UTS VLSI forum

VHDL Coding basics – Combinational logic circuits

By

K. Vasu

Senior Application engineer

vasu@unistring.com, vasu.vlsi@gmail.com

Unistring Tech Solutions Pvt. Ltd

D10, 5th Floor, Eureka court, Beside Image hospitals, Ameerpet, Hyderabad, ph:040-23732798, 09440318188

www.unistring.com

String Technologies

#309, Vederi complex, opp Dilshuk Nagar bus Depot, Dilsukh nagar, Hyderabad, ph: 040 – 24151900

www.stringtechnologies.net

History of VHDL

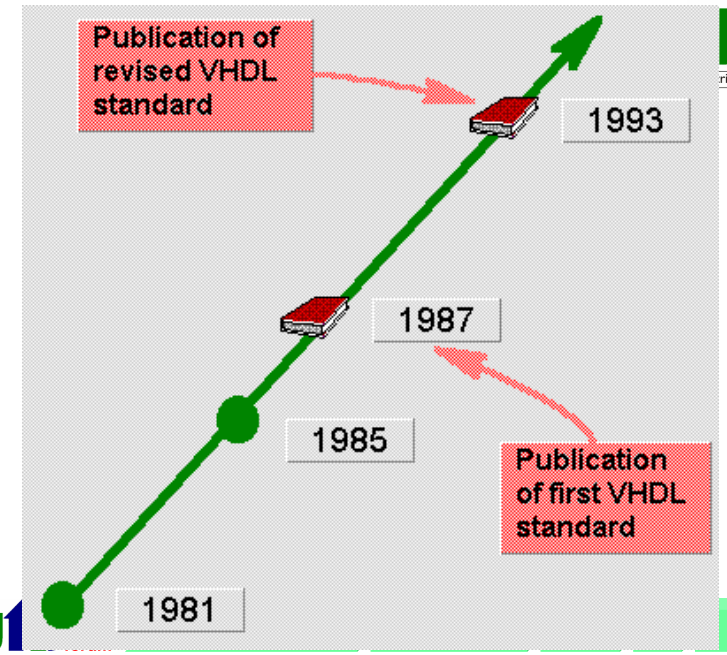
1981 - Initiated by US DoD to address hardware life-cycle crisis

1983-85 - Development of baseline language by Intermetrics, IBM and TI

1986 - All rights transferred to IEEE

1987 - Publication of IEEE Standard

1987 - Mil Std 454 requires comprehensive VHDL descriptions to be delivered with ASICs



VHDL

- **VHDL is a Hardware Description Language**
- **VHDL stands for VHSIC Hardware Description Language.**

  (VHSIC stands for Very High Speed Integrated Circuits)
- **Initiated by by United States DOD in the 1980s**
- **The first version was VHDL 87 ; Upgraded to VHDL 93**
- **Standardized by the (IEEE) Institute of Electrical and Electronic Engineers -- IEEE 1076**
- **An additional standard IEEE1164, was later added to introduced multi valued logic system.**
- **VHDL is intended for circuit simulation and as well as circuit synthesis**
- **VHDL is a standard Technology/vendor independent language, and is therefore portable and reusable.**

Design Abstraction Levels

Transistor level - dealing with discrete transistors and connecting them together to form the circuit

Gate level - working with logic gates to build the circuit

register-transfer level - concerned about how the data is transferred between the various registers and functional units to solve the problem at hand

Behavioral level - describing the behavior or operation of the circuit using a hardware description language.

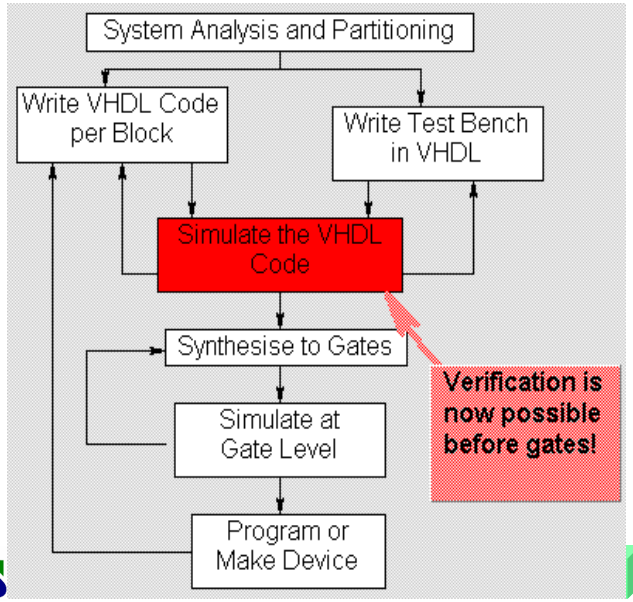
RTL is an acronym for Register Transfer Level

In RTL coding the logic is realized by transferring the data between various registers , with appropriate combinational logic.

