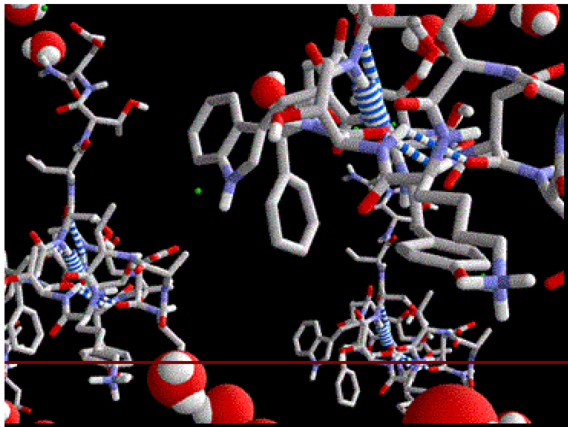
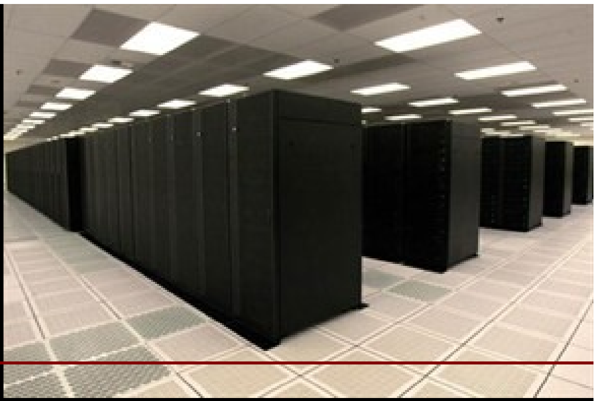
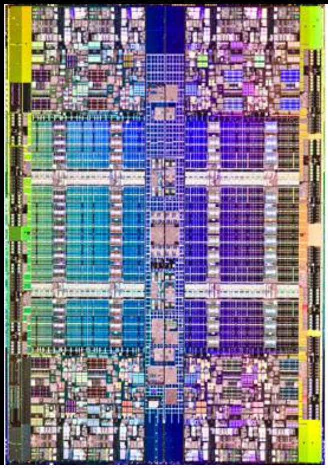



CS2204 Digital Logic and State Machine Design

VHDL Programming

Vijay Polavarapu

Fall 2013



POLYTECHNIC INSTITUTE OF NYU

Agenda

Acknowledgements

- Prof. Haldun Hadimioglu
- John Wakerly, Cisco Systems, Stanford University
- System Design using VHDL-Charles H. Roth

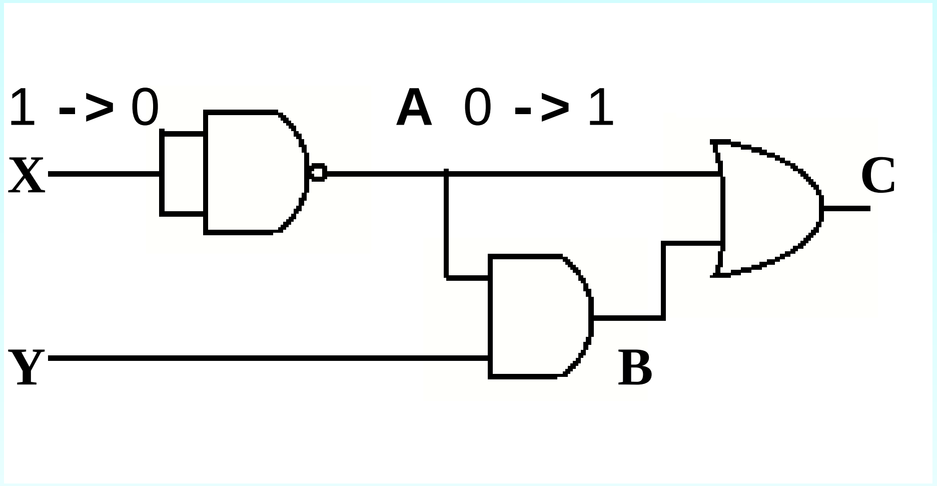
VHDL

Revisited

Why HDLs?

- In software everything is sequential
- Sequence of statements is significant, since they are executed in that order
- In hardware events are concurrent, so a software language cannot be used for describing and simulating hardware.

e.g. C = (not (X) and Y) or (not (X))



Different outputs with software programming languages with ‘0’ initial values

Case 1
A = not X
B = A and Y
C = A or B
Result:
C = 1

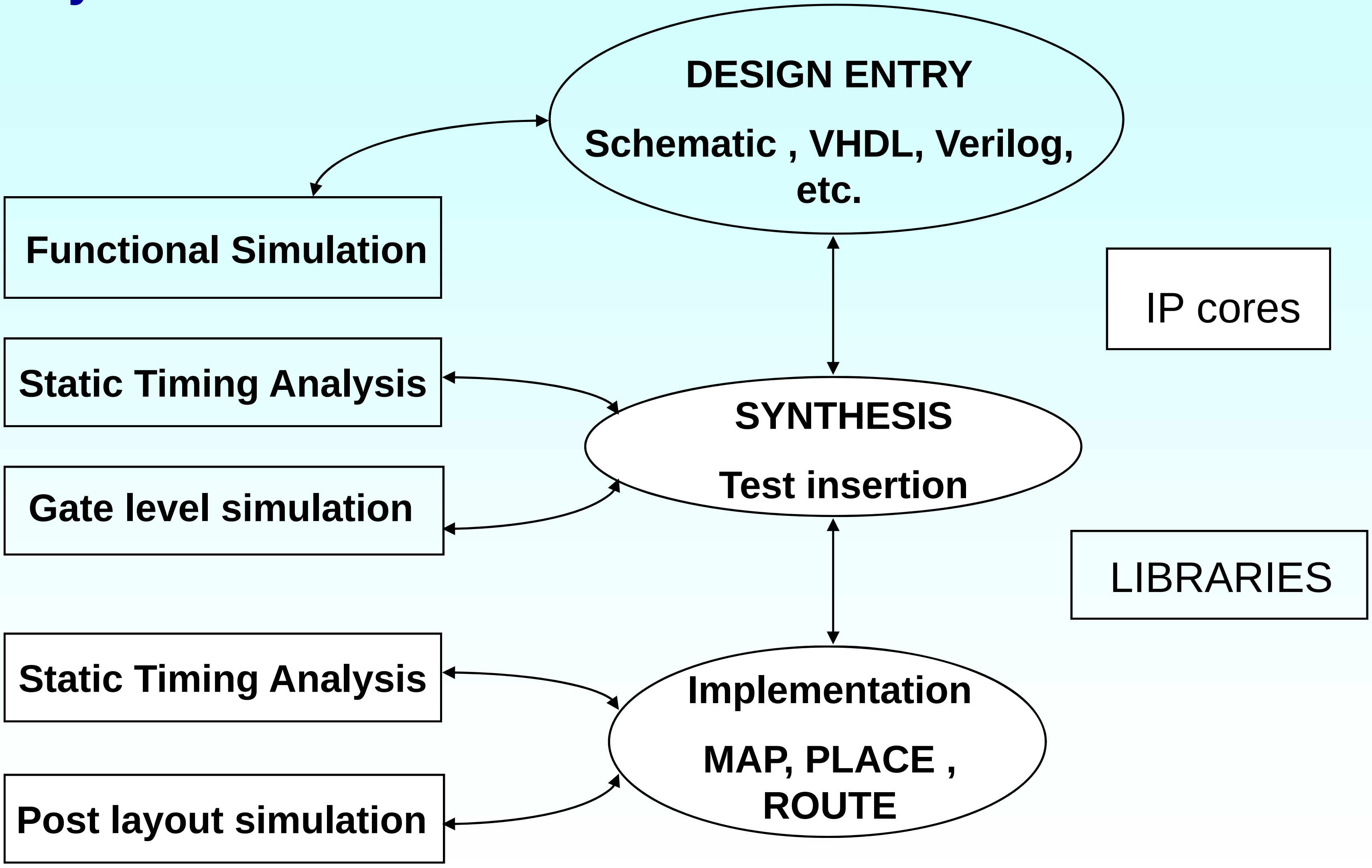
Case 2
B = A and Y
C = A or B
A = not X
Result:
C = 0

Case 3
C = A or B
A = not X
B = A and Y
Result:
C = 0

Features of HDLs

- Concurrent Descriptions
- Synchronizing mechanisms between concurrent flows
- Event Scheduling
- Special object types and data types
- Hierarchy

HDL Implementation Design Cycle



Advantages of using Hardware Description Languages

- Designs can be described at various levels of abstractions
- Top-Down Approach and hierarchical designs for large projects
- Functional Simulation Early in the Design Flow
- Automatic Conversion of HDL Code to Gates

  With user level control. Consistent quality. Fast.
- Early Testing of Various Design Implementations

  Due to fast synthesis, there is a scope for trying different implementations.
- Design Reuse

  Technology independence, standardization, portability, ease of maintenance.

