- If A3 = 1 and B3 = 0, number A is greater than number B.
- If A3 = 0 and B3 = 1, number A is less than number B.
- If A3 = B3, then you must examine the next lower bit position for an inequality.

Comparator - VHDL

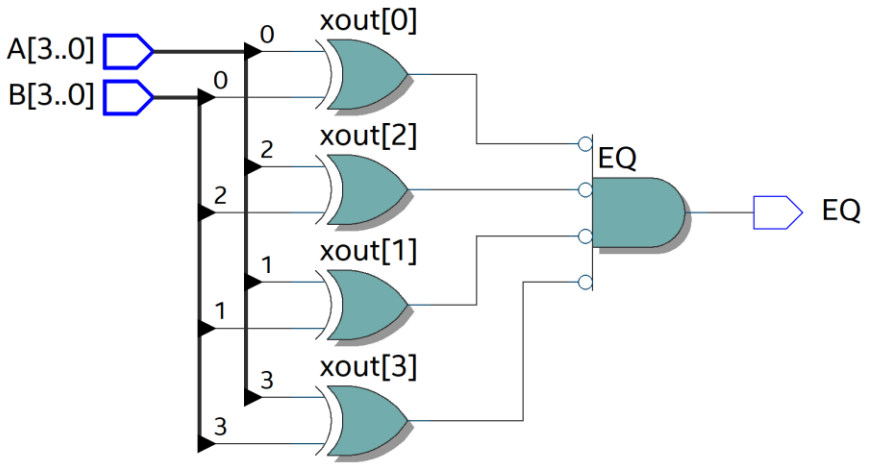



```vhdl
entity xnor2 is
    port ( A, B: in std_logic;
           Y: out std_logic );
end entity xnor2;

architecture behav of xnor2 is
begin
    Y <= not (A xor B);
end architecture behav;
```

```vhdl
entity and4 is
    port ( A, B, C, D: in std_logic;
           Y: out std_logic );
end entity and4;

architecture behav of and4 is
begin
    Y <= A and B and C and D;
end architecture behav;
```

```vhdl
entity comp4eq is
    port( A, B: in std_logic_vector(3 downto 0);
          EQ: out std_logic );
end entity;

architecture struct of comp4eq is
signal xout: std_logic_vector(A'range);

begin
inst_x0: entity work.xnor2 port map(A=>A(0), B=>B(0), Y=>xout(0) );
inst_x1: entity work.xnor2 port map(A=>A(1), B=>B(1), Y=>xout(1) );
inst_x2: entity work.xnor2 port map(A=>A(2), B=>B(2), Y=>xout(2) );
inst_x3: entity work.xnor2 port map(A=>A(3), B=>B(3), Y=>xout(3) );
inst_a0: entity work.and4
          port map(A=>xout(0), B=>xout(1), C=>xout(2), D=>xout(3), Y=>EQ);
end architecture;
```

Comparator - VHDL




```vhdl
entity comp4eq is
    port( A, B: in std_logic_vector(3 downto 0);
          EQ: out std_logic );
end entity;

architecture behav of comp4eq is
signal xout: std_logic_vector(A'range);
begin
    EQ <= xout(3) and xout(1) and xout(2) and xout(0);
    xout <= A xnor B;
end architecture;
```

```vhdl
entity compNeq is
    generic(N: positive := 32);
    port( A, B: in std_logic_vector(N-1 downto 0);
          EQ: out std_logic );
end entity;

architecture behav of compNeq is

begin
    EQ <= '1' when (A=B) else '0';
end architecture;
```

Comparator – FPGA implementation

4-input LUTs

```
39  entity compN is
40      generic(N: positive := 32);
41      port( A, B: in std_logic_vector(N-1 downto 0);
42            EQ, AthenB, BthenA: out std_logic );
43  end entity;
44
45  architecture behav1 of compN is
46  signal xout: std_logic_vector(2 downto 0);
47  begin
48      EQ<=xout(2); AthenB<=xout(1); BthenA<=xout(0);
49      xout <= "100" when (A=B) else
50              "010" when (A>B) else
51              "001" when (A<B) else
52              "000";
53  end architecture;
```

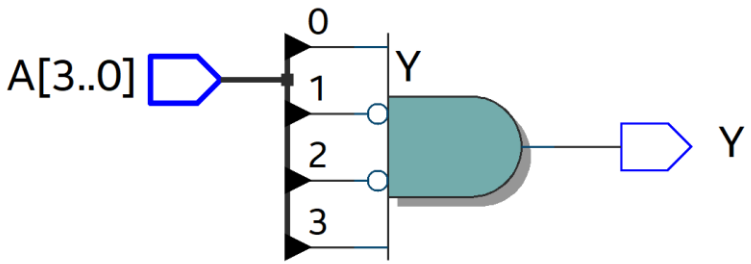
- 82

```
55  architecture behav2 of compN is
56  signal eqi: std_logic:='0';
57  begin
58  eqi <= '1' when (A=B) else '0';
59  AthenB <= '1' when (A>B) else '0';
60  BthenA <= '0' when (A>B) else (not eqi);
61  EQ <= eqi;
62  end architecture;
```