This is suitable coding style for several synthesis tools.

70% of design time at RTL in most of the digital designs

Design Flow using VHDL



Simulation

Simulation is a process in which the logic and functional verification of algorithm/design will be verified, by subjecting it to appropriate input test stimulus.

Synthesis

In synthesis the VHDL/Verilog code is translated into low level netlist, to program a device or make a chip.

VLSI Design entry (part of front end design)

In semi custom VLSI design style the required logic can be described by any one, or combination of the following techniques

- Schematic Entry
- Hardware Description Languages
  - VHDL
  - Verilog
  - AHDL
- State machine models
- High level programming languages (latest trend)
  - systemC (C++ class library with Verilog syntax)
- Few Other techniques

VLSI Synthesis (or) Back end Design

In semi custom VLSI design style the required logic (which is in suitable form) can be synthesized for any one of the following devices.

- FPGAs
- CPLDs

} PLD synthesis

- using ASIC cell libraries

} ASIC synthesis

- Gate Arrays

} Structured ASIC synthesis

VHDL code structure

A simple VHDL code consists of at least three fundamental sections

Library declarations

contains a list of all libraries and packages to be used in the design

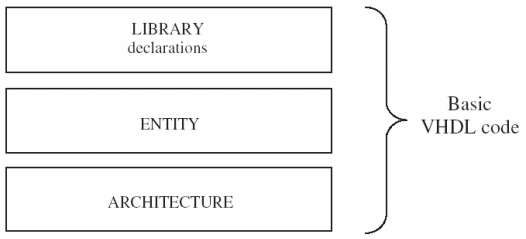
For eg ieee, work, std etc

Entity

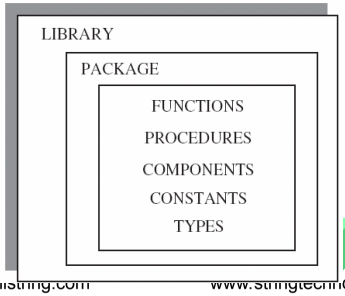
Entity specifies the I/O ports of the circuit

Architecture

Contains the VHDL code, which describes how the circuit should behave




```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```



```
LIBRARY ieee;                   -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all;    -- the end of a statement or

LIBRARY std;                    -- declaration, while a double
USE std.standard.all;           -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;
```

The libraries *std* and *work* shown above are made visible by default, so there is no need to declare them; only the *ieee* library must be explicitly written. However, the latter is only necessary when the STD_LOGIC (or STD_ULOGIC) data type is employed in the design

Three styles of modeling

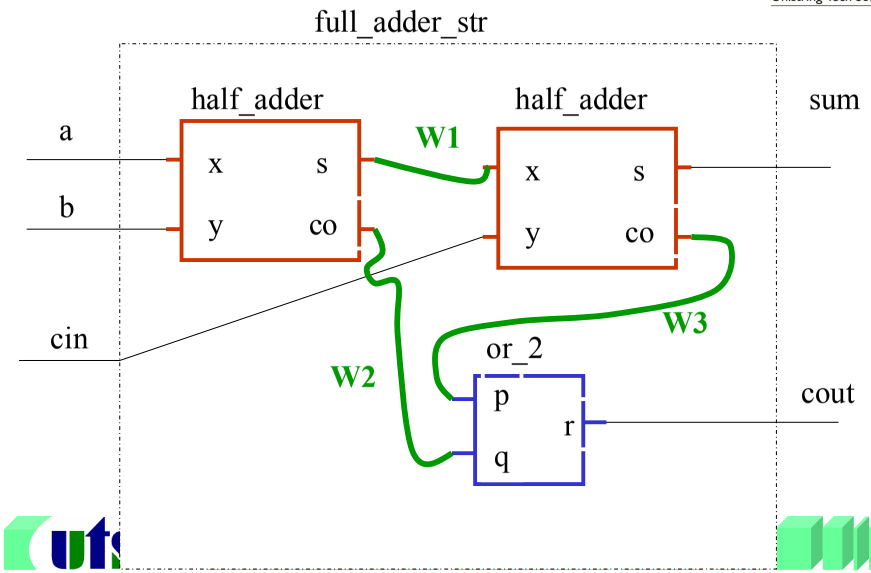
- Data flow style of modeling
- Structural style of modeling
- Behavioral style of modeling