All these result in low risk, high convergence, fast time to market, more money.

A Brief History Of VHDL

- VHDL stands for Very high speed integrated circuit Hardware Description Language
- Funded by the US Department of Defense in the 80's
- Originally meant for design standardisation, documentation, simulation and ease of maintenance.
- Established as IEEE standard IEEE 1076 in 1987. An updated standard, IEEE 1164 was adopted in 1993. In 1996 IEEE 1076.3 became a VHDL synthesis standard.
- Today VHDL is widely used across the industry for design description, simulation and synthesis.

About VHDL

- VHDL is not case sensitive
- VHDL is a free form language. You can write the whole program on a single line.

```
-- This is a VHDL comment
entity my_exor is -- one more comment
begin
...
end my_exor;
```

my EXOR gate

```
-- This is my first VHDL program

library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;
```

entity declaration - describes the boundaries of the object.
It defines the names of the ports, their mode and their type.

my EXOR gate

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;
```

entity - defines the interface.

Mode of the port :
Direction of flow.
It can be
in, out or inout

my EXOR gate

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;
```

entity - defines the interface.

Mode of the port :
It can be
in, out or inout

std_logic is the type of the port.
Standard logic is defined by the standard IEEE 1164.
It is defined in the IEEE library.
Any node of type std_logic can take 9 different values.
‘0’ , ’1’ , ’H’ , ’L’ , ’Z’ ,
’U’ , ’X’ , ’W’ , ’-’

my EXOR gate

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;
```

Library : Collection of design elements, type declarations, sub programs, etc.

my EXOR gate

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;

architecture my_exor_beh of my_exor is
begin
  op1 <= (ip1 and (not ip2)) or
             (ip2 and (not ip1));
end my_exor_beh;
```

Library : Collection of design elements, type declarations,sub programs, etc.

entity - defines the interface.

std_logic is the type of the port
It is defined in the IEEE library.
Any node of type std_logic can take 9 different values.
‘0’ , ’1’ , ’H’ , ’L’ , ’Z’ , ’U’ , ’X’ , ’W’ , ’-’

Mode of the port :
It can be
in, out or inout

The architecture describes the behaviour (function), interconnections and the relationship between different inputs and outputs of the entity.

my EXOR gate

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;


architecture my_exor_beh of my_exor is
begin
  op1 <= (ip1 and (not ip2)) or
             (ip2 and (not ip1));
end my_exor_beh;

configuration my_exor_C of my_exor is
  for my_exor_beh
  end for;
end my_exor_C;
```

Library : Collection of design elements, type declarations, sub programs, etc.

entity - defines the interface.

std_logic is the type of the port
It is defined in the IEEE library.
Any node of type std_logic can take 9 different value.
'0' , '1' , 'H' , 'L' , 'Z' , 'U' , 'X' , 'W' , '-'

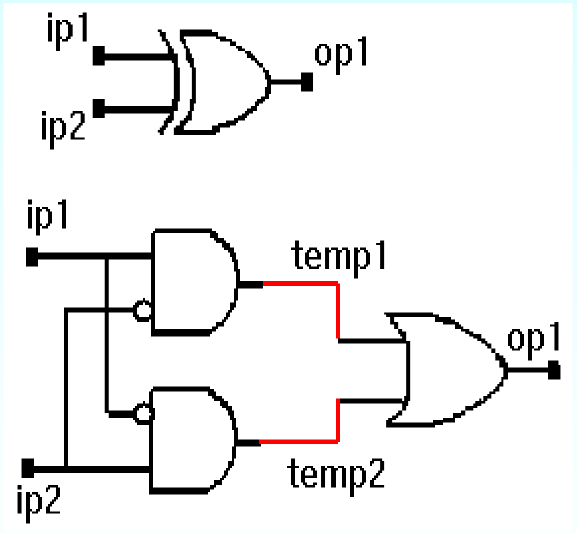
Mode of the port :
It can be
in, out or inout

The architecture describes the behaviour(function), interconnections and the relationship between different inputsand outputs.

The configuration is optional.
It defines the entity architecture bindings.
More about configurations later.

Internal connections are made using signals. Signals are defined inside the architecture.

```
architecture my_exor_beh of my_exor is
  signal temp1 : std_logic;
  signal temp2 : std_logic;
begin
  ......
end my_exor_beh;
```



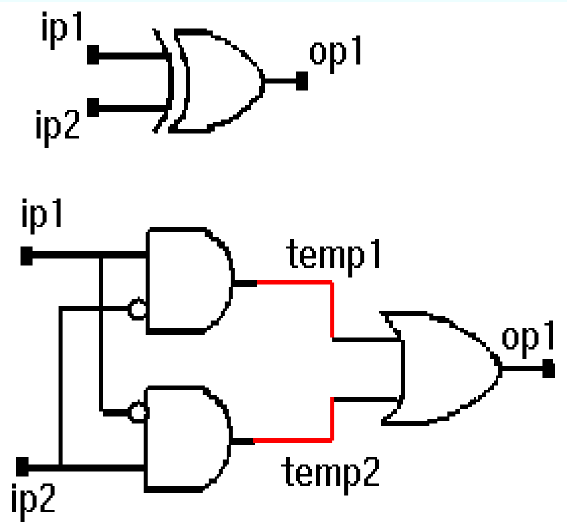
my EXOR with internal signals

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_exor is
port (ip1   : in  std_logic;
      ip2   : in  std_logic;
      op1   : out std_logic
     );
end my_exor;

architecture exor_w_sig of my_exor is
  signal temp1, temp2 : std_logic;
begin
  temp1 <= ip1 and (not ip2);
  temp2 <= ip2 and (not ip1);
  op1   <= temp1 or temp2;
end exor_w_sig;

configuration my_exor_C of my_exor is
  for exor_w_sig
  end for;
end my_exor_C;
```



SUMMARY

Introduction to:

- VHDL flow
- Comments
- Library declaration
- Entity declaration (ports, modes, std_logic type)
- Architecture
- Signal declarations
- Signal assignments
- Component declaration and instantiation
- Configuration statement

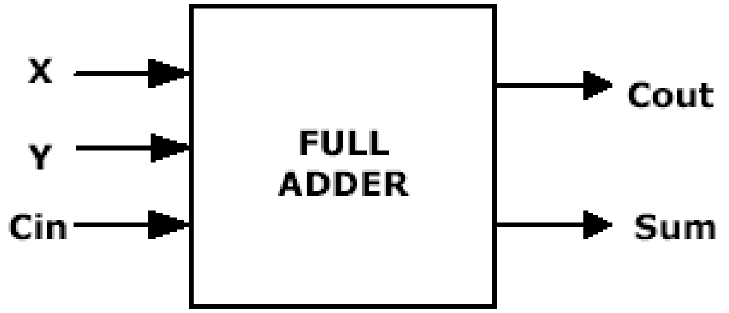
Design Hierarchy Levels (Modeling Styles)

- Structural
  - ➢ Define explicit components and the connections between them.
- Dataflow
  - ➢ Most are like assigning expressions to signals
- Behavioral
  - ➢ Write an algorithm that describes the circuit's output

Dataflow Level

- Dataflow description
  - The detail is less with data dependencies described, not the components and connections
  - Includes “when” and “select” (case) statements

