Combinational Building Blocks

Prof. Dr. Miloš Krstić

Multi-valued Logic

- **Example - Three-State Buffer**

- **We can define the types with different values**

  type tri is (‘0’,’1’,’Z’);

  The signal can be then defined as

  Signal a, b, c: tri;

- **How to use tri same as bit signals?**

  For example how to calculate:

  B<= a and c after 5 ns;

- **Defining the function**

AND	0	1	Z
0	0	0	0
1	0	1	1
Z	0	1	Z

Overloading Operator

- We can define function which can overload standard operators (such as and)

```
function “and” (Left, Right:tri) return tri is
 type tri_array is array (tri,tri) of tri;
 constant and_table: tri_array:=((‘0’,’0’,’0’),
                                 (‘0’,’1’,’1’),
                                 (‘0’,’1’,’Z’));
begin
 return and_table(Left, Right);
end function “and”;
```

We cannot mix different operators now! (for example bit and tri)

Overloading must be used very carefully

Standard Logic Type

- **Normally the logic implemented in hardware is modelled with more than binary values (‘0’ and ‘1’)**

  ‘Z’ –high impedance

  ‘L’ – weak 0

  ‘H’ – weak 1

  ‘U’ – undefined

  ‘X’ – strong unknown

  ‘W’ – weak unknown

  ‘-’ – don’t care

  type std_ulogic is (‘U’,’X’,’0’,’1’,’Z’,’W’,’L’,’H’,’-’);

  How AND truth table for std_ulogic could be defined?

  subtype std_logic is resolved std_ulogic;

  Standard types and functions are organized in a package

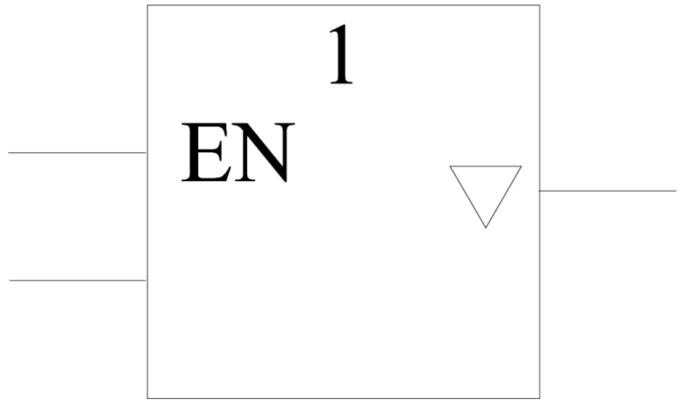
  We have to use them from the library

  library IEEE;

  use IEEE.std_logic_1164.all;

Building Three State Buffer

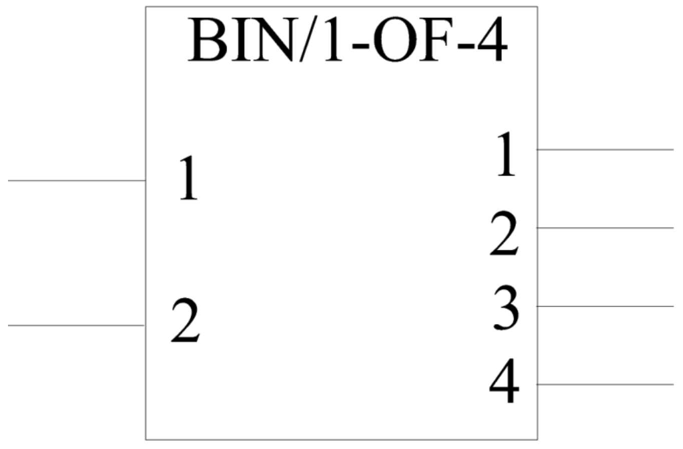
```
library IEEE;
use IEEE.std_logic_1164.all;
entity three_state is
  port (a, enable : in std_logic;
        z : out std_logic);
end entity three_state;

architecture when_else of three_state is
begin
  z <= a when enable = '1' else 'Z';
end architecture when_else;

architecture after_when_else of three_state is
begin
  z <= a after 4 NS when enable = '1' else 'Z';
end architecture after_when_else;
```



Decoder

- **Converts data into some other form**

Inputs		Outputs			
A1	**A0**	**Z3**	**Z2**	**Z1**	**Z0**
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



- **How to define vectors in VHDL**

type std_logic_vector is array (natural range <>) of std_logic;