- 50

Decoder

- A decoder is a digital circuit that detects the presence of a specified combination of bits (code) on its inputs
- In general form, a decoder has n input lines to handle n bits and from one to 2^n output lines to indicate the presence of one or more n-bit combinations

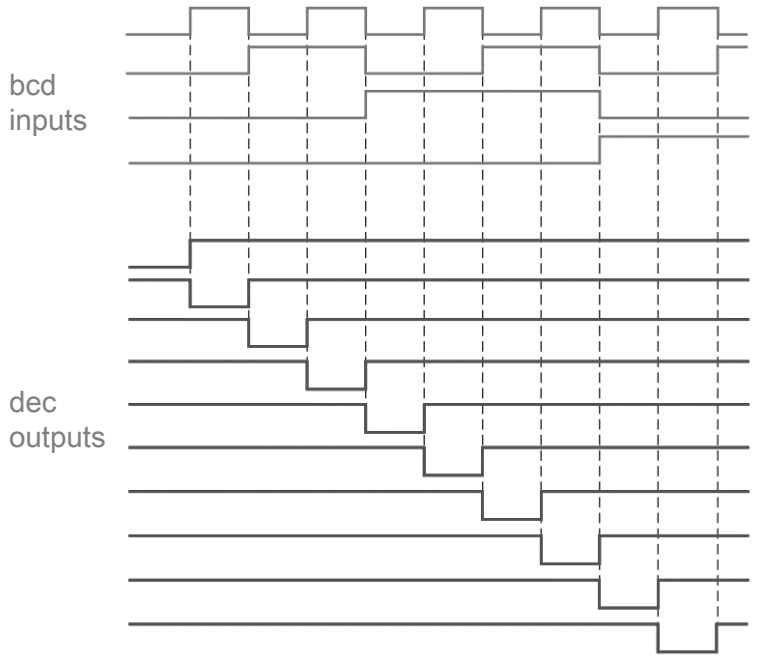


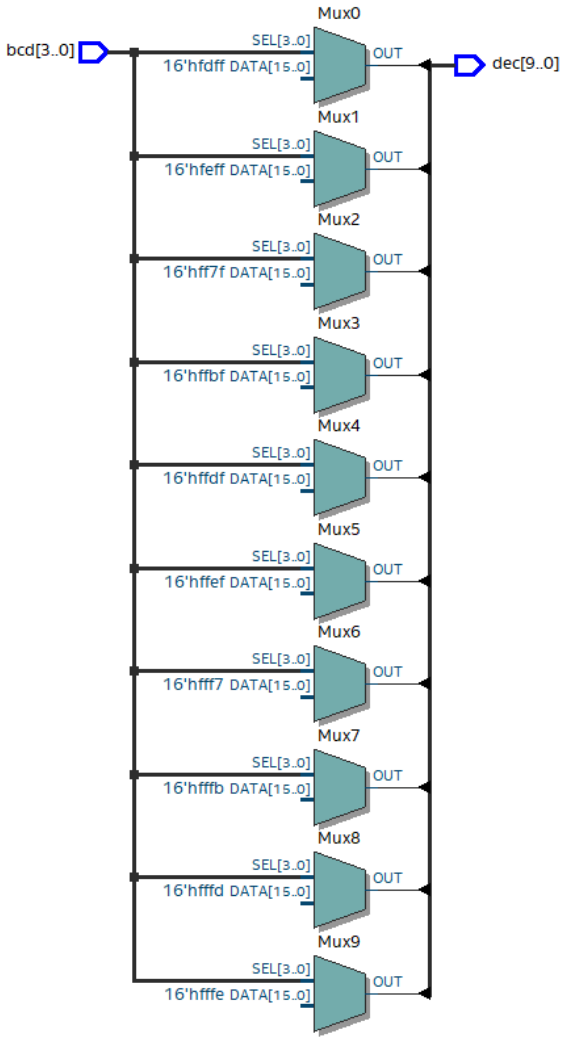

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity simple_decoder is
6      port ( A: in std_logic_vector(3 downto 0);
7             Y: out std_logic );
8  end entity simple_decoder;
9
10 architecture nine of simple_decoder is
11 begin
12     Y <= A(3) and (not A(2)) and (not A(1)) and A(0);
13 end architecture nine;
```

BCD-2-Decimal

- converts each BCD code (8421 code) into one of ten possible decimal digit indications
- frequently referred as a *4-line-to-10-line decoder* or a *1-of-10 decoder*