Full Adder - Data flow



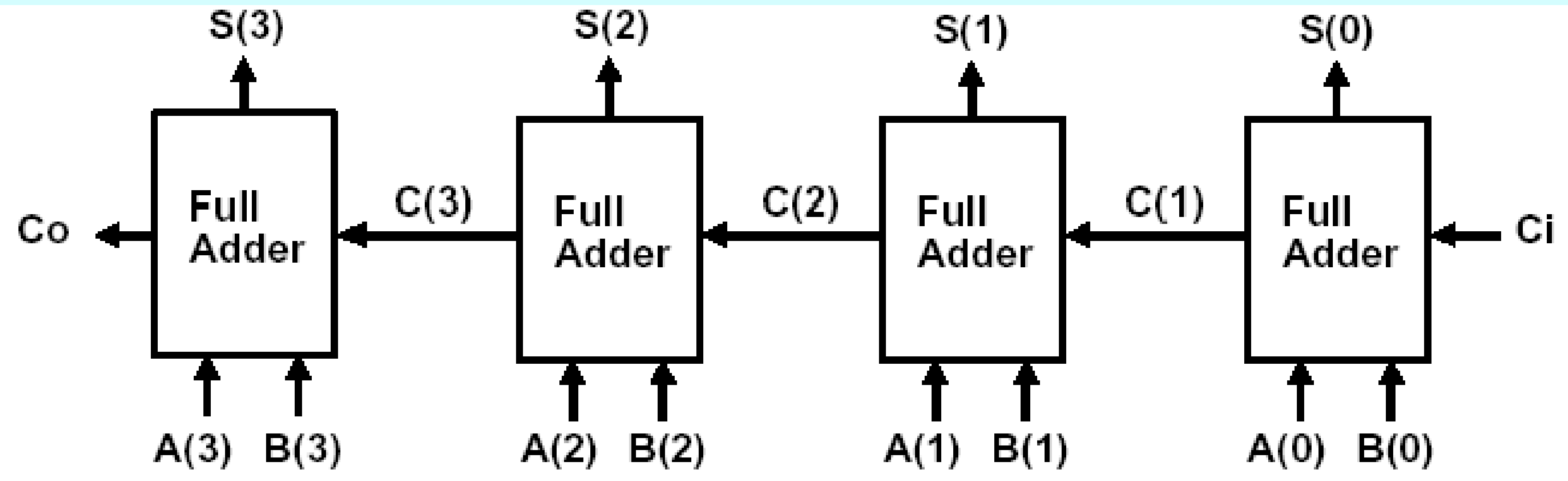
```
entity FullAdder is
  port (X, Y, Cin: in bit;       -- Inputs
    Cout, Sum: out bit);         -- Outputs
end FullAdder;

architecture Equations of FullAdder is
begin                          -- Concurrent Assignments
  Sum  <= X xor Y xor Cin ;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) ;
end Equations;
```

Structural Level

- A structural description is like the schematic, describing the components and their interconnections precisely
  - ➢ Includes concurrent statements
    - A component statement is a concurrent statement

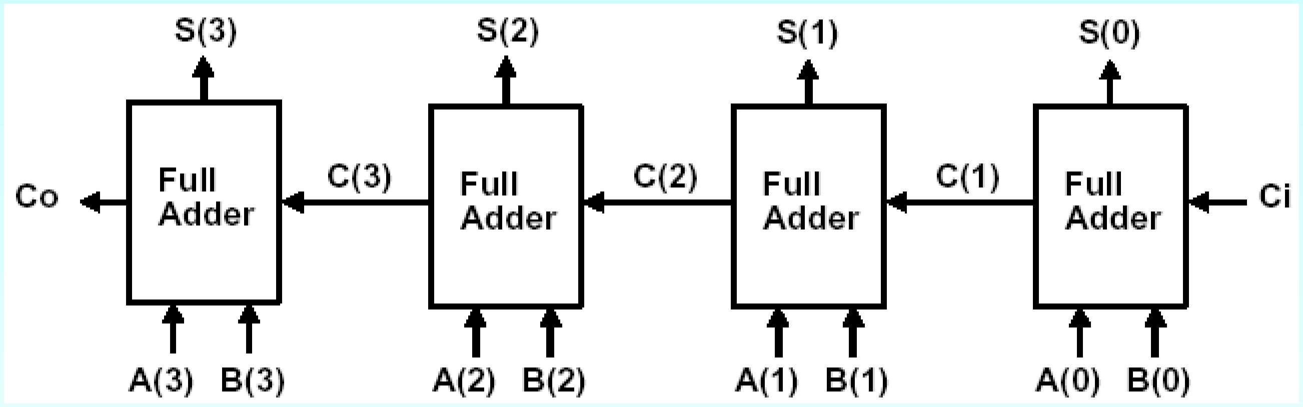
4-bit Ripple-Carry Adder - Structural Description



Structural Description of 4-bit Adder

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;       -- Inputs
         S: out bit_vector(3 downto 0); Co: out bit);      -- Outputs
end Adder4;
```

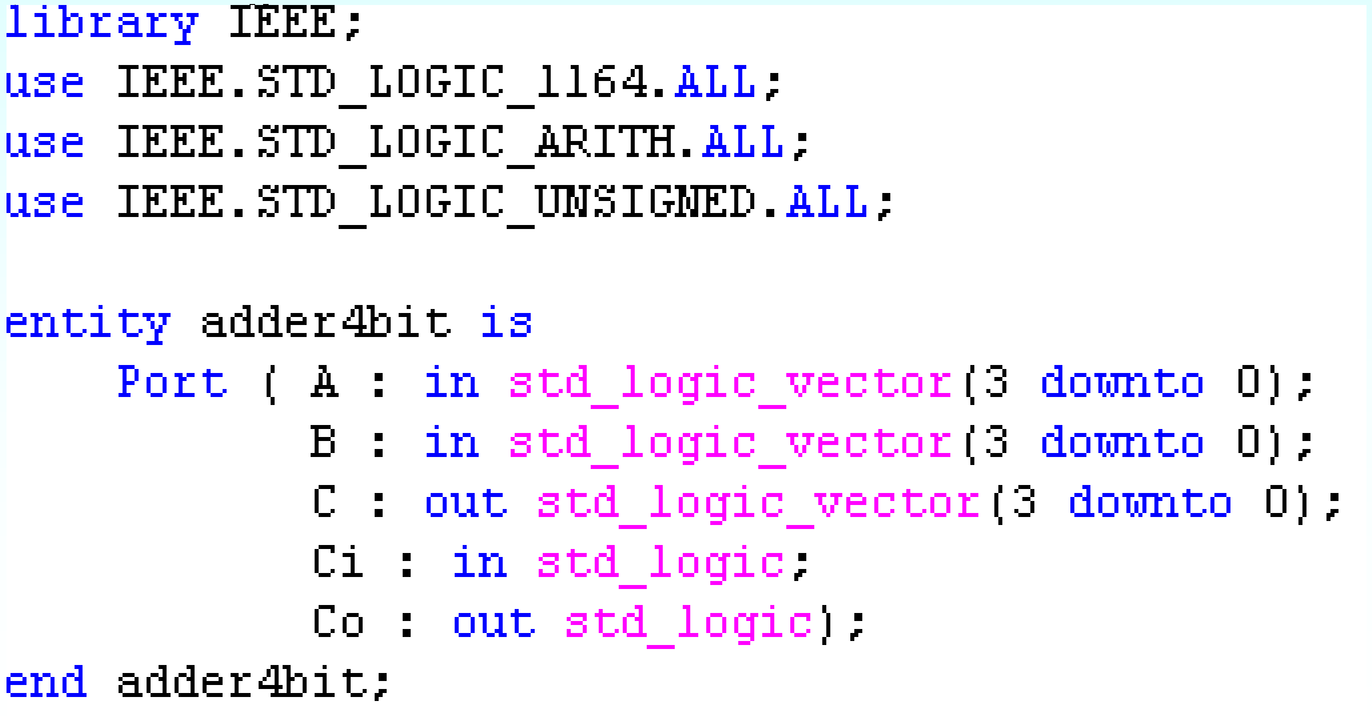
4-bit Ripple-Carry Adder - Structural Description cntd.