```
-- This is dataflow style of full adder VHDL code
entity full_adder_dat is
       port(a,b,cin:in bit;
            sum,cout:out bit);
end full_adder_dat;

architecture arc_full_adder_dat of full_adder_dat is
begin
      sum <= a xor b xor cin;
      cout<= (a and b) or (b and cin) or (a and cin);
end arc_full_adder_dat;
```

concurrent statements

Structure modeling style of Full Adder



```
-- This is structural implementaion of full adder
entity full_adder_str is
      port(a,b,cin:in bit;
           sum,cout:out bit);
end full_adder_str;
architecture arc_full_adder_str of full_adder_str is
      component half_adder is
           port(x,y:in bit;
                s,co:out bit);
      end component;
      component or_2 is
           port(p,q:in bit;
                r:out bit);
      end component;
      signal w1,w2,w3:bit;
begin
      HA1: half_adder port map(a,b,w1,w2);
      HA2: half_adder port map(w1,cin,sum,w3);
      gate1: or_2 port map(w3,w2,cout);
end arc_full_adder_str;
```



```
-- This is half adder component which is used in
--                                  full_adder_str.vhd

entity half_adder is
      port(x,y: in bit;
           s,co:out bit);
end half_adder;

architecture arc_half_adder of half_adder is
begin
      s <= x xor y;
      co <= x and y;
end arc_half_adder;
```



```
-- This is OR gate component which is used in
--                                  full_adder_str.vhd
entity or_2 is
      port(p,q:in bit;
           r:out bit);
end or_2;

architecture arc_or_2 of or_2 is
begin
      r<= p or q;
end arc_or_2;
```



```
-- This is behavioural style code for full adder

entity full_adder_beh is
      port(a,b,cin:in bit;
           sum,cout:out bit);
end full_adder_beh;