Decoder

- One possibility for implementation is

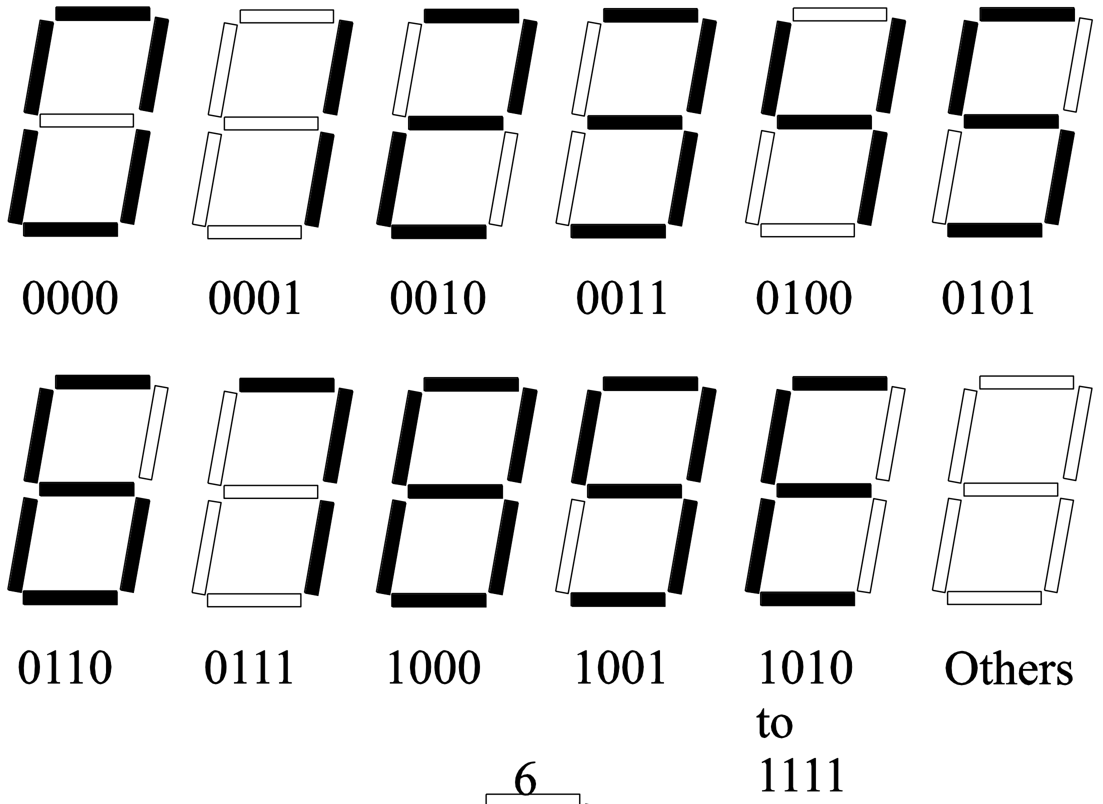
```
library IEEE;
use IEEE.std_logic_1164.all;
entity decoder is
  port (a : in std_logic_vector(1 downto 0);
        z : out std_logic_vector(3 downto 0));
end entity decoder;

architecture when_else of decoder is
begin
  z <= "0001" when a = "00" else
       "0010" when a = "01" else
       "0100" when a = "10" else
       "1000" when a = "11" else
       "XXXX";
end architecture when_else;
```

Decoder - alternative

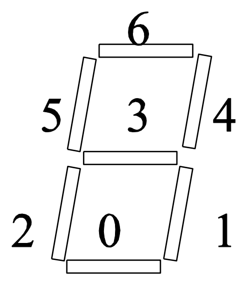
```
library IEEE;
use IEEE.std_logic_1164.all;
entity decoder is
  port (a : in std_logic_vector(1 downto 0);
        z : out std_logic_vector(3 downto 0));
end entity decoder;

architecture with_select of decoder is
begin
  with a select
    z <= "0001" when "00",
         "0010" when "01",
         "0100" when "10",
         "1000" when "11",
         "XXXX" when others;
end architecture with_select;
```

Practical example - Seven Segment Decoder

Please try at home and in labs!