```
architecture Structure of Adder4 is
component FullAdder
   port (X, Y, Cin: in bit;          -- Inputs
         Cout, Sum: out bit);        -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin      --instantiate four copies of the FullAdder
   FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
   FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
   FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
   FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

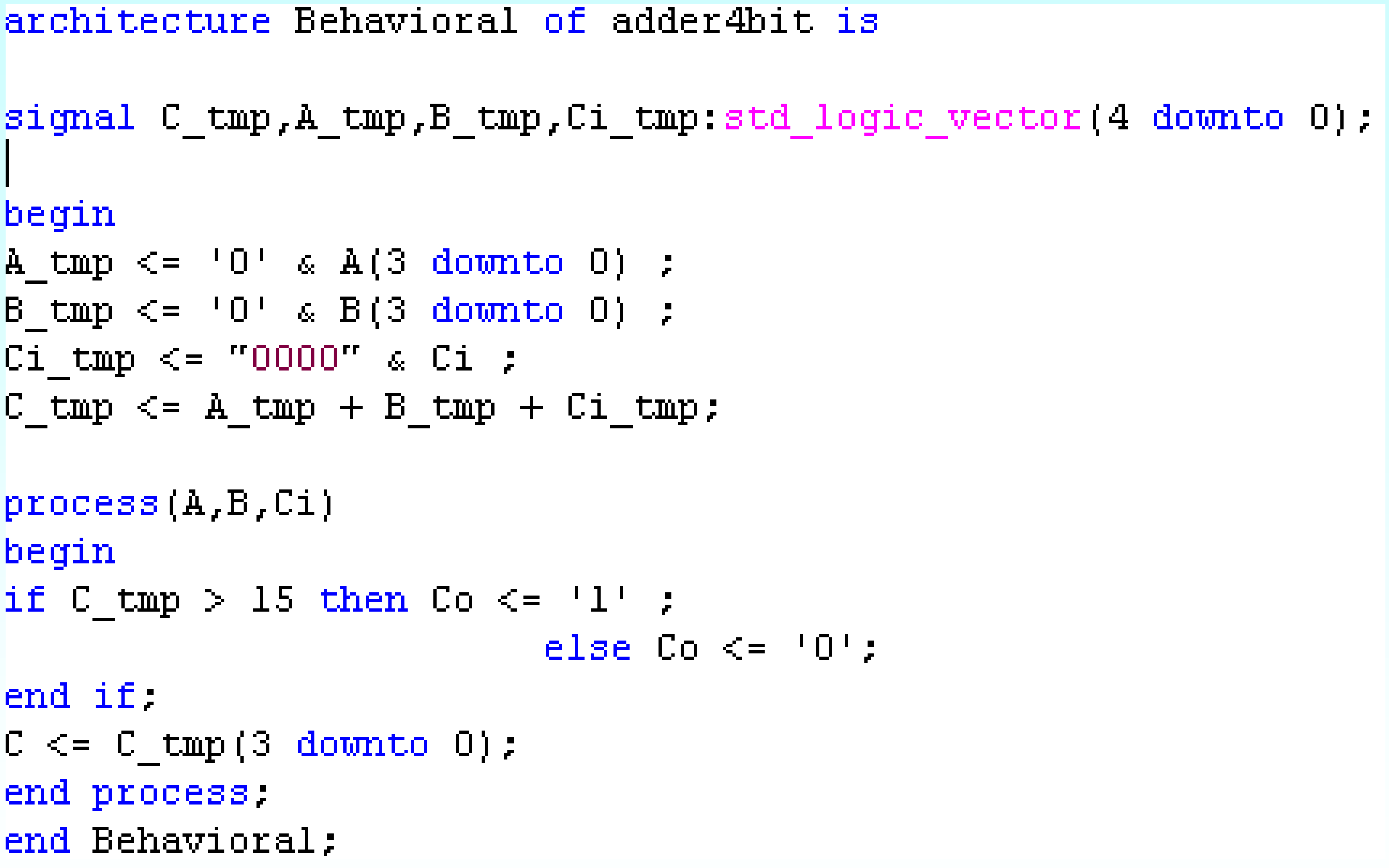
Behavioral Level

- Behavioral description
  - May not be synthesizable or may lead to a very large circuit
  - Primarily used for simulation



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder4bit is
    Port ( A : in std_logic_vector(3 downto 0);
           B : in std_logic_vector(3 downto 0);
           C : out std_logic_vector(3 downto 0);
           Ci : in std_logic;
           Co : out std_logic);
end adder4bit;
```



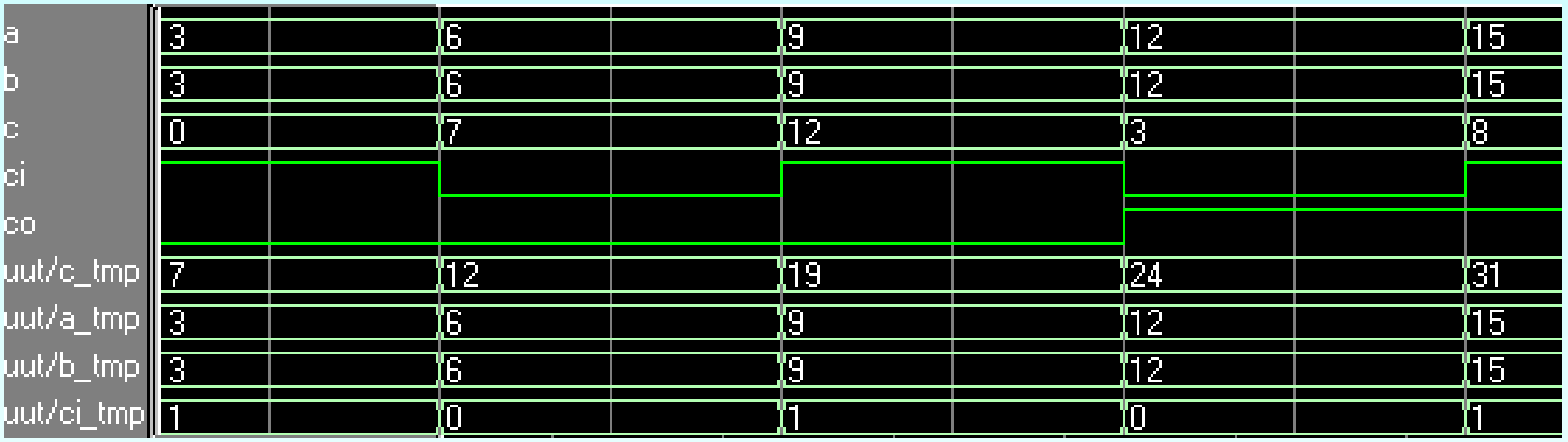
```vhdl
architecture Behavioral of adder4bit is

signal C_tmp,A_tmp,B_tmp,Ci_tmp:std_logic_vector(4 downto 0);

begin
A_tmp <= '0' & A(3 downto 0) ;
B_tmp <= '0' & B(3 downto 0) ;
Ci_tmp <= "0000" & Ci ;
C_tmp <= A_tmp + B_tmp + Ci_tmp;

process(A,B,Ci)
begin
if C_tmp > 15 then Co <= '1' ;
                        else Co <= '0';
end if;
C <= C_tmp(3 downto 0);
end process;
end Behavioral;
```

Simulation results (temp. signals also shown)



HDL Synthesis Report

Macro Statistics

# Adders/Subtractors	: 2
5-bit adder	: 2
# Comparators	: 1
5-bit comparator greater	: 1

A strong reason to think of hardware being designed, while writing VHDL behavioral code.

Constructs in VHDL

Concurrent Statements

- All concurrent statements in an architecture are executed simultaneously.
- Concurrent statements are used to express parallel activity as is the case with any digital circuit.
- Concurrent statements are executed with no predefined order by the simulator . So the order in which the code is written does not have any effect on its function.
- They can be used for behavioral and structural and data flow descriptions.

Concurrent statements contd.

- Process is a concurrent statement in which sequential statements are allowed.
- All processes in an architecture are executed simultaneously.
- Concurrent statements are executed by the simulator when one of the signals in its sensitivity list changes . This is called occurrence of an ‘event’.

  eg : c <= a or b;

  is executed when either signal ‘a’ or signal ‘b’ changes.

  process(clk , reset) ...

  is executed when either ‘clk’ or ‘reset’ changes
- Signals are concurrent whereas variables are sequential objects.

Conditional signal assignment

- The ‘when‘ statement
  - This type of assignment has one target but multiple condition expressions.
  - This statement assigns value based on the priority of the condition.
  - syntax