```
15  library IEEE;
16  use IEEE.STD_LOGIC_1164.ALL;
17
18  entity bcd2dec is
19      Port ( bcd : in std_logic_vector(3 downto 0);
20             dec : out std_logic_vector(9 downto 0));
21  end entity bcd2dec;
22
23  architecture Behavioral of bcd2dec is
24  begin
25     with bcd select
26      dec <= "1111111110" when "0000",     --0
27             "1111111101" when "0001",     --1
28             "1111111011" when "0010",     --2
29             "1111110111" when "0011",     --3
30             "1111101111" when "0100",     --4
31             "1111011111" when "0101",     --5
32             "1110111111" when "0110",     --6
33             "1101111111" when "0111",     --7
34             "1011111111" when "1000",     --8
35             "0111111111" when "1001",     --9
36             "1111111111" when others;     --
37  end architecture Behavioral;
```

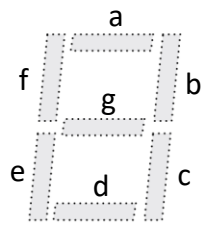




source: Thomas Floyd: Digital Fundamentals

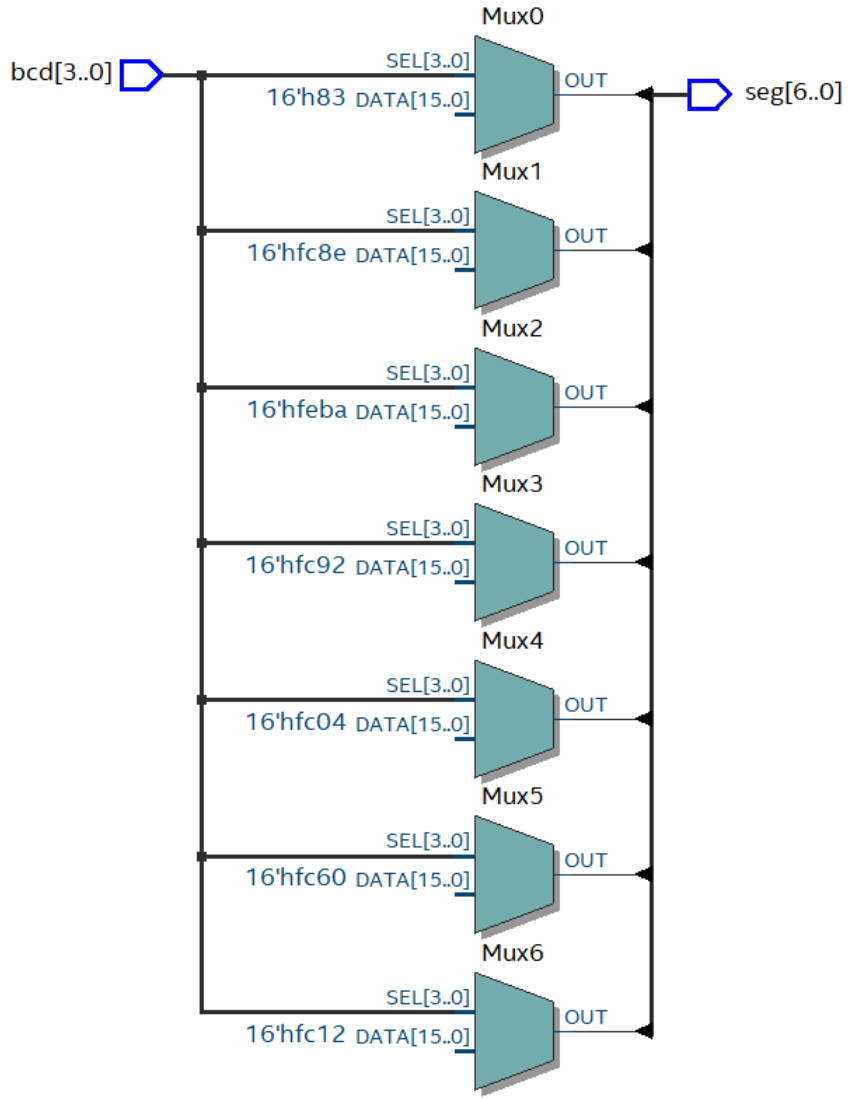
BCD-2-7Segment

- The BCD-to-7-segment decoder accepts the BCD code on its inputs and provides outputs to drive 7-segment display devices to produce a decimal readout



	seg_out(6:0)						
	g	f	e	d	c	b	a
1	1	1	1	1	0	0	1
2	0	1	0	0	1	0	0
3	0	1	1	0	0	0	0
4	0	0	1	1	0	0	1
5	0	0	1	0	0	1	0
6	0	0	0	0	0	1	0
7	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0
9	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0

