Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

VHDL
What will be taking about?

• How to code up Digital logic and realize it in hardware...

• What do we mean by Digital Logic??
• How’s it different from Analog Logic??
Origins

• VHDL was developed as a language for **modelling and simulation**.
  – To create coherence between projects that US DoD offloaded to external vendors.

• Primary goal was simulation...
• Later-on *synthesis* (inferring hardware from the code) also became an application.

• Important to note that there is mismatch between synthesis and simulation...
  – Most constructs are good for simulation but not synthesizable.
  – Synthesizable subset of VHDL is relatively small % of all constructs.
Other HDLs

• **Verilog**
  – Syntax like C... quite common in US markets.

• **SystemC**
  – C++ based library. Quite useful for rapid prototyping.
  – Evolve simulation/abstract system description into detailed hardware as time progresses.

• **System Verilog**
  – Evolved version of Verilog with even advanced Verification constructs.

• **Matlab Simulink**
  – Specially useful for DSP applications.

• **Why VHDL?**
  – It’s like the assembly language of HDLs.
  – Simple
  – Extremely typed – very difficult (not impossible though!) to make mistakes.
Some ‘Zen’ teaching stuff...

• Keep in mind it’s HDL...
  – Used to ‘DESCRIBE’ Hardware...
  – That means one should know what Hardware is to be described...
  – It’s not just coding the flow as in CS... we’ll see what difference does it make...

• To area, speed, cost, time of development...
What does hardware look like...

• Lots of chips interconnected together with wires (either external or on PCB)...

• Some logic inside these chips executing as per specifications...

• Some interfaces to interact with external world...

• Lets say we want to describe this board...
  – How should we start ...


32 external pins
IC 1
IC 2
IC 3
IC 4
IC 5
Structure of a VHDL program

• Libraries
  – For compiler to interpret base functions.

• Entity
  – Information regarding the interface of the module/chip.
  – Eg: 1 Vcc pin, 1 Gnd pin, 8 inputs, 4 outputs.

• Architecture
  – Functionality of the module/chip.
Some Data types and Libraries

Data Types

• **Integer**
• **Bit**
  - For defining an ideal wire.
  - 2 binary values
• **Bit_vector**
  - For defining a bus... lot of wires together.
• **Std_logic**
  - For defining an actual wire.
  - 9 logic values:
    • 0,1
    • X – Unknown, multiple signals driving the same wire – kind of short circuit.
    • U - Uninitialized
    • Z – High impedance
• **Std_logic_vector**

Libraries

• **Std_logic_1164**
  - The std_logic data types and a few functions.
• **std_logic_arith**
  - some types and basic arithmetic operations for representing integers in standard ways.
• **std_logic_unsigned**
  - extends the std_logic_arith library to handle std_logic_vector values as unsigned integers.
• **std_logic_signed**
  - extends the std_logic_arith library to handle std_logic_vector values as signed integers.
• **std_logic_textio**
  - File handling operations for simulation.
• Library IEEE;
• Use IEEE.std_logic_1164.all;

• Entity IC_7402 is
  Port (  
p1 : out std_logic;
p2: in std_logic;
p3 : in std_logic;
... and so on ...
 );
End IC_7402;

OR

• Library IEEE;
• Use IEEE.std_logic_1164.all;

• Entity IC_7402 is
  Port (  
outp : out std_logic_vector(3 downto 0);
inp_a : in std_logic_vector(3 downto 0);
inp_b : in std_logic_vector(3 downto 0);
 );
End IC_7402;
Suppose we want to declare a ‘N’ input IC...

• Library IEEE;
• Use IEEE.std_logic_1164.all;

• Entity IC_7402 is
  Generic (N : integer range 7 downto 0 := 4);
  Port (outp : out std_logic_vector(N-1 downto 0);
         inp_a : in std_logic_vector(N-1 downto 0);
         inp_b : in std_logic_vector(N-1 downto 0);
       );
End IC_7402;

• Generics
  – Used to pass certain properties into a design to make it more general.
    • Bus widths.
    • Delays.
Entity Declaration

- An entity declaration describes the interface of the component.
- PORT clause indicates input and output ports.
- An entity can be thought of as a symbol for a component.

```plaintext
ENTITY half_adder IS
    PORT( x, y, enable: IN bit;
          carry, result: OUT bit);
END half_adder;
```
Port Declaration

• PORT declaration establishes the interface of the object to the outside world.