Bit Ordering



Programmable Decoders

- **How to ensure parametric design?**

  Use of generics is essential in VHDL

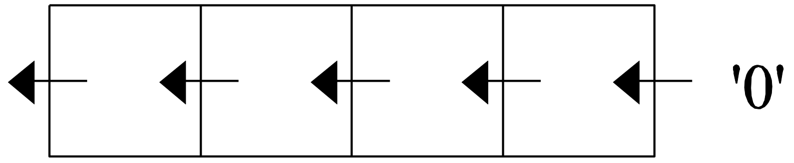
```
entity decoder is
  generic (n : POSITIVE);
  port (a : in std_logic_vector(n-1 downto 0);
        z : out std_logic_vector(2**n-1 downto 0));
end entity decoder;
```

- **How to use shift operator to build decoder?**

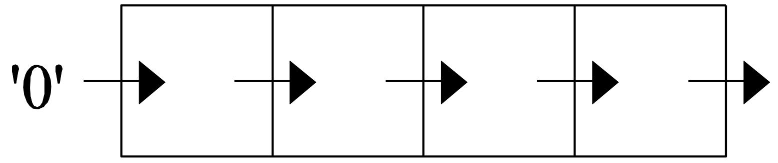
```
architecture rotate of decoder is
  constant z_out : BIT_VECTOR(2**n-1 downto 0) :=
                (0 => '1', others => '0');
begin
  z <= to_StdLogicVector(z_out sll to_integer(unsigned(a)));
end architecture rotate;
```

VHDL Shift Operators

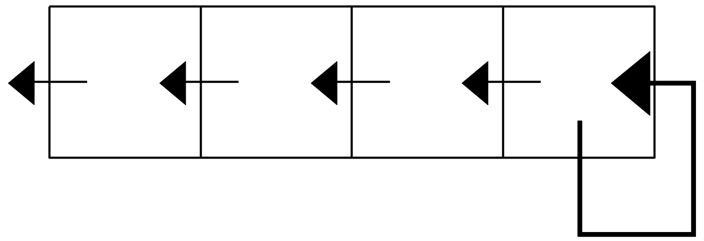
sll



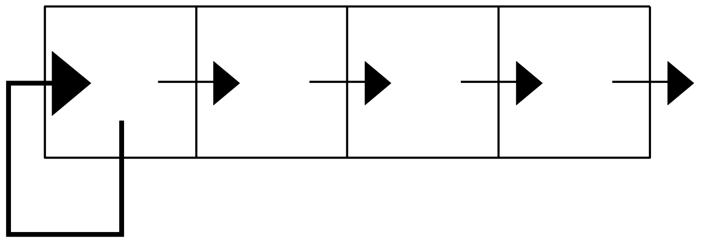
srl



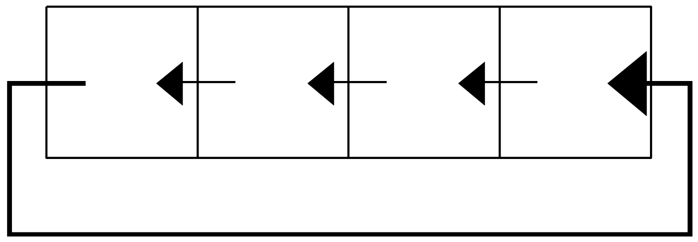
sla



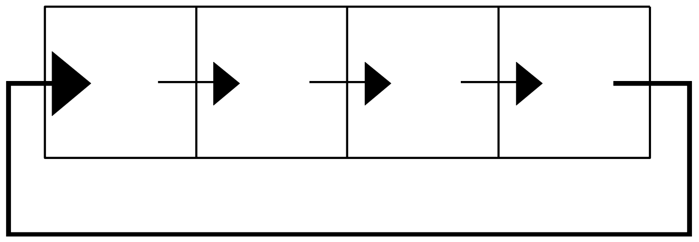
sra



rol



ror



Types of Arithmetic and Conversion in VHDL

- **Standard std_logic_1164 package supports std_logic_vectors and integers**
  For conversion one can use to_StdLogicVector and to_integer functions

- **For arithmetic we need definition whether this is signed or unsigned**
  This is supported by numeric_std package

  Conversion std_logic_vector ⇔ (un) signed is simple

  ```
  x<= unsigned(y);
  y<= std_logic_vector(x);
  ```

  From integer we need to define the number of bits:

  ```
  to_unsigned(i,n)
  ```

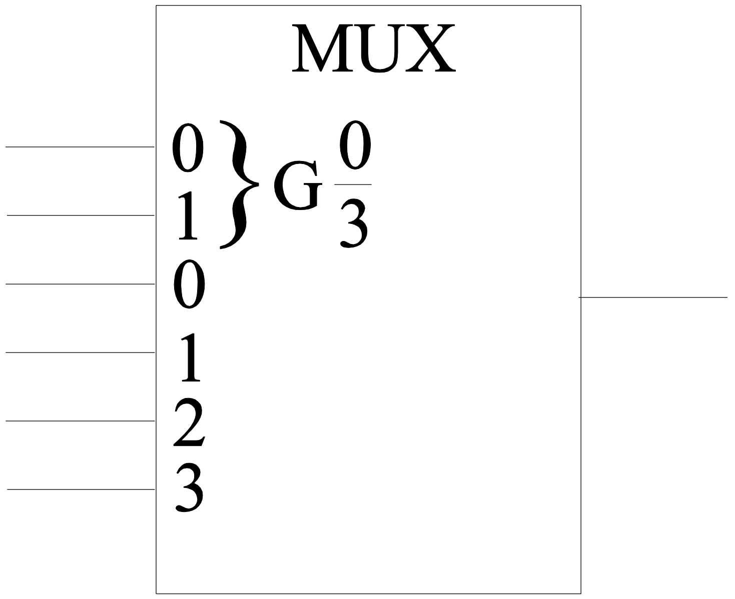
  Take care about the number of bits when using integers!