```
sig_name <= exp1 when condition1 else
            exp2 when condition2 else
            exp3;
```

```
entity my_nand is
port (a, b : in  std_logic;
      c    : out std_logic);
end my_nand;
architecture beh of my_nand is
begin
  c <= ‘0’ when a = ‘1’ and b = ‘1’ else
       ‘1’ ;
end beh;
```

```
entity tri_state is
port (a, en : in  std_logic;
      b     : out std_logic);
end tri_state;
architecture beh of tri_state is
begin
  b <= a when en = ‘1’ else
      ‘Z’;
end beh;
```

example

```
architecture try_A of try is
begin
   Y <= i1 when s1 = ‘0’ and s0 = ‘0’ else
        i2 when s1 = ‘0’ and s0 = ‘1’ else
        i3 when s1 = ‘1’ and s0 = ‘0’ else
        i4 when s1 = ‘1’ and s0 = ‘1’ else
        ‘0’ ;
end try_A;
```

Incomplete specification is not allowed

example

```
architecture when_grant of bus_grant is
  signal …
begin
   data_bus <= a and b when e1 = ‘1’
     else
                    e or f  when a = b else
                    g & h   when e3 = ‘1’
            else
                    (others => ‘Z’);
end when_grant;
```

Selective signal assignment

The **with** statement

- This statement is similar to the case statement
- syntax

```
with expression select
 target <= expression1 when choice1
           expression2 when choice2
           expressionN when choiceN;
```

- all possible choices must be enumerated
- **when others** choice takes care of all the remaining alternatives.

Difference between `with` and `when` statements

- Each choice in the with statement should be unique

- Compared to the ‘when’ statement, in the ‘with’ statement, choice is limited to the choices provided by the with ‘expression’, whereas for the ‘when’ statement each choice itself can be a separate expression.

- The when statement is prioritized (since each choice can be a different expression, more than one condition can be true at the same time, thus necessitating a priority based assignment) whereas the with statement does not have any priority (since choices are mutually exclusive)

```
entity my_mux is
   port (a, b, c, d : in  std_logic;
         sel0, sel1 : in  std_logic;
         e          : out std_logic);
end my_mux;

architecture my_mux_A of my_mux is
  signal sel: std_logic_vector(1 downto 0);
begin
  sel <= sel1 & sel0;
  with sel select
    e <= a when “00”
         b when “01”
         c when “10”
         d when others;
end my_mux_A;
```

Component Instantiation

- A component represents an entity architecture pair.
- Component allows hierarchical design of complex circuits.
- A component instantiation statement defines a part lower in the hierarchy of the design entity in which it appears. It associates ports of the component with the signals of the entity. It assigns values to the generics of the component.
- A component has to be declared in either a package or in the declaration part of the architecture prior to its instantiation.

Component Declaration and Instantiation

- Syntax(Declaration)

```
component component_name
   [generic list]
   [port list]
end component;
```

- Syntax(Instantiation)

```
label:component_name
[generic map]
port map;
```

```
entity my_and is
  port( a : in  std_logic;
        b : in  std_logic;
        c : out std_logic);
end my_and;

architecture my_and_A of my_and is
  component and2
    generic (tpd: time := 2 ns);
    port (x : in  std_logic;
          y : in  std_logic;
          z : out std_logic);
  end component;
  signal temp : std_logic;
begin
  c <= temp;
  -- component instantiation here
end my_and_A;
```

```
U1: my_and
  generic map (tpd => 5 ns)
  port map (x => a,
            y => b,
            z => temp);
```

```
U2: my_and
  generic map (tpd => 2 ns)
  port map (x => a,
            y => b,
            z => temp);
```

```
architecture exor_A of exor is
  component my_or
    port     (a  : in   std_logic;
              b  : in   std_logic;
              y  : out  std_logic
              );
  end component;
  component my_and
    port     (a  : in   std_logic;
              b  : in   std_logic;
              y  : out  std_logic
             );
  end component;
  signal a_n, b_n : std_logic;
  signal y1, y2, y3 : std_logic;
begin
 . . . . .
end exor_A;
```

```
u1 : my_or
    port map (y2,
              y3,
              y1);
u2 : my_and
    port map (a_n,
              b,
              y2);
u3 : my_and
    port map (a,
              b_n,
              y3);

a_n <= not a ;
b_n <= not b ;
```

Component Instantiation contd.

- Positional association

```
U1: my_and
generic map(5 ns)
port map(a, b, temp);
```

- Named Association

```
U1:my_and
generic map (tpd => 5 ns)
port map (x => a,
          y => b,
          z => temp);
```

The formal and the actual can have the same name

Component Instantiation contd.

- Named association is preferred because it makes the code more readable and pins can be specified in any order whereas in positional association order should be maintained as defined in the component and all the pins need to be connected .

- Multiple instantiation of the same component should have different labels.

Process statement

- The process statement is a concurrent statement , which delineates a part of an architecture where sequential statements are executed.
- Syntax

```
label: process [(sensitivity list )]
  declarations
begin
  sequential statements
end process;
```

Process statement

- All processes in an architecture are executed concurrently with all other concurrent statements.
- Process is synchronized with the other concurrent statements using the sensitivity list or a wait statement.
- Process should either have sensitivity list or an explicit wait statement. Both should not be present in the same process statement.
- The order of execution of statements is the order in which the statements appear in the process
- All the statements in the process are executed continuously in a loop .

Process contd.

- The simulator runs a process when any one of the signals in the sensitivity list changes. For a wait statement, the simulator executes the process after the wait is over.
- The simulator takes 0 simulation time to execute all the statements in the process. (provided there is no wait)

```
process
begin
     if (reset = ‘1’) then
         A <= ‘0’ ;
     elsif (clk’event and clk = ‘1’) then
         A <= ‘B’;
      end if;
      wait on reset, clk;
end process;
```

```
process (clk,reset)
begin
    if (reset = ‘1’) then
         A <= ‘0’;
    elsif (clk’event and clk = ‘1’) then
         A <= ‘B’;
    end if;
end process;
```

Sequential Statements

- Sequential statements are statements which are analyzed serially one after the other. The final output depends on the order of the statements, unlike concurrent statements where the order is inconsequential.

- Sequential statements are allowed only inside process and subprograms (function and procedure)
- Process and subprograms can have only sequential statements within them.
- Only sequential statements can use variables.
- The Process statement is the primary concurrent VHDL statement used to describe sequential behaviour.

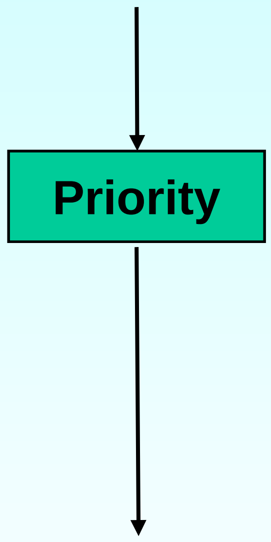
Sequential Statements contd.

- Sequential statements can be used to generate
  - Combinational logic
  - Sequential logic
- Clocked process
  - It is easily possible to infer flip-flops using if statements and ‘event attribute.
- Combinatorial process
  - generates purely combinatorial logic.
  - All the inputs must be present in the sensitivity list. Otherwise the simulation and synthesis results will not match.

The if statement

- Syntax

```
if condition1 then
  statements
[elsif condition2 then
   statements]
[else
   statements]
end if;
```



- An if statement selects one or none of a sequence of events to execute . The choice depends on one or more conditions.

The if statement contd.