• Three parts of the PORT declaration
  • Name
    • Any identifier that is not a reserved word.
  • Mode
    • In, Out, Inout
  • Data type
    • Any declared or predefined datatype.

• Sample PORT declaration syntax:

```vhdl
ENTITY test IS
  PORT( name : mode data_type);
END test;
```
Ok... interface has been defined... now what??

Library IEEE;
Use IEEE.std_logic_1164.all;

• Entity IC_7402 is
  Port (  
    outp : out std_logic_vector(3 downto 0);
    inp_a : in std_logic_vector(3 downto 0);
    inp_b : in std_logic_vector(3 downto 0);
  );
End IC_7402;

• Architecture IC_7402_arch of IC_7402 is
  Begin
    outp(3) <= inp_a(3) nor inp_b(3);
    outp(2) <= inp_a(2) nor inp_b(2);
    outp(1) <= inp_a(1) nor inp_b(1);
    outp(0) <= inp_a(0) nor inp_b(0);
  End architecture IC_7402_arch;

Sensitivity list for execution – concept of delta delays

Concurrent statements of execution
Architecture Declaration

- Architecture declarations describe the operation of the component.
- Many architectures may exist for one entity, but only one may be active at a time.
- An architecture is similar to a schematic of the component.

```
ARCHITECTURE behave OF half_adder IS
BEGIN
    PROCESS (enable, x, y)
    BEGIN
        IF (enable = '1') THEN
            result <= x XOR y;
            carry  <= x AND y;
        ELSE
            carry  <= '0';
            result <= '0';
        END IF;
    END PROCESS;
END behave;
```
How does the simulation work?
What is the output of C?

NAND gate evaluated first:
- IN: 1->0
- A: 0->1
- B: 1->0
- C: 0->0

AND gate evaluated first:
- IN: 1->0
- A: 0->1
- C: 0->1
- B: 1->0
- C: 1->0
The two-phase simulation cycle

1) Go through all functions. Compute the next value to appear on the output using current input values and store it in a local data area (a value table inside the function).

2) Go through all functions. Transfer the new value from the local table inside to the data area holding the values of the outputs (=inputs to the next circuit)
Cycle-based simulators

Go through all functions using current inputs and compute next output

Update outputs & increase time with 1 delay unit
Event-based Simulators

if change(i)
q=f(i)

Go through all functions whose inputs has changed and compute next output

o=q
t=t+1

Update outputs & increase time with 1 delay unit
Event-based simulators with event queues

if change(i)
    q=f(i)
    t\text{next}=\text{time}

Go through all functions whose inputs has changed and compute value and time for next output change

\text{t=first}(t\text{next})
\ o=q

Increase time to first scheduled event & update signals
VHDL Simulation Cycle

- VHDL uses a simulation cycle to model the stimulus and response nature of digital hardware.
What did we cover till now...

• Philosophy of VHDL coding...
• Entity-Architecture declarations...
• How simulator works and concept of delta delay...
Before we proceed... lets look at some other VHDL constructs...

- library ieee;
- use ieee.std_logic_1164.all;

---

- entity Comparator is
- Generic (n: natural :=2);
- Port ( A: in std_logic_vector(n-1 downto 0);
- B: in std_logic_vector(n-1 downto 0);
- less: out std_logic;
- equal: out std_logic;
- greater: out std_logic
- );
- end Comparator;
Architecture in a sequential manner

• architecture behav of Comparator is
• begin
• process(A,B)
• begin
• if (A<B) then
• less <= '1';
• equal <= '0';
• greater <= '0';
• elsif (A=B) then
• less <= '0';
• equal <= '1';
• greater <= '0';
• else
• less <= '0';
• equal <= '0';
• greater <= '1';
• end if;
• end process;
• end behv;
Architecture in a concurrent manner

- architecture behav of Comparator is
- begin
- Less <= ‘1’ when a < b else ‘0’;
- Greater <= ‘1’ when a > b else ‘0’;
- Equal <= ‘1’ when a = b else ‘0’;
- End behav;
Testbench

- library ieee;
- use ieee.std_logic_1164.all;
- use ieee.std_logic_unsigned.all;
- use ieee.std_logic_arith.all;

- entity Comparator_TB is
- end Comparator_TB;
Testbench Architecture

- architecture TB of Comparator_TB is

  component Comparator
  port(
    A: in std_logic_vector(1 downto 0);
    B: in std_logic_vector(1 downto 0);
    less: out std_logic;
    equal: out std_logic;
    greater: out std_logic
  );
  end component;

  signal A, B: std_logic_vector(1 downto 0) := "00";
  signal less, equal, greater: std_logic;

  begin
Testbech Architecture ...