```
42  entity dec7seg is
43      Port ( bcd : in std_logic_vector(3 downto 0);
44             seg : out std_logic_vector(6 downto 0)); --low
45  end entity dec7seg;
46
47  architecture ttable of dec7seg is
48  begin
49   with bcd select                    --        0
50    seg<= "1111001" when x"1",        --       -----
51          "0100100" when x"2",        --    5|     |1
52          "0110000" when x"3",        --     |  6  |
53          "0011001" when x"4",        --      -----
54          "0010010" when x"5",        --    4|     |2
55          "0000010" when x"6",        --     |     |
56          "1111000" when x"7",        --      -----
57          "0000000" when x"8",        --        3
58          "0010000" when x"9",        --
59          "1000000" when x"0",        --
60          "0111111" when others;      --
61  end architecture ttable;
```

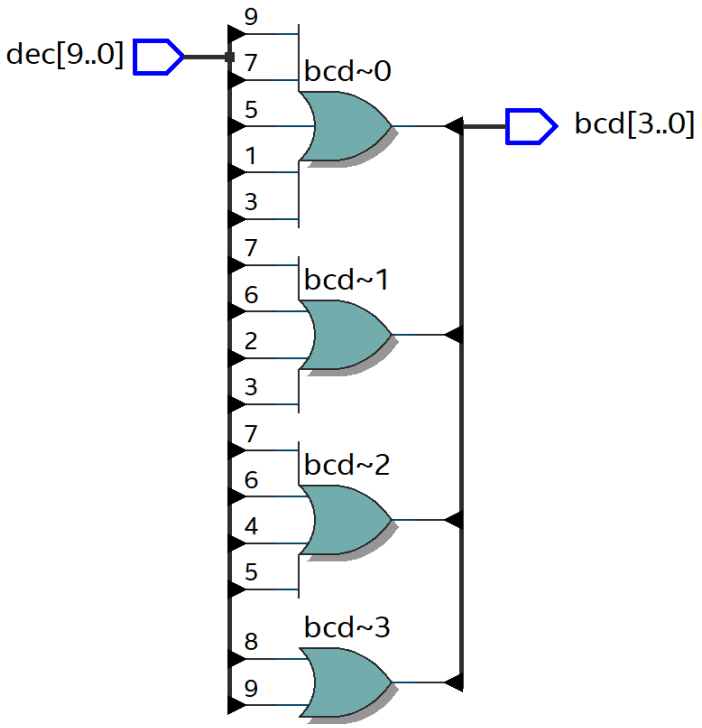


Encoders

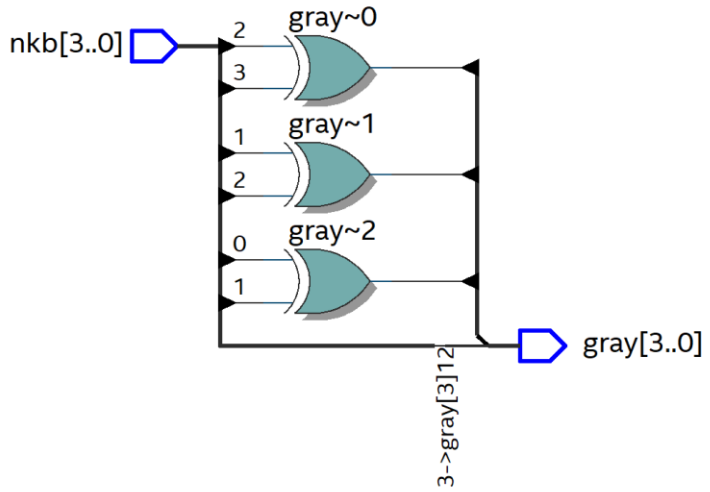
- An encoder is a combinational logic circuit that performs a “reverse” decoder function
- Accepts an active level on one of its inputs representing a digit and converts it to a coded output, such as BCD or binary

Decimal Digit	BCD Code			
	A_3	A_2	A_1	A_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

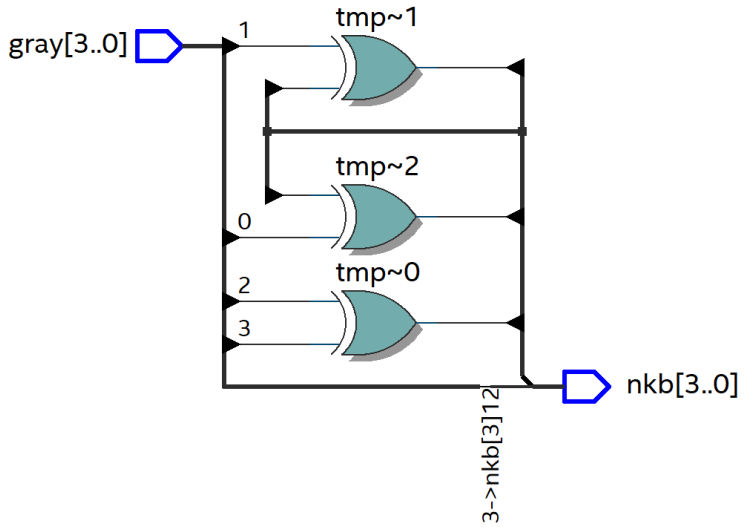
```
5   entity encoder is
6       Port ( dec : in std_logic_vector(9 downto 0);
7              bcd : out std_logic_vector(3 downto 0));
8   end entity encoder;
9
10  architecture equation of encoder is
11  begin
12      bcd(0) <= dec(1) or dec(3) or dec(5) or dec(7) or dec(9);
13      bcd(1) <= dec(2) or dec(3) or dec(6) or dec(7);
14      bcd(2) <= dec(4) or dec(5) or dec(6) or dec(7);
15      bcd(3) <= dec(8) or dec(9);
16  end architecture;
```



Code Converters



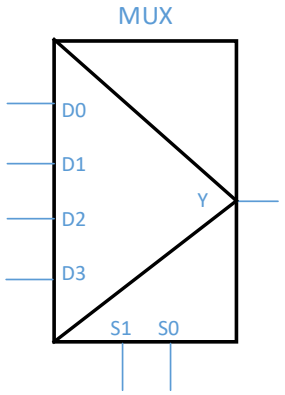

```
4  entity nkb2gray is
5      Port ( nkb : in  STD_LOGIC_VECTOR (3 downto 0); -- 0-lsb
6             gray : out  STD_LOGIC_VECTOR (3 downto 0)
7           );
8  end nkb2gray;
9
10 architecture Behavioral of nkb2gray is
11
12 begin
13  gray(3) <= nkb(3);
14  gray(2) <= nkb(2) xor nkb(3);
15  gray(1) <= nkb(1) xor nkb(2);
16  gray(0) <= nkb(0) xor nkb(1);
17
18 end Behavioral;
```