```
if sel = ‘1’ then
  c <= a;
else
  c <= b;
end if;
```

```
if (sel = “00”) then
  o <= a;
elsif sel = “01” then
  x <= b;
elsif (color = red) then
  y <= c;
else
  o <= d;
end if;
```

- If statements can be nested.
- If statement generates a priority structure
- If corresponds to when else concurrent statement.

The case statement - syntax

```
case expression is
  when choice 1 =>
    statements
  when choice 3 to 5 =>
    statements
  when choice 8 downto 6 =>
    statements
  when choice 9 | 13 | 17 =>
    statements
  when others =>
    statements
end case;
```

The case statement

- The case statement selects, for execution one of a number of alternative sequences of statements .
- Corresponds to with select in concurrent statements .
- Case statement does not result in prioritized logic structure unlike the if statement.

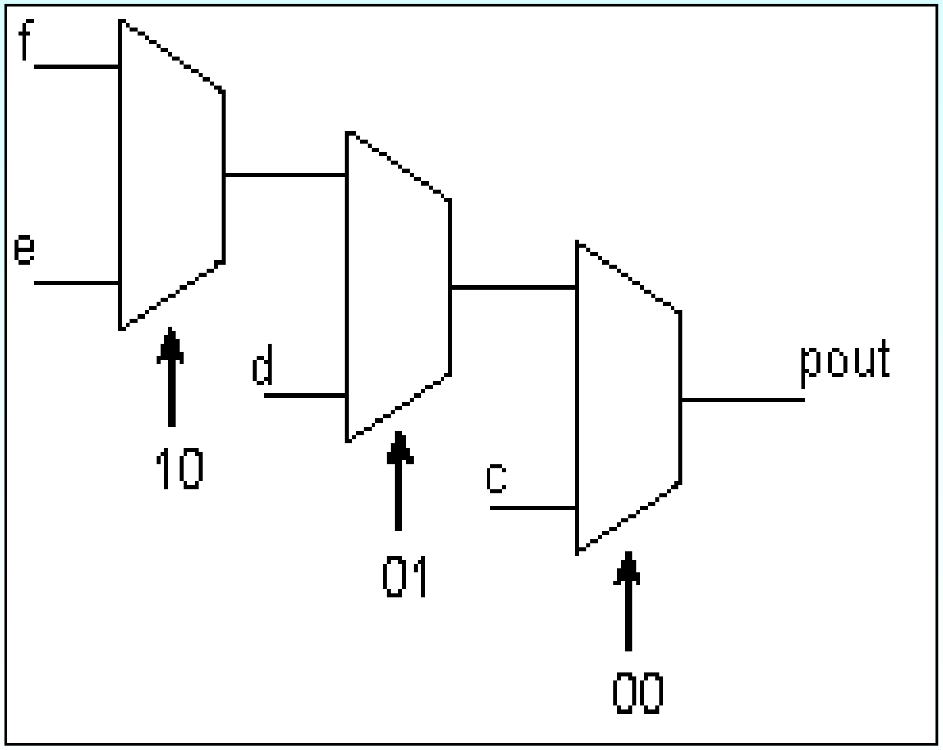
The case statement contd.

```
process (count)
begin
  case count is
    when 0 =>
      dout <= “00”;
    when 1 to 15 =>
      dout <= “01”;
    when 16 to 255 =>
      dout <= “10”;
    when others =>
      null;
  end case;
end process;
```

```
process(sel, a, b, c, d)
begin
  case sel is
    when “00” =>
      dout <= a;
    when “01” =>
      dout <= b;
    when “10” =>
      dout <= c;
    when “11” =>
      dout <= d;
    when others =>
      null;
  end case;
end process;
```

Think Hardware! (Mutually exclusive conditions)

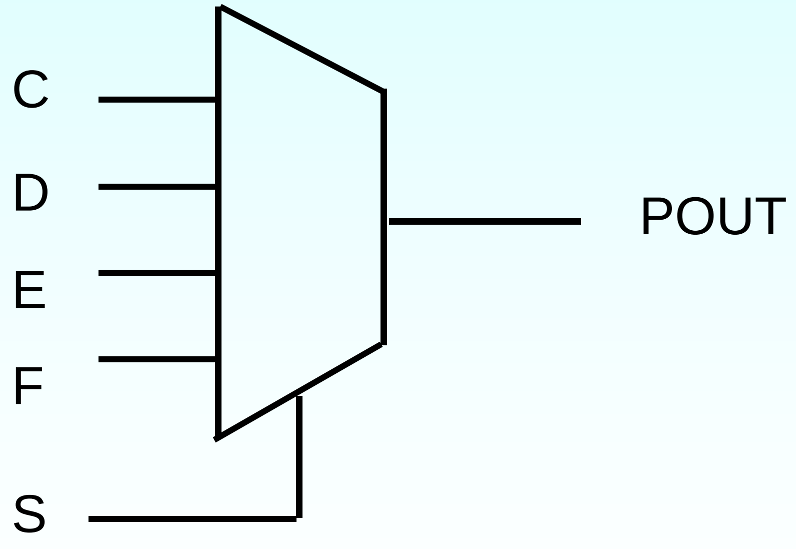
```
myif_pro: process (s, c, d, e, f)
begin
  if s = "00" then
    pout <= c;
  elsif s = "01" then
    pout <= d;
  elsif s = "10" then
    pout <= e;
  else
    pout <= f;
  end if;
end process myif_pro;
```



This priority is useful for timings.

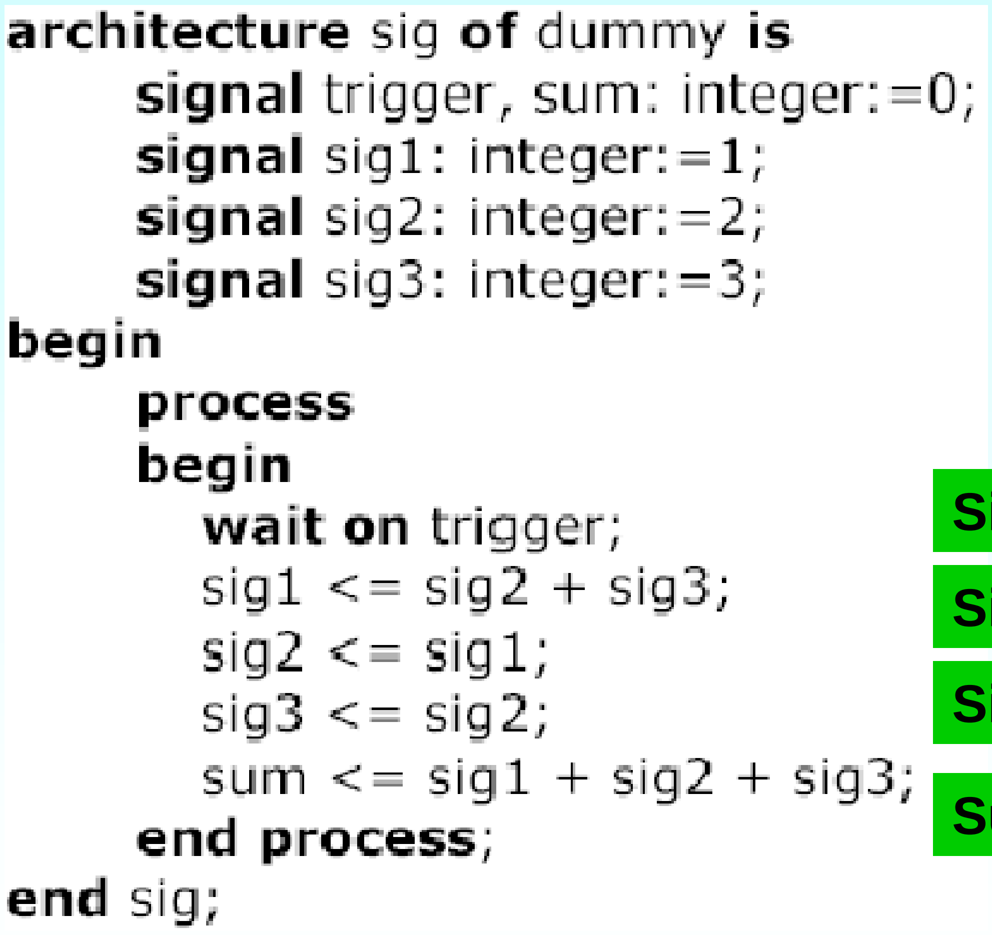
Think Hardware! Use a case for mutually exclusive things