Unit: Comparator port map (A, B, less, equal, greater);

process
  variable err_cnt: integer := 0;
begin
  -- Case 1 (using the loop statement)
  A <= "11";
  B <= "00";
  for i in 0 to 2 loop
    wait for 10 ns;
    assert (greater='1') report "Comparison Error detected!"
    severity error;
    if (greater/='1') then
      err_cnt:=err_cnt+1;
    end if;
    B <= B + '1';
  end loop;

  -- Case 2 (using the loop statement)
  A <= "00";
  B <= "01";
  for i in 0 to 2 loop
    wait for 10 ns;
    assert (less='1') report "Comparison Error detected!"
    severity error;
    if (less/='1') then
      err_cnt:=err_cnt+1;
    end if;
    B <= B + '1';
  end loop;

  -- Case 3
  A <= "01";
  B <= "01";
  wait for 10 ns;
  assert (equal='1') report "Comparison Error detected!"
  severity error;
  if (equal/='1') then
    err_cnt:=err_cnt+1;
  end if;

  -- summary of all the tests
  if (err_cnt=0) then
    assert false
    report "Testbench of Adder completed successfully!"
    severity note;
  else
    assert true
    report "Something wrong, try again"
    severity error;
  end if;

  wait;
end process;
end TB;
Configuration

- configuration CFG_TB of Comparator_TB is
  - for TB
  - end for;
- end CFG_TB;
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’.
  
  \[
  \text{eg: } \quad c \leq a \text{ or } b; \\
  \text{is executed when either signal ‘a’ or signal ‘b’ changes.} \\
  \text{process(clk, reset) ...} \\
  \text{is executed when either ‘clk’ or ‘reset’ changes}
  \]

- 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

```vhdl
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 …
begins
    data_bus <= a and b when el = ‘1’
    else
      e or f when a = b else
      g & h when e3 = ‘1’ else
      (others => ‘Z’);
end when_grant;
Selective signal assignment

The \textbf{with} statement

- This statement is similar to the case statement
- syntax

\begin{verbatim}
with expression select
  target <= expression1 when choice1
  expression2 when choice2
  expressionN when choiceN;
\end{verbatim}

- all possible choices must be enumerated
- \textbf{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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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).</td>
</tr>
</tbody>
</table>
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)

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

### Syntax (Instantiation)

```vhdl
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;
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;
Component Instantiation contd.

- **Positional association**

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

- **Named Association**

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

```plaintext
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;
```
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)

```vhdl
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;
beg
process
variable var1: integer := 1;
variable var2: integer := 2;
variable var3: integer := 3;
be
wait on trigger;
var1 := var2 + var3;
var2 := var1;
var3 := var2;
sum <= var1 + var2 + var3;
end process;
end var;
Behavioral Description of a 3-to-8 Decoder

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

```vhdl
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

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

• Delay is created by scheduling a signal assignment for a future time.

• Delay in a VHDL cycle can be of several types
  • Inertial
  • Transport
  • Delta
Inertial Delay

- Default delay type
- Allows for user specified delay
- Absorbs pulses of shorter duration than the specified delay

```
-- Inertial is the default
Output <= NOT Input AFTER 10 ns;
```
Transport Delay

- Must be explicitly specified by user
- Allows for user specified delay
- Passes all input transitions with delay

```plaintext
-- TRANSPORT must be specified
Output <= TRANSPORT NOT Input AFTER 10 ns;
```
Delta Delay

• Delta delay needed to provide support for concurrent operations with zero delay
  – The order of execution for components with zero delay is not clear
• Scheduling of zero delay devices requires the delta delay
  – A delta delay is necessary if no other delay is specified
  – A delta delay does not advance simulator time
  – One delta delay is an infinitesimal amount of time
  – The delta is a scheduling device to ensure repeatability
Example – Delta Delay

Using delta delay scheduling

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>Delta</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>IN: 1-&gt;0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eval inverter</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A: 0-&gt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eval NAND, AND</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>B: 1-&gt;0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C: 0-&gt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>eval AND</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>C: 1-&gt;0</td>
</tr>
</tbody>
</table>
Sequential vs Concurrent Statements

- VHDL provides two different types of execution: sequential and concurrent.
- Different types of execution are useful for modeling of real hardware.
  - Supports various levels of abstraction.
- Sequential statements view hardware from a “programmer” approach.
- Concurrent statements are order-independent and asynchronous.
Sequential Style