Multiplexers

- **Multiplexer switches one of many inputs to a single output**

  They enable effective reuse of hardware

```
entity mux is
  port (a, b, c, d: in std_logic;
        s: in std_logic_vector(1 downto 0);
        y: out std_logic);
end entity mux;

architecture mux1 of mux is
begin
  with s select
    y <= a when "00",
         b when "01",
         c when "10",
         d when "11",
         'X' when others;
end architecture mux1;
```



Priority Encoders

- In many situations the output of the combinational block is irrelevant to the complete set of inputs

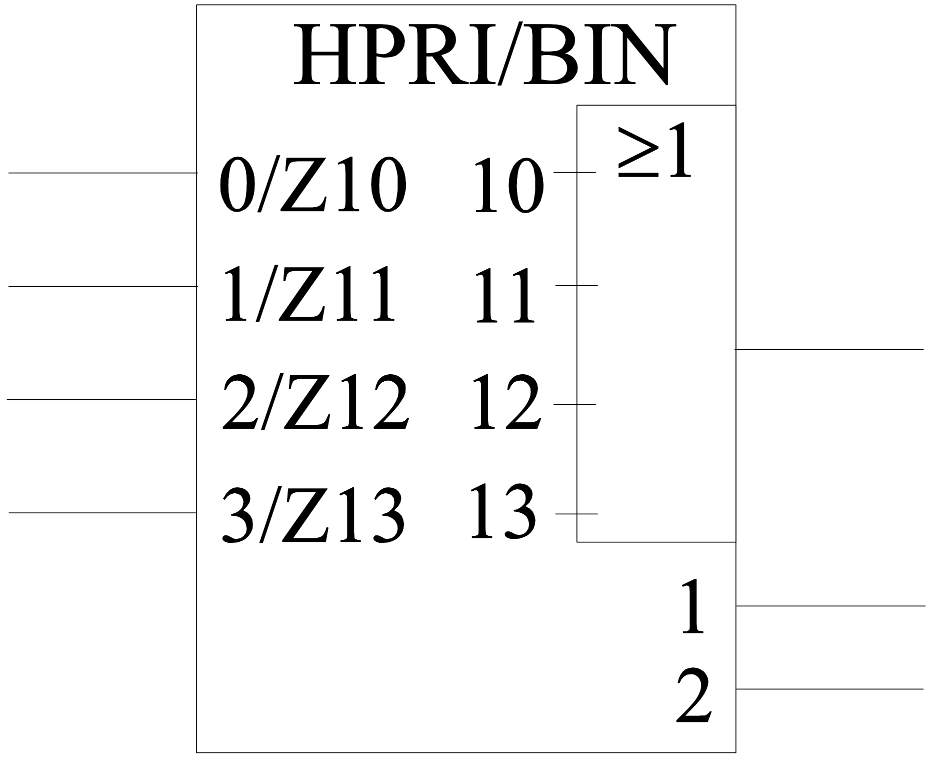
In this case we can use don’t care values

Example priority encoder

Inputs				Outputs		
A1	A2	A3	A4	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