architecture arc_full_adder_beh of full_adder_beh is
begin
      process(a,b,cin)
      begin
           sum<= a xor b xor cin;
           cout<= (a and b) or (b and cin) or (a and cin);
      end process;
end arc_full_adder_beh;
```

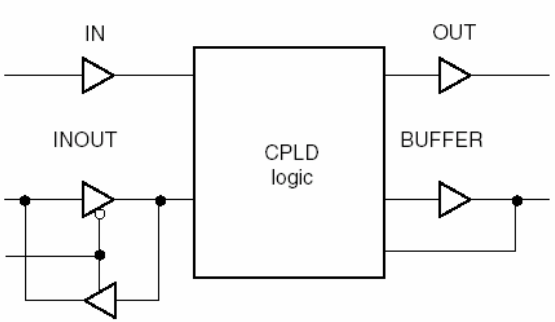
Sequentially executed statements

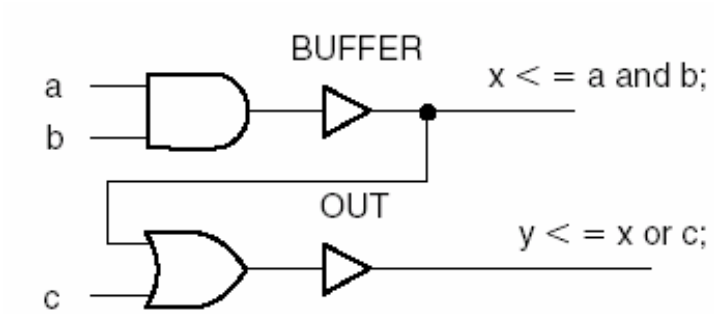
Sensitivity list

Process Statement

- The process statement consists of a number of parts.
- The first part is called the sensitivity list; the second part is called the process declarative part; and the third is the statement part.
- In the preceding example, the list of signals in parentheses after the keyword **PROCESS** is called the sensitivity list.
- This list enumerates exactly which signals cause the process statement to be executed.

Port Modes in VHDL





- The above figure shows the difference between BUFFER and OUT modes.
- Port x (defined by `x <= a and b;`) must be of mode BUFFER because it is fed back and used as part of the expression for port y (defined by `y <= x or c;`).
- Port y can be of mode OUT since it has no feedback, only an output.

VHDL coding for combinational circuits

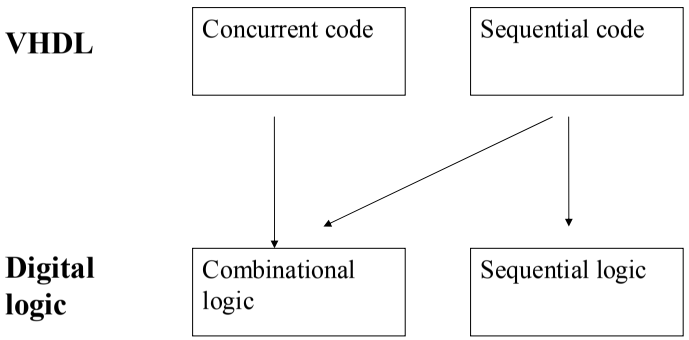
Combinational and Sequential circuits

- Combinational circuits
  - Present outputs depend only on the present set of inputs.
  - Eg: multiplexers, full adder, decoders
- Sequential circuits
  - Present outputs depend on present sep of inputs and previous inputs/outputs
  - Eg: Flip flops, counters, state machines

Concurrent code and Sequential code in VHDL

- Concurrent code
  - All statements of code are simulated simultaneously
  - Useful to describe the hardware using structural and data flow styles
  - Fundamentally useful for describing the combinational logic circuits
- Sequential code
  - Code statements are simulated sequentially one after another
  - Main purpose is to allow be behavioral modeling of digital circuits
  - Can be used to describe combinational and sequential circuits

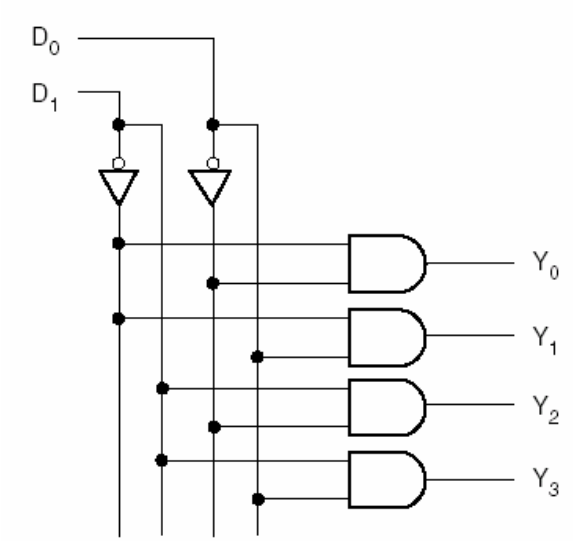
VHDL modeling styles & Digital logic types



VHDL constructs for concurrent coding

- Operators;
- The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
- The GENERATE statement;

Example 2-line-to-4-line decoder.

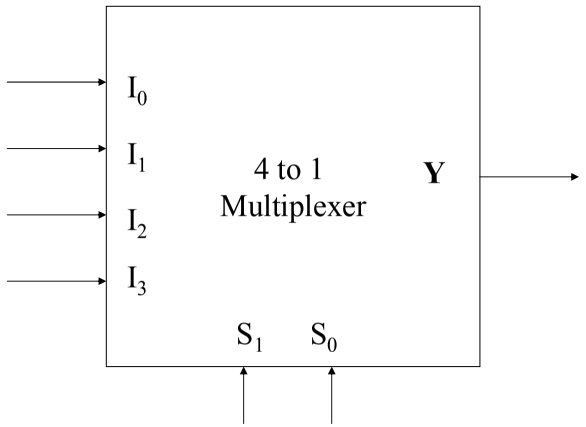