XOR-gate

```vhdl
process(a, b)
begin
    if (a /= b) then
        q <= '1';
    else
        q <= '0';
    end if;
end process;
```
Sequential Style Syntax

• Assignments are executed sequentially inside processes.
Concurrent Process Equivalents

- All concurrent statements correspond to a process equivalent.

\[
U0: q \leftarrow a \text{xor} b \text{ after 5 ns;}
\]

is short hand notation for

\[
U0: \text{process}
\]

\[
\begin{align*}
\text{begin} \\
q &\leftarrow a \text{xor} b \text{ after 5 ns;} \\
\text{wait on } a, b; \\
\text{end process;}
\end{align*}
\]
Sequential Statements

- {Signal, Variable} assignments
- Flow control
  - if <condition> then <statments>
  - [elsif <condition> then <statments>]
  - else <statments>
  - end if;
- for <range> loop <statments> end loop;
- while <condition> loop <statments> end loop;
- case <condition> is
  - when <value> => <statements>;
  - when <value> => <statements>;
  - when others => <statements>;
- Wait on <signal> until <expression> for <time>;
Data Objects

• There are three types of data objects:
  • Signals
    • Can be considered as wires in a schematic.
    • Can have current value and future values.
  • Variables and Constants
    • Used to model the behavior of a circuit.
    • Used in processes, procedures and functions.
Constant Declaration

• A constant can have a single value of a given type.
• A constant’s value cannot be changed during the simulation.
• Constants declared at the start of an architecture can be used anywhere in the architecture.
• Constants declared in a process can only be used inside the specific process.

CONSTANT constant_name : type_name [ := value];

CONSTANT rise_fall_time : TIME := 2 ns;
CONSTANT data_bus : INTEGER := 16;
Variable Declaration

- Variables are used for local storage of data.
- Variables are generally not available to multiple components or processes.
- All variable assignments take place immediately.
- Variables are more convenient than signals for the storage of (temporary) data.

```vhdl
VARIABLE variable_name : type_name [:=value];
VARIABLE opcode : BIT_VECTOR(3 DOWNTO 0) := "0000";
VARIABLE freq : INTEGER;
```

Variables are tricky... if you don’t understand them properly, you’ll definitely mess up 😊
Signal Declaration

- Signals are used for communication between components.
- Signals are declared outside the process.
- Signals can be seen as real, physical signals.
- Some delay must be incurred in a signal assignment.

```
SIGNAL signal_name : type_name [:=value];
SIGNAL brdy : BIT;
SIGNAL output : INTEGER := 2;
```
Signal Assignment

• A key difference between variables and signals is the assignment delay.

```
ARCHITECTURE signals OF test IS
  SIGNAL a, b, c, out_1, out_2: BIT;
BEGIN
  out_1 <= a NAND b;
  out_2 <= out_1 XOR c;
END signals;
```

<table>
<thead>
<tr>
<th>Time</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>out_1</th>
<th>out_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1+d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1+2d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
ARCHITECTURE variables OF test IS
BEGIN
  PROCESS (a, b, c)
  VARIABLE a,b,c,out_3,out_4: BIT;
  BEGIN
    out_3 := a NAND b;
    out_4 := out_3 XOR c;
  END PROCESS;
END example;

<table>
<thead>
<tr>
<th>Time</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>out_3</th>
<th>out_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
IF – vs CASE – statement Syntax

if (a='1') then
    q <= '1';
elsif (b='1') then
    q <= '1';
else
    q <= '0';
end if;

case (a&b) is
    when "00" =>
        q <= '0';
    when others =>
        q <= '1';
end case;
FOR – vs WHILE – statement Syntax

```
for i in 0 to 9 loop
    q(i) <= a(i) and b(i);
end loop;
```

For is considered to be a combinational circuit by some synthesis tools. Thus, it cannot have a wait statement to be synthesized.

```
i:=0;
while (i<9) loop
    q <= a(i) and b(i);
    WAIT ON clk UNTIL clk='1';
end loop;
```

While is considered to be an FSM by some synthesis tools. Thus, it needs a wait statement to be synthesized.
WAIT – statement Syntax

• The wait statement causes the suspension of a process statement or a procedure.

  wait [sensitivity_clause] [condition_clause] [timeout_clause];
  • Sensitivity_clause ::= on signal_name
    
    wait on CLOCK;
  • Condition_clause ::= until boolean_expression
    
    wait until Clock = '1';
  • Timeout_clause ::= for time_expression
    
    wait for 150 ns;
Sensitivity-lists vs Wait-on - statement

if you put a sensitivity list in a process, you can’t have a wait statement!

if you put a wait statement in a process, you can’t have a sensitivity list!
Component Declaration

• The component declaration declares the interface of the component to the architecture.
• Necessary if the component interface is not declared elsewhere (package, library).