```
mycase_pro: process (s, c, d, e, f)
  begin
    case s is
    when "00" =>
      pout <= c;
    when "01" =>
      pout <= d;
    when "10" =>
      pout <= e;
    when others =>
      pout <= f;
    end if;
  end process mycase_pro;
```



There is no priority with case.

BEHAVIORAL (Processes using signals)



```
architecture sig of dummy is
        signal trigger, sum: integer:=0;
        signal sig1: integer:=1;
        signal sig2: integer:=2;
        signal sig3: integer:=3;
begin
        process
        begin
            wait on trigger;
            sig1 <= sig2 + sig3;
            sig2 <= sig1;
            sig3 <= sig2;
            sum <= sig1 + sig2 + sig3;
        end process;
end sig;
```

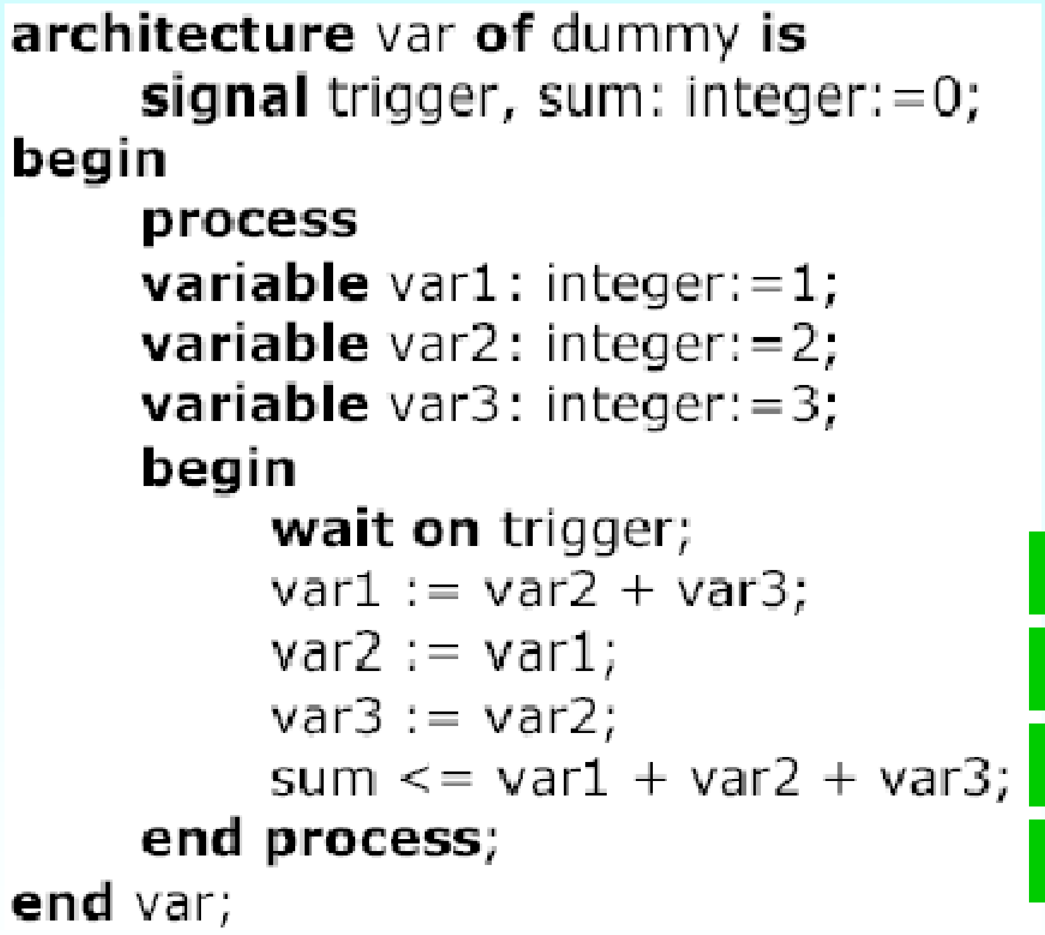
Sig1 = 2 + 3 = 5

Sig2 = 1

Sig3 = 2

Sum = 1 + 2 + 3 = 6

BEHAVIORAL (Processes using Variables)



```
architecture var of dummy is
        signal trigger, sum: integer:=0;
begin
        process
        variable var1: integer:=1;
        variable var2: integer:=2;
        variable var3: integer:=3;
        begin
                wait on trigger;
                var1 := var2 + var3;
                var2 := var1;
                var3 := var2;
                sum <= var1 + var2 + var3;
        end process;
end var;
```

var1 = 2 + 3 = 5

var2 = 5

var3 = 5

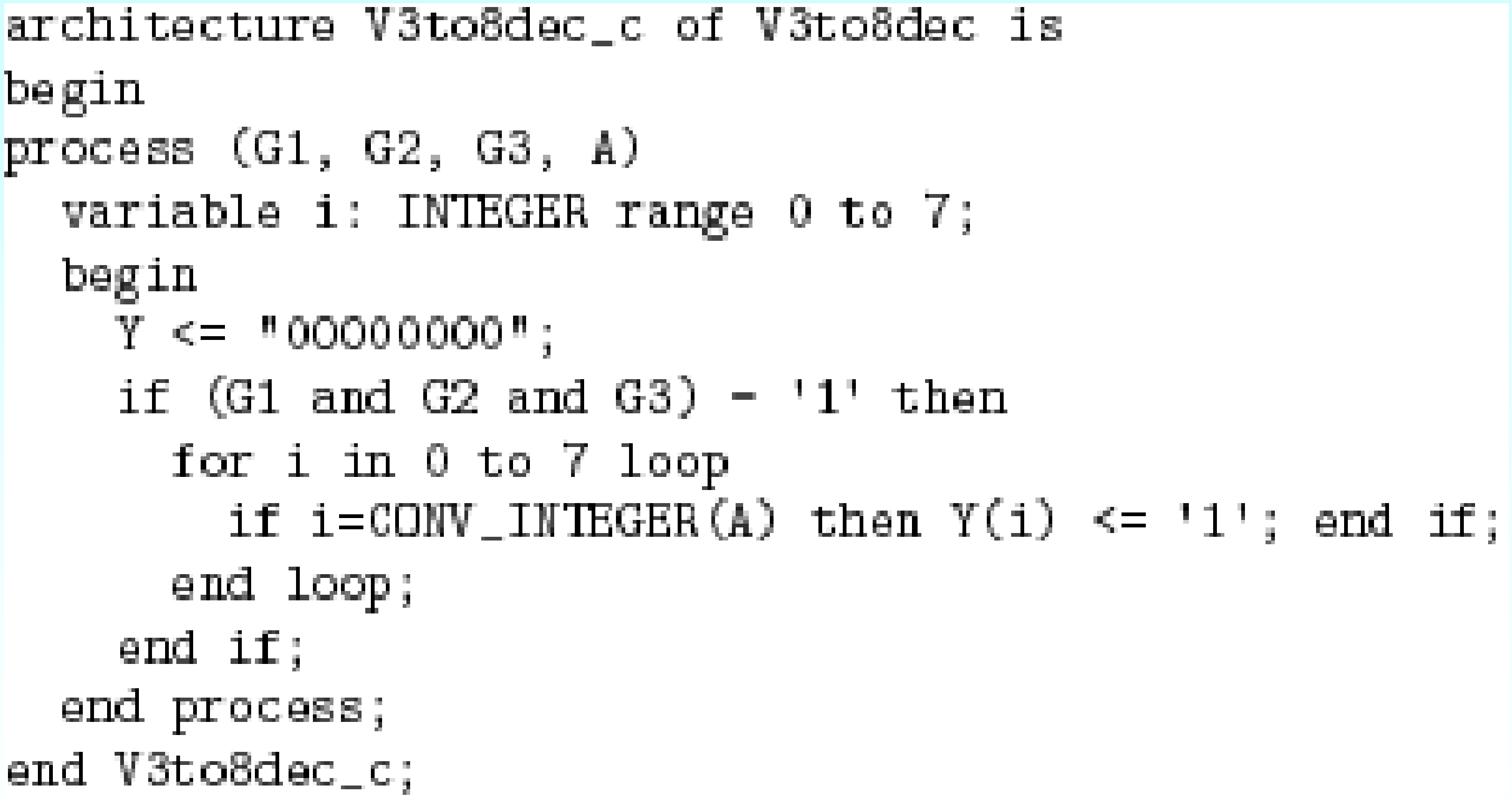
Sum = 5 + 5 + 5 = 15

Behavioral Description of a 3-to-8 Decoder

Except for different syntax, approach is not all that different from the dataflow version