```
4  entity gray2nkb is
5      Port ( gray : in  STD_LOGIC_VECTOR (3 downto 0); -- 0-lsb
6             nkb : out  STD_LOGIC_VECTOR (3 downto 0)
7           );
8  end gray2nkb;
9
10 architecture Behavioral of gray2nkb is
11 signal tmp: std_logic_vector(3 downto 0);
12
13 begin
14  nkb <= tmp;
15  tmp(3) <= gray(3);
16  tmp(2) <= tmp(3) xor gray(2);
17  tmp(1) <= tmp(2) xor gray(1);
18  tmp(0) <= tmp(1) xor gray(0);
19
20 end Behavioral;
```

Multiplexer

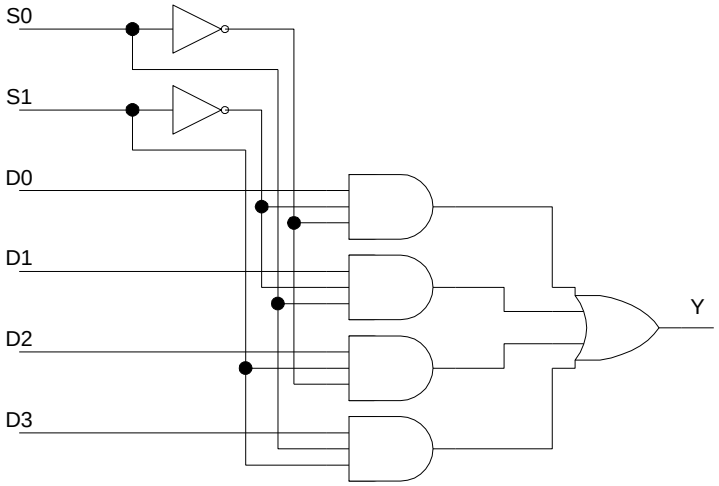
- A multiplexer (MUX) is a device that allows digital information from several sources to be routed onto a single line
- Multiplexers are also known as data selectors

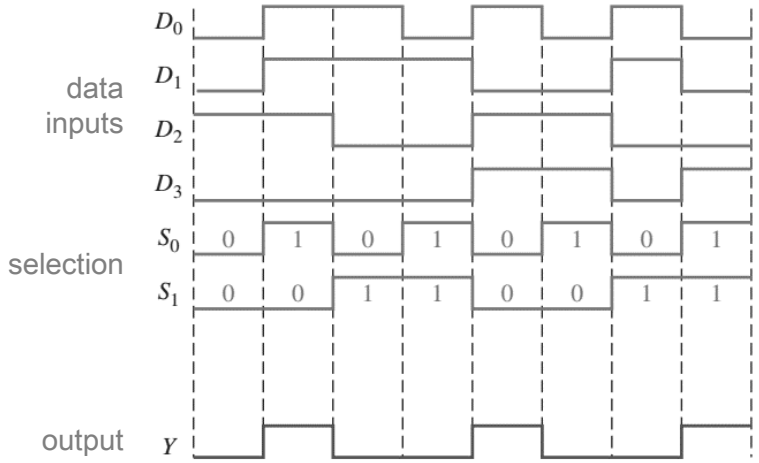


Data selection for a 1-of-4-multiplexer.

Data-Select Inputs		
S_1	S_0	Input Selected
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

$$Y = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1S_0 + D_2S_1\bar{S}_0 + D_3S_1S_0$$



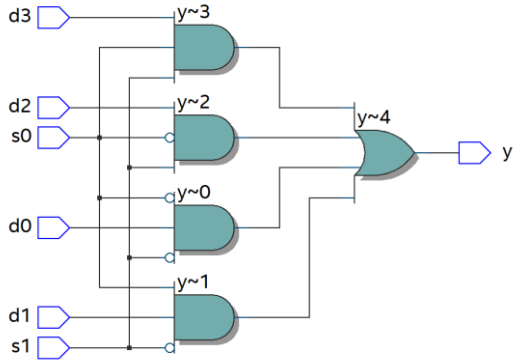


source: Thomas Floyd: Digital Fundamentals

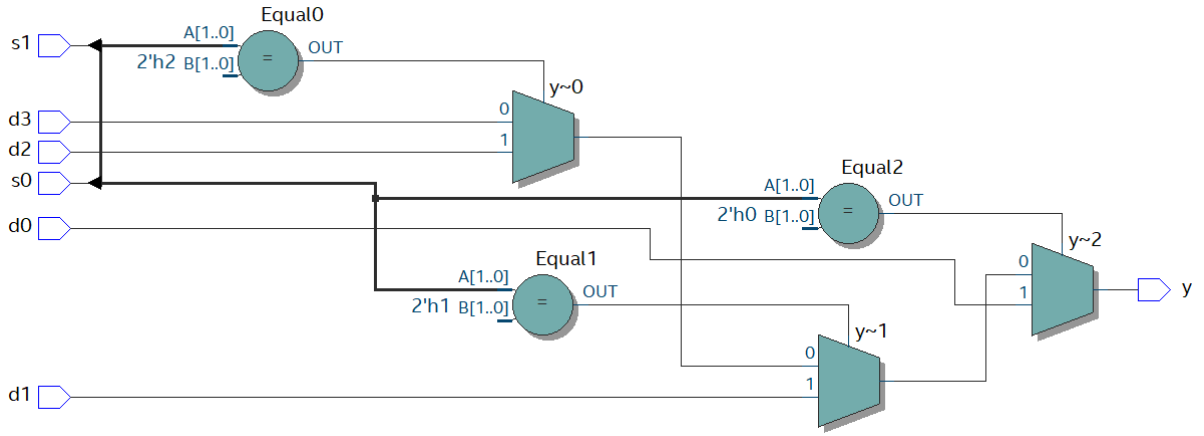
Multiplexer