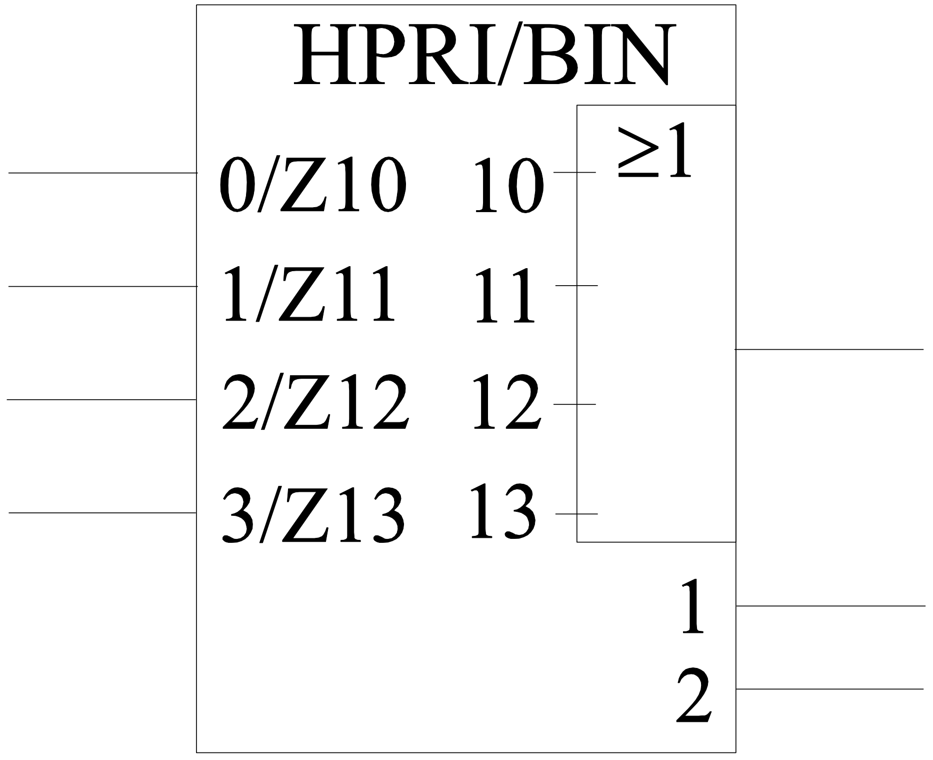
Priority Encoder - Implementation

```
library IEEE;
use IEEE.std_logic_1164.all;
entity priority is
   port (a: in std_logic_vector(3 downto 0);
        y: out std_logic_vector(1 downto 0);
        valid: out std_logic);
end entity priority;

architecture DontCare of priority is
begin
 with a select
   y <= "00" when "0001",
        "01" when "001-",
        "10" when "01--",
        "11" when "1---",
        "00" when others;
   valid <= '1' when a(0)='1' or a(1)='1' or a(2)='1' or a(3) = '1'
             else '0';
end architecture DontCare;
```



Priority Encoder – Alternative Implementation

```
library IEEE;
use IEEE.std_logic_1164.all;
entity priority is
   port (a: in std_logic_vector(3 downto 0);
        y: out std_logic_vector(1 downto 0);
        valid: out std_logic);
end entity priority;

architecture Ordered of priority is
begin
  y <= "11" when a(3)='1' else
      "10" when a(2)='1' else
      "01" when a(1)='1' else
      "00" when a(0)='1' else
      "00";
   valid <= '1' when a(0)='1' or a(1)='1' or a(2)='1' or a(3) = '1'
           else '0';
end architecture Ordered;
```



VHDL Modelling Styles

- **Three ways of VHDL coding, two have been covered before**

  Structural – netlist type

  u1: inv port map (A, X);

  Dataflow – concurrent signal assignments

  out <= ‘0’ when In1=In2 else ‘1’;

  Sequential – can be used in functions, procedures and processes

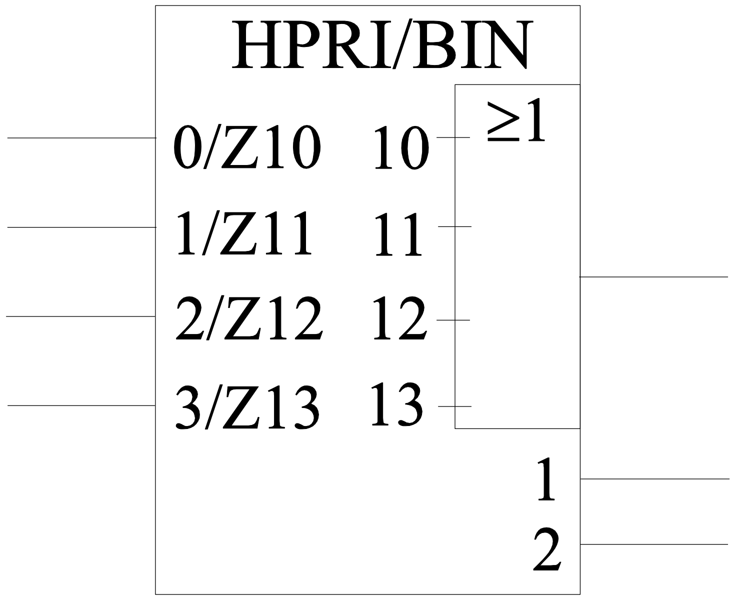
  Process – one of the most important structures in VHDL