ARCHITECTURE test OF test_entity
COMPONENT and_gate
  PORT ( in1, in2 : IN BIT;
       out1 : OUT BIT);
END COMPONENT;
... more statements ...
Component Instantiation

- The instantiation statement maps the interface of the component to other objects in the architecture.

```
ARCHITECTURE test OF test_entity
COMPONENT and_gate
    PORT ( in1, in2 : IN BIT;
          out1 : OUT BIT);
END COMPONENT;
SIGNAL S1, S2, S3 : BIT;
BEGIN
    U0 : and_gate PORT MAP (in1 => S1,
                              in2 => S2, out1 => S3);
END test;
```
Component Instantiation Syntax

- The instantiation has 3 key parts
  - Name
  - Component type

U0 : and_gate PORT MAP
(in1 => S1, in2 => S2, out1 => S3);
Supplementary info
VHDL Hierarchy

- Package
  - Generics
  - Entity
    - Ports
    - Architecture
      - Concurrent Statements
    - Architecture
      - Concurrent Statements
    - Architecture (structural)
      - Process
        - Sequential Statements
Std_logic_1164

- The std_ulogic type
- The std_logic type
- The std_ulogic_vector type
- The std_logic_vector type
- The to_bit function
- The to_stdulogic function
- The to_bitvector function
- The to_stdlogicvector function
- The rising_edge function
- The falling_edge function
- The is_x function
std_logic_arith

• The unsigned type
• The signed type
• The arithmetic functions: +, -, *
• The comparison functions: <, <=, >, >=, =, /=
• The shift functions: shl, shr
• The conv_integer function
• The conv_unsigned function
• The conv_signed function
• The conv_std_logic_vector function
std_logic_unsigned

• This library defines all of the same arithmetic (+, -, *), comparison (<, <=, >, >=, =, /=) and shift (shl, shr) operations as the std_logic_arith library. This difference is that the extensions will take std_logic_vector values as arguments and treat them as unsigned integers (ie. just like type unsigned values).

• The function conv_integer is also defined on std_logic_vector and treats the value like an unsigned integer:

  function conv_integer(arg: std_logic_vector) return integer;
VHDL Data Types

- Access
- Composite
  - Array
  - Record
- Scalar
  - Integer
  - Real
  - Enumerated
  - Physical
Predefined Data Types

• bit (‘0’ or ‘1’)
• bit_vector (array of bits)
• integer
• real
• time (physical data type)
Integer

- Integer
  - Minimum range for any implementation as defined by standard: -2,147,483,647 to 2,147,483,647
- Integer assignment example

```vhdl
ARCHITECTURE test_int OF test IS
BEGIN
  PROCESS (X)
  VARIABLE a: INTEGER;
  BEGIN
    a := 1; -- OK
    a := -1; -- OK
    a := 1.0; -- bad
  END PROCESS;
END TEST;
```
Real

- Real
  - Minimum range for any implementation as defined by standard: -1.0E38 to 1.0E38
  - Real assignment example

```vhdl
ARCHITECTURE test_real OF test IS
BEGIN
  PROCESS (X)
  VARIABLE a: REAL;
  BEGIN
    a := 1.3;  -- OK
    a := -7.5; -- OK
    a := 1;   -- bad
    a := 1.7E13;  --OK
    a := 5.3 ns; -- bad
  END PROCESS;
END TEST;
```
Enumerated

- Enumerated
  - User defined range
  - Enumerated example

```vhdl
TYPE binary IS (ON, OFF);
... some statements ...
ARCHITECTURE test_enum OF test IS
BEGIN
  PROCESS (X)
  VARIABLE a: binary;
  BEGIN
    a := ON;     -- OK
    ... more statements ...
    a := off;    -- OK
    ... more statements ...
  END PROCESS;
END TEST;
```
Physical

- Physical
  - Can be user defined range
  - Physical type example

```vhdl
TYPE resistance IS RANGE 0 to 10000000

UNITS
  ohm;  -- ohm
  Kohm = 1000 ohm;  -- 1 K\Omega
  Mohm = 1000 kohm;  -- 1 M\Omega
END UNITS;
```