```
5   entity mux4x1 is
6       Port ( d0,d1,d2,d3 : in std_logic;
7              s0,s1 : std_logic;
8              y : out std_logic);
9   end entity mux4x1;
```

```
11  architecture equation of mux4x1 is
12  begin
13  y <= (d0 and (not s1 and not s0) ) or
14       (d1 and (not s1 and     s0) ) or
15       (d2 and (    s1 and not s0) ) or
16       (d3 and (    s1 and     s0) );
17  end architecture;
```

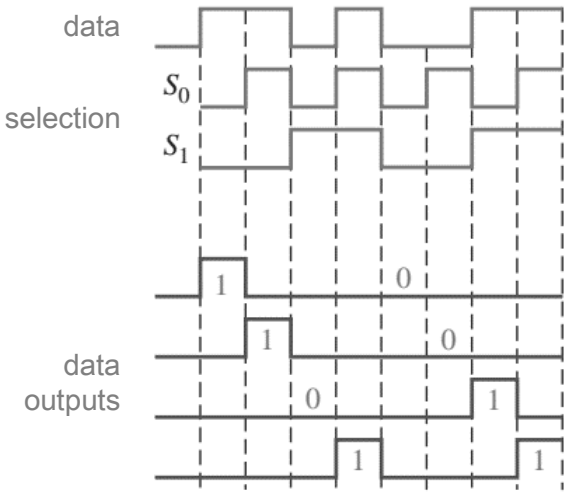


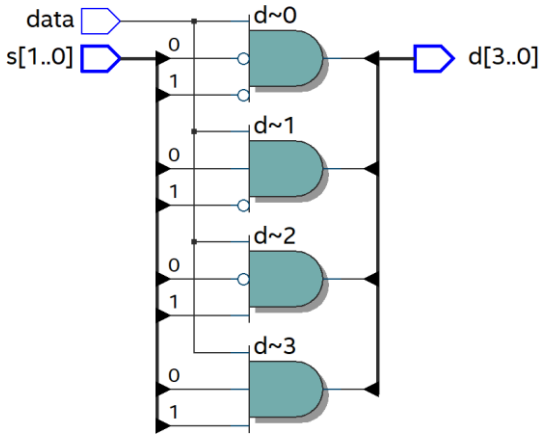