  Sensitivity list

  if statements

Priority Encoder – sequential Implementation

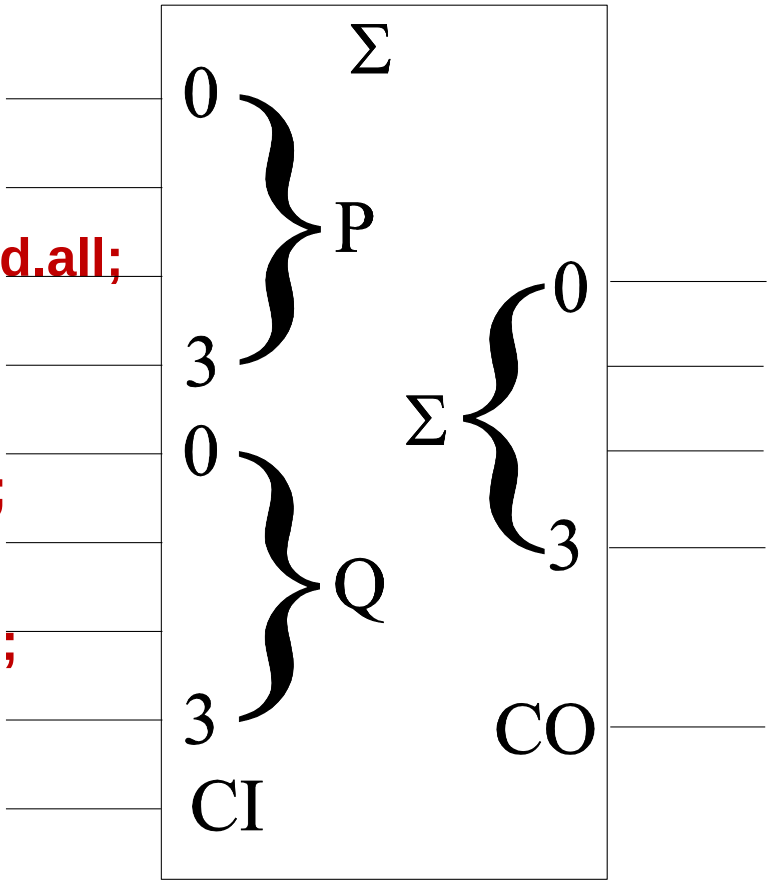
```
library IEEE;
use IEEE.std_logic_1164.all;
entity priority is port (a: in std_logic_vector(3 downto 0); y: out std_logic_vector(1 downto 0); valid: out
std_logic); end entity priority;

architecture Sequential of priority is
begin
  process (a) is
  begin
    if a(3)='1' then
      y <= "11"; valid <= '1';
    elsif a(2)='1' then
      y <= "10"; valid <= '1';
    elsif a(1)='1' then
      y <= "01";  valid <= '1';
    elsif a(0)='1' then
      y <= "00";  valid <= '1';
    else
      y <= "00"; valid <= '0';
    end if;
  end process;
end architecture Sequential;
```



Adders

- **Entity declaration**

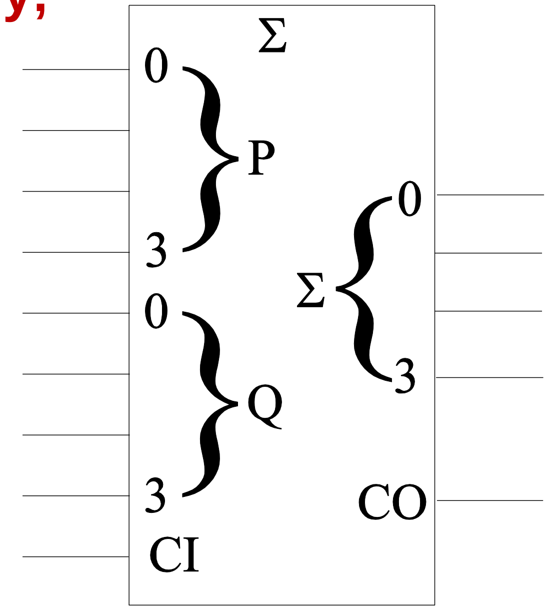
```
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
entity NBitAdder is
  generic (n: NATURAL :=4);
  port(A, B: in std_logic_vector(n-1 downto 0);
       Cin : in std_logic;
       Sum : out std_logic_vector(n-1 downto 0);
       Cout: out std_logic);
end entity NBitAdder;
```