```
-- 2-line-to-4-line decoder implemented
--                 using data flow style
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY decode1 IS
      PORT( d1, d0 : IN STD_LOGIC;
            y0, y1, y2, y3 : OUT STD_LOGIC);
END decode1;
ARCHITECTURE decoder1 OF decode1 IS
BEGIN
      y0 <= (not d1) and (not d0);
      y1 <= (not d1) and ( d0);
      y2 <= ( d1) and (not d0);
      y3 <= ( d1) and ( d0);
END decoder1;
```



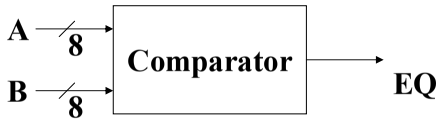
```
-- 2-line-to-4-line decoder implemented
--                 using data flow style
--    Implemented with std_logic_vectors
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY decode2 IS
      PORT (
      d : IN STD_LOGIC_VECTOR (1 downto 0);
      y : OUT STD_LOGIC_VECTOR (3 downto 0));
END decode2;
ARCHITECTURE decoder2 OF decode2 IS
BEGIN
      y(0) <= (not d(1)) and (not d(0));
      y(1) <= (not d(1)) and ( d(0));
      y(2) <= ( d(1)) and (not d(0));
      y(3) <= ( d(1)) and ( d(0));
END decoder2;
```


Example 4 to 1 Multiplexer



```
-- 4 to 1 multiplexer implemented
--                  using data flow style
entity mux4to1 is
      port(I0,I1,I2,I3:in bit;
           s1,s0:in bit;
           F:out bit);
end mux4to1;
architecture arc_mux of mux4to1 is
begin
      f<=(not s1 and not s0 and I0)
      or(not s1 and s0 and I1)
       or(s1 and not s0 and I2)
       or(s1 and s0 and I3);
end arc_mux;
```


Example 8-bit comparator




```
-- Simple 8 bit comparator using When-else statment
entity compare is
    port( A, B: in bit_vector(0 to 7);
          EQ: out bit);
end compare;

architecture compare1 of compare is
begin
    EQ <= '1' when (A = B) else '0';
end compare1
```


WHEN is one of the fundamental concurrent statements (along with operators and GENERATE). It appears in two forms: WHEN / ELSE (*simple* WHEN) and WITH / SELECT / WHEN (*selected* WHEN). Its syntax is shown below.

WHEN / ELSE:

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH / SELECT / WHEN:

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

Whenever WITH / SELECT / WHEN is used, all permutations must be tested, so the keyword OTHERS is often useful. Another important keyword is UNAFFECTED, which should be used when no action is to take place.

4 to 1 Multiplexer using *when - else*

```
------- Solution 1: with WHEN/ELSE --------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE mux1 OF mux IS
BEGIN
   y <=  a WHEN sel="00" ELSE
         b WHEN sel="01" ELSE
         c WHEN sel="10" ELSE
         d;
END mux1;
```

4 to 1 Mux using *with select when*

```
--- Solution 2: with WITH/SELECT/WHEN -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
          sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE mux2 OF mux IS
BEGIN
   WITH sel SELECT
      y <=  a WHEN "00",      -- notice "," instead of ";"
            b WHEN "01",
            c WHEN "10",
            d WHEN OTHERS;    -- cannot be "d WHEN "11" "
END mux2;
-----------------------------------------
```


Generate statements

- For – generate
  - Used for repeating some component instances and concurrent statements
  - Results in multiple instants of hardware when synthesized
  - Must be labled
- If – generate
  - Used for condition based digital logic realization
  - When synthesized results in enable and other control signals

generic statement

- VHDL allows scaling a component with *generic*
- The vector sizes and number of component instances we can pass from higher level module
  - Which increases the reusability of the code
  - Makes the code scalable for required sizes

n bit adder with generic and for-generate

```
-- n bit adder with for generate statment
entity adder_nbit is
   generic(n:integer:=6);
   port(a,b:in bit_vector(n-1 downto 0);
      z:out bit_vector(n downto 0));
end adder_nbit;

architecture arc_nbit_adder of adder_nbit is
   signal c:bit_vector(n downto 0);
begin
   c(0) <='0';
ANBIT: for i in a'low to a'high generate
      z(i) <= a(i) xor b(i) xor c(i);
      c(i+1) <= (a(i) and b(i)) or (a(i) and c(i)) or (b(i) and c(i));
   end generate;
   z(z'high) <= c(c'high);
end arc_nbit_adder;
```

Unistring Tech Solutions

Thank you