```
19  architecture behav of mux4x1 is
20  signal sel: std_logic_vector(1 downto 0);
21  begin
22  y <= d0 when (sel="00") else
23       d1 when (sel="01") else
24       d2 when (sel="10") else
25       d3 ;
26  sel <= s1 & s0;
27  end architecture;
```



Demultiplexer

- A demultiplexer (DEMUX) basically reverses the multiplexing function.
  It takes digital information from one line and distributes it to a given number of output lines
- The demultiplexer is also known as a data distributor
- Decoders can also be used as demultiplexers





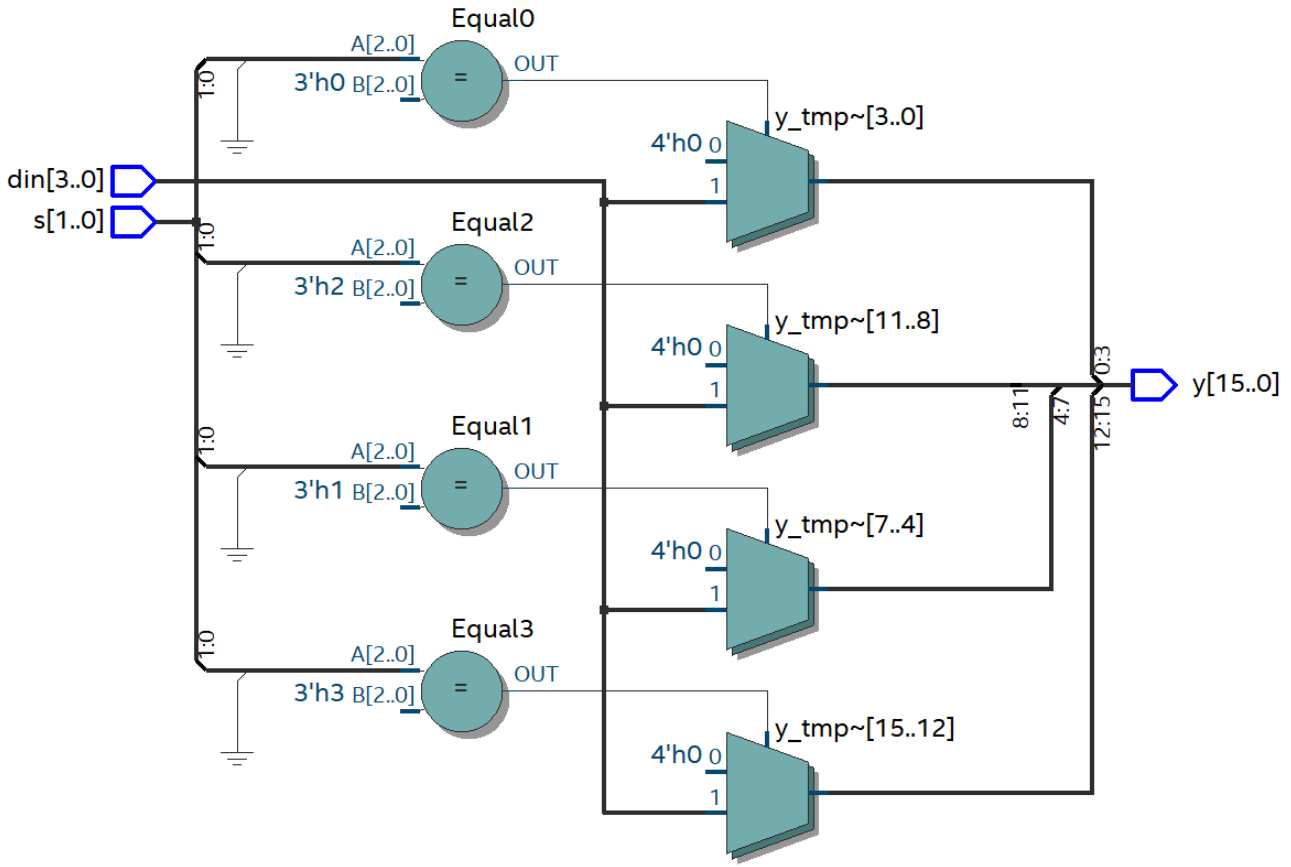

```
5  entity demux is
6      Port ( data : in std_logic;
7             s : in std_logic_vector(1 downto 0);
8             d : out std_logic_vector(3 downto 0));
9  end entity demux;
10
11 architecture equation of demux is
12     signal invs : std_logic_vector(1 downto 0);
13 begin
14     d(0) <= data and ( invs(0) and invs(1) );
15     d(1) <= data and (    s(0) and invs(1) );
16     d(2) <= data and ( invs(0) and    s(1) );
17     d(3) <= data and (    s(0) and    s(1) );
18     invs <= not s;
19 end architecture;
```


source: Thomas Floyd: Digital Fundamentals

Demultiplexer

- The demultiplexer is also known as a data distributor
- Decoders can also be used as demultiplexers