- **Two types of addition**
  - **Signed**
  - **Unsigned**

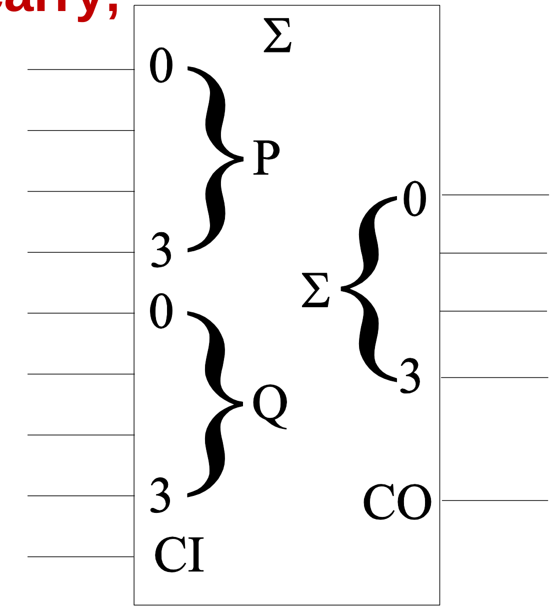
Adders – Unsigned Implementation

```
architecture unsgned of NBitAdder is
  signal result : unsigned(n downto 0);
  signal carry : unsigned(n downto 0);
  constant zeros : unsigned(n-1 downto 0) := (others => '0');
begin
  carry <= (zeros & Cin);
  result <= ('0' & unsigned(A)) + ('0' & unsigned(B)) + carry;
  Sum <= std_logic_vector(result(n-1 downto 0));
  Cout <= result(n);
end architecture unsgned;
```



Adders – Signed Implementation

```
architecture sgned of NBitAdder is
  signal result : signed(n downto 0);
  signal carry : signed(n downto 0);
  constant zeros : signed(n-1 downto 0) := (others => '0');
begin
  carry <= (zeros & Cin);
  result <= (A(n-1) & signed(A)) + (B(n-1) & signed(B)) + carry;
  Sum <= std_logic_vector(result(n-1 downto 0));
  Cout <= result(n);
end architecture sgned;
```



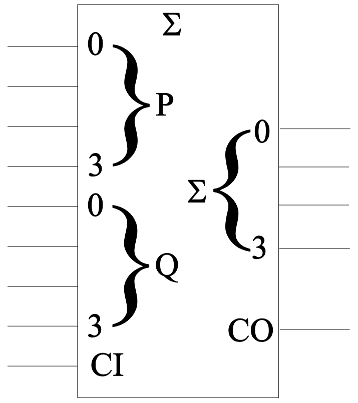
Single Bit Full Adder

```
library IEEE;
use IEEE.std_logic_1164.all;
entity FullAdder is
  port (a, b, Cin : in std_logic; Sum, Cout: out std_logic);
end entity FullAdder;

architecture concurrent of FullAdder is
begin
  Sum <= a xor b xor Cin;
  Cout <= (a and b) or (a and Cin) or (b and Cin);
end architecture concurrent;
```

- **Two assignments are concurrent!**

  This is a general rule in VHDL

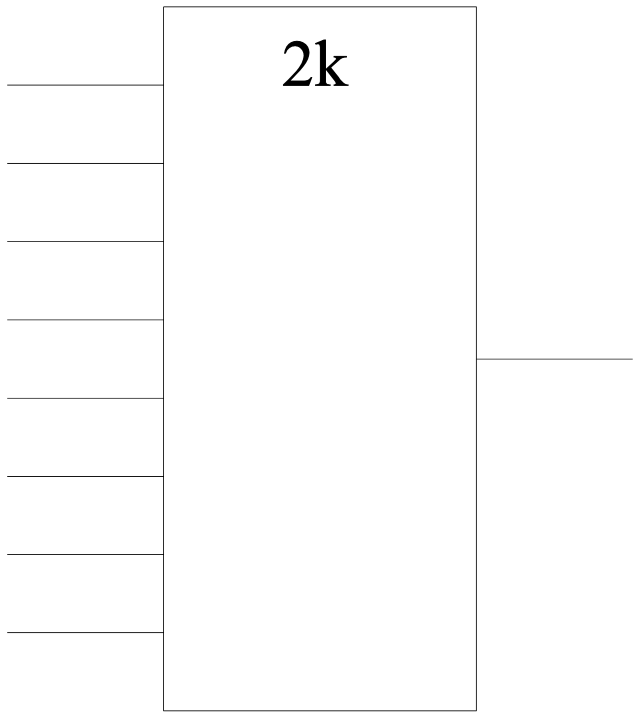
Adder – Iterative Implementation



```
architecture StructIterative of NBitAdder is
  signal Carry: std_logic_vector(0 to n);
begin
  g1 : for i in 0 to n-1 generate
    lt : if i = 0 generate
           f0 : entity work.FullAdder port map (A(i), B(i), Cin, Sum(i), Carry(i+1));
         end generate lt;
    rt : if i = n-1 generate
           fn : entity work.FullAdder port map (A(i), B(i), Carry(i), Sum(i), Cout);
         end generate rt;
    md : if i > 0 and i < n-1 generate
           fm : entity work.FullAdder port map (A(i), B(i), Carry(i), Sum(i), Carry(i+1));
         end generate md;
    end generate g1;
end architecture StructIterative;
```