```
architecture V3to8dec_b of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
process(A, G1, G2, G3, Y_s)
  begin
    case A is
      when "000" => Y_s <= "10000000";
      when "001" => Y_s <= "01000000";
      when "010" => Y_s <= "00100000";
      when "011" => Y_s <= "00010000";
      when "100" => Y_s <= "00001000";
      when "101" => Y_s <= "00000100";
      when "110" => Y_s <= "00000010";
      when "111" => Y_s <= "00000001";
      when others => Y_s <= "00000000";
    end case;
    if (G1 and G2 and G3)='1' then Y <= Y_s;
    else Y <= "00000000";
    end if;
  end process;
end V3to8dec_b;
```

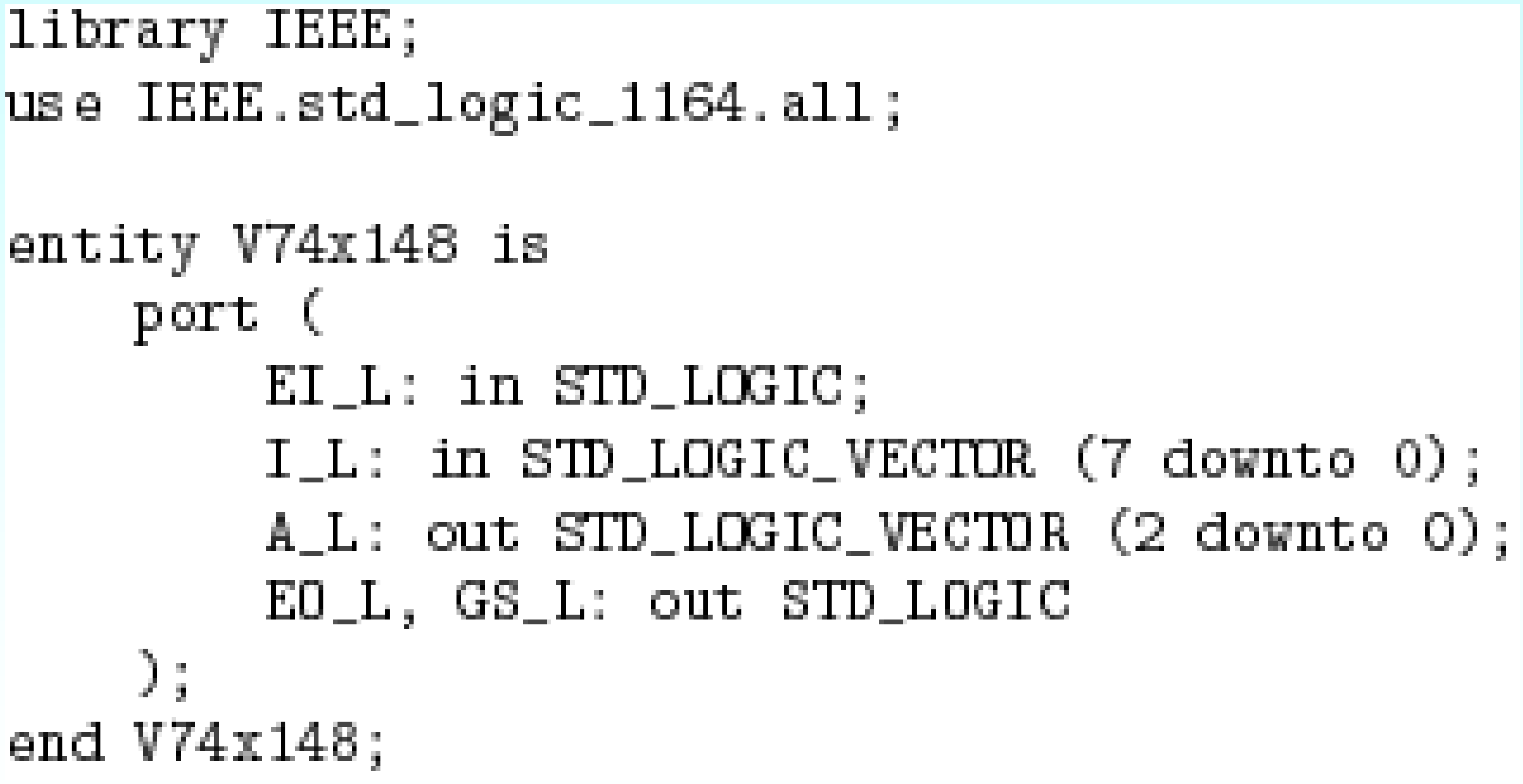
A Different Behavioral Description of a 3-to-8 Decoder



```
architecture V3to8dec_c of V3to8dec is
begin
process (G1, G2, G3, A)
  variable i: INTEGER range 0 to 7;
  begin
    Y <= "00000000";
    if (G1 and G2 and G3) - '1' then
      for i in 0 to 7 loop
        if i=CONV_INTEGER(A) then Y(i) <= '1'; end if;
      end loop;
    end if;
  end process;
end V3to8dec_c;
```

May not be synthesizable,
or may have a slow or inefficient realization.
But just fine for simulation and verification.

74x148 behavioral description (8 to 3 line cascadable Priority Encoder)



```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x148 is
    port (
        EI_L: in STD_LOGIC;
        I_L: in STD_LOGIC_VECTOR (7 downto 0);
        A_L: out STD_LOGIC_VECTOR (2 downto 0);
        EO_L, GS_L: out STD_LOGIC
    );
end V74x148;
```

```
architecture V74x148p of V74x148 is
  signal EI: STD_LOGIC;                          -- active-high version of input
  signal I: STD_LOGIC_VECTOR (7 downto 0);  -- active-high version of inputs
  signal EO, GS: STD_LOGIC;                      -- active-high version of outputs
  signal A: STD_LOGIC_VECTOR (2 downto 0);  -- active-high version of outputs
begin
  process (EI_L, I_L, EI, EO, GS, I, A)
  variable j: INTEGER range 7 downto 0;
  begin
    EI <= not EI_L; -- convert input
    I <= not I_L;   -- convert inputs
    EO <= '1'; GS <= '0'; A <= "000";
    if (EI)='0' then EO <= '0';
    else for j in 7 downto 0 loop
        if I(j)='1' then
          GS <= '1'; EO <= '0'; A <= CONV_STD_LOGIC_VECTOR(j,3);
          exit;
        end if;
      end loop;
    end if;
    EO_L <= not EO; -- convert output
    GS_L <= not GS; -- convert output
    A_L <= not A;   -- convert outputs
  end process;
end V74x148p;
```

--EI - Enable I/P
--EO - O/P Enable
--I - I/P(data to be encoded)
--A - O/P

type conversion

CONCLUSION

- Many VHDL constructs, although useful for simulation and other stages in the design process, are not relevant to synthesis. A sub-set of VHDL only can be used for synthesis.
- A construct may be fully supported, ignored, or unsupported.
- Ignored means that the construct will be allowed in the VHDL file but will be ignored by the synthesis tool.
- Unsupported means that the construct is not allowed and the code will not be accepted for synthesis.
- See the documentation of tools for exact details.