```
6   entity demux2 is
7     generic (delay: time := 3 ns);
8     port (s: in std_logic_vector(1 downto 0);
9           din: in std_logic_vector(3 downto 0);
10          y: out std_logic_vector(15 downto 0) );
11  end entity demux2;
12
13  architecture with_delay of demux2 is
14      signal y_tmp: std_logic_vector(y'range);
15  begin
16  y <= y_tmp after delay;
17  y_tmp(3 downto 0)   <= din when (s=0) else x"0";
18  y_tmp(7 downto 4)   <= din when (s=1) else x"0";
19  y_tmp(11 downto 8)  <= din when (s=2) else x"0";
20  y_tmp(15 downto 12) <= din when (s=3) else x"0";
21
22  end architecture with_delay;
```



Logic Function Generator

A useful application of the data selector/multiplexer is in the generation of combinational logic functions in sum-of-products form

- Case 1: F(A,B,C,D) = Σ{0,1,3,4,5,7,9,10,13,14}

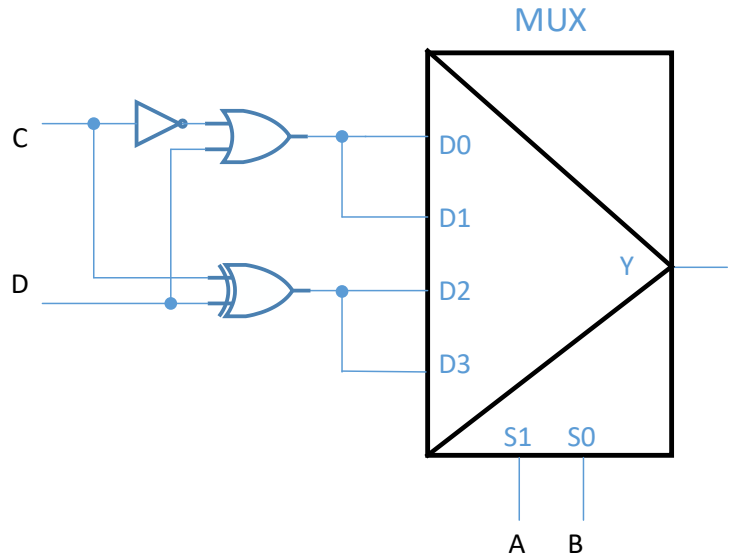
	A	B	C	D	F
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	
7	0	1	1	1	1
8	1	0	0	0	
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	
12	1	1	0	0	
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	

F0(AB=00) = /C or D

F1(AB=01) = /C or D

F2(AB=10) = /CD and C/D

F3(AB=11) = /CD and C/D



Logic Function Generator

- Case 2: F(A,B,C,D) = Σ{0,1,3,4,5,7,9,10,13,14}

1. determining the form of the minimal function
2. selection of address variables
3. determining equations for data inputs

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	0
11	0	1	0	1
10	0	1	0	1

1.

F =
/A/C +
/AD +
/CD +
AC/D

A	B	C	D
0	X	0	X
0	X	X	1
X	X	0	1
1	X	1	0
1	4	1	1

sel={A,C}
data={B,D}

2.

	A	B	C	D	F
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	
7	0	1	1	1	1
8	1	0	0	0	
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	
12	1	1	0	0	
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	

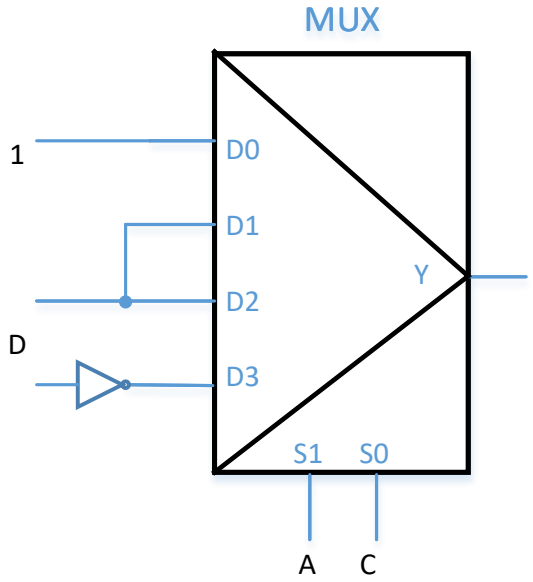
3.

F0(AC=00) = 1

F1(AC=01) = D

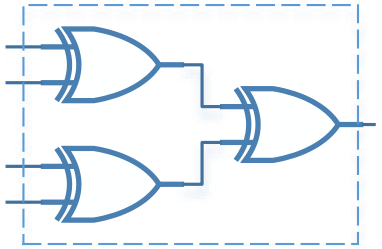
F2(AC=10) = D

F3(AC=11) = /D



Basic Parity Logic

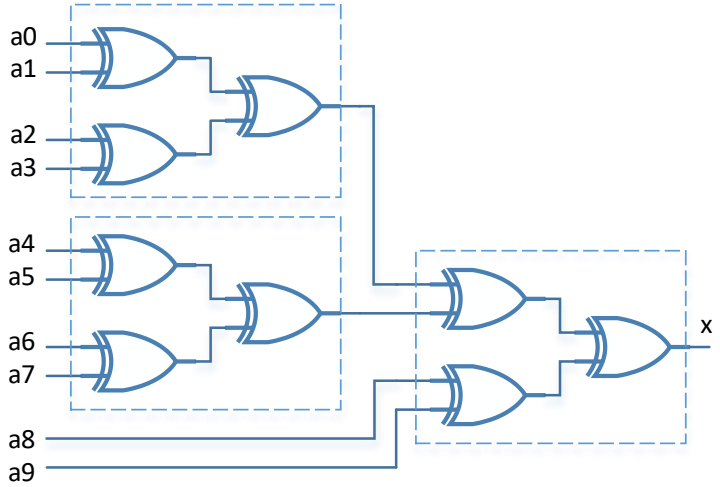
- A parity bit indicates if the number of 1s in a code is even or odd for the purpose of error detection.
- The sum (disregarding carries) of an even number of 1s is always 0, and the sum of an odd number of 1s is always 1.



Summing of four bits

- 10-bits parity checker

```
4   entity ParityCheck is
5       port(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9: in std_logic;
6            x: out std_logic);
7   end ParityCheck;
8
9   architecture gates of ParityCheck is
10  begin
11      x <= ((a0 xor a1) xor (a2 xor a3)) xor
12           ((a4 xor a5) xor (a6 xor a7)) xor
13            (a8 xor a9);
14  end gates;
```



Basic Parity Logic

```
4   entity reduced_xor is
5       generic(WIDTH: natural:=8);
6       port(a: in std_logic_vector(WIDTH-1 downto 0);
7            y: out std_logic);
8   end reduced_xor;
9
10  architecture gen_if_arch of reduced_xor is
11  signal tmp: std_logic_vector(WIDTH-2 downto 1);
12  begin
13      xor_gen: for i in 1 to (WIDTH-1) generate
14          left_gen: if i=1 generate                          -- leftmost stage
15             tmp(i) <= a(i) xor a(0);
16          end generate;
17          middle_gen: if (1<i) and (i < (WIDTH-1)) generate -- middle stages
18             tmp(i) <= a(i) xor tmp(i-1);
19          end generate;
20          right_gen: if i=(WIDTH-1) generate                 -- rightmost stage
21             y <= a(i) xor tmp(i-1);
22          end generate;
23      end generate;
24  end gen_if_arch;
```