Parity Checker

```
 library IEEE;
use IEEE.std_logic_1164.all;
entity parity is
  port (a : in std_logic_vector;
        y : out std_logic);
end entity parity;

architecture iterative of parity is
begin
  process (a) is
    variable even : std_logic;
  begin
    even := '0';
    for i in a'RANGE loop
      if a(i) = '1' then
        even := not even;
      end if;
    end loop;
    y <= even;
  end process;
end architecture iterative;
```



New VHDL constructs

attribute RANGE

use of variables

Immediately Update

for loop

Principle of loop unrolling

Testbenches for Combinational Blocks

- **Two important functions of a testbench**

  Generation of input stimuli

  Checking of results validity

- **Example Adder testbench**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity TestNBitAdder is
end entity TestNBitAdder;

architecture TestBench_1 of TestNBitAdder is
 constant n: NATURAL := 4;
 signal A, B, Sum: std_logic_vector (n-1 downto 0);
 signal Cin, Cout: std_logic;
begin
 s0: entity WORK.NBitAdder(unsgned) generic map (n) port map(A, B, Cin, Sum, Cout);
 Cin <= '0', '1' after 10 NS, '0' after 25 NS;
 A <= "0000", "1111" after 5 NS, "0111" after 15 NS;
 B <= "0000", "1111" after 20 NS;
end architecture TestBench_1;
```

Testbenches in a Process

```
architecture TestBench_3 of TestNBitAdder is
 constant n: NATURAL := 4;
 signal A, B, Sumint : NATURAL;
 signal Aslv, Bslv, Sum: std_logic_vector (n-1 downto 0);
 signal Cin, Cout: std_logic;
begin
 s0: entity WORK.NBitAdder(unsgned) generic map (n) port map(Aslv, Bslv, Cin, Sum, Cout);
     Aslv <= std_logic_vector(to_unsigned(A, n));   Bslv <= std_logic_vector(to_unsigned(B, n));
     Sumint <= to_integer(unsigned(Cout & Sum));
stim:   process is
 begin
        Cin <= '0';
        A <= 0;
        B <= 0;
        wait for 5 NS;
        A <= 15;
        wait for 5 NS;
        Cin <= '1';
        wait for 5 NS;
        A <= 7;
        wait for 5 NS;
        B <= 15;
        wait for 5 NS;
        Cin <= '0';
        wait;
 end process;
end architecture TestBench_3;
```

Testbench – monitoring of the results

```
architecture TestBench_4 of TestNBitAdder is
 constant n: NATURAL := 4;
 signal A, B, Sumint : NATURAL; signal Aslv, Bslv, Sum: std_logic_vector (n-1 downto 0);
 signal Cin, Cout: std_logic; signal error: BOOLEAN := FALSE;
begin
 s0: entity WORK.NBitAdder(unsgned) generic map (n) port map(Aslv, Bslv, Cin, Sum, Cout);
     Aslv <= std_logic_vector(to_unsigned(A, n));      Bslv <= std_logic_vector(to_unsigned(B, n));
     Sumint <= to_integer(unsigned(Cout & Sum));
stim:   process is
 begin
          Cin <= '0'; A <= 0; B <= 0;
          wait for 5 NS;
          A <= 15;
          wait for 5 NS;
          Cin <= '1';
          wait for 5 NS;
          A <= 7;
          wait for 5 NS;
          B <= 15;
          wait for 5 NS;
          Cin <= '0';
          wait;
 end process;
resp:   process (Cout, Sum) is
     begin
          error <= (A + B + BIT'POS(to_bit(Cin))) /= Sumint;
     end process;
end architecture TestBench_4;
```

Conclusions

- We have learnt a number of basic combinational logic blocks
- In order to define them several VHDL constructs have been introduced
  - Packages
  - When…else
  - With…select
  - Generate
  - Shift operators
  - Processes
- Testbench generation is also presented