- Time units are the only predefined physical type in VHDL.
Array

- Array
  - Used to collect one or more elements of a similar type in a single construct.
  - Elements can be any VHDL data type.

```vhdl
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;

O...element numbers... 31
0 | ...array values... | 1

VARIABLE X: data_bus;
VARIABLE Y: BIT

Y := X(12); -- Y gets value of 12th element

TYPE register IS ARRAY (15 DOWNTO 0) OF BIT;
```
Record

- Record
  - Used to collect one or more elements of different types in a single construct.
  - Elements can be any VHDL data type.
  - Elements are accessed through field name.

```vhdl
TYPE binary IS (ON, OFF);
TYPE switch_info IS
  RECORD
    status : binary;
    IDnumber : integer;
  END RECORD;

VARIABLE switch : switch_info;

switch.status := on;  -- status of the switch
switch.IDnumber := 30;  -- number of the switch
```
Subtype

• Subtype
  • Allows for user defined constraints on a data type.
  • May include entire range of base type.
  • Assignments that are out of the subtype range result in error.
  • Subtype example

```
SUBTYPE name IS base_type RANGE <user range>;
SUBTYPE first_ten IS INTEGER RANGE 0 to 9;
```
Natural and Positive Integers

• Integer subtypes:
  • Subtype Natural is integer range 0 to integer’high;
  • Subtype Positive is integer range 1 to integer’high;
Boolean, Bit and Bit_vector

- type Boolean is (false, true);
- type Bit is (‘0’, ‘1’);
- type Bit_vector is array (integer range <>) of bit;
Char and String

- type Char is (NUL, SOH, ..., DEL);
  - 128 chars in VHDL’87
  - 256 chars in VHDL’93

- type String is array (positive range <>) of Char;
IEEE Predefined data types

• type Std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
  • 'U' -- Uninitialized
  • 'X' -- Forcing unknown
  • '0' -- Forcing zero
  • '1' -- Forcing one
  • 'Z' -- High impedance
  • 'W' -- Weak Unknown
  • 'L' -- Weak Low
  • 'H' -- Weak High
  • '-' -- Don’t care

• type std_logic is resolved std_ulogic;
• type std_logic_vector is array (integer range <>) of std_logic;
Assignments

- constant a: integer := 523;
- signal b: bit_vector(11 downto 0);
  
  b <= "000000010010";
  b <= B"000000010010";
  b <= B"0000_0001_0010";
  b <= X"012";
  b <= O"0022";
Vector & Array assignments

- subtype instruction: bit_vector(31 downto 0);
- signal regs: array(0 to 15) of instruction;

```
regs(2) <= regs(0) + regs(1);
regs(1)(7 downto 0) <= regs(0)(11 downto 4);
```
Alias Statement

- Signal instruction: bit_vector(31 downto 0);
- Alias op1: bit_vector(3 downto 0) is instruction(23 downto 20);
- Alias op2: bit_vector(3 downto 0) is instruction(19 downto 16);
- Alias op3: bit_vector(3 downto 0) is instruction(15 downto 12);

  - Op1 <= “0000”;
  - Op2 <= “0001”;
  - Op3 <= “0010”;

  - Regs(bit2int(op3)) <= regs(bit2int(op1)) + regs(bit2int(op2));
Type Conversion (Similar Base)

• Similar but not the same base type:
  
  • signal  i: integer;
  • signal  r: real;

  • i <= integer(r);
  • r <= real(i);
Type Conversion (Same Base)

- Same base type:
  ```
  type a_type is array(0 to 4) of bit;
  signal a:a_type;
  signal s:bit_vector(0 to 4);
  ```

  a<="00101" -- Error, is RHS a bit_vector or an a_type?
  a<=a_type’("00101"); -- type qualifier
  a<=a_type(s); -- type conversion
Type Conversion (Different Base)

- Different base types:
  Function int2bits(value:integer;ret_size:integer) return bit_vector;
  Function bits2int(value:bit_vector) return integer:

    signal i:integer;
    signal b:bit_vector(3 downto 0)

    i<=bits2int(b);
    b<=int2bits(i,4);
Built-In Operators

- Logic operators
  - AND, OR, NAND, NOR, XOR, XNOR (XNOR in VHDL’93 only!!)

- Relational operators
  - =, /=, <, <=, >, >=

- Addition operators
  - +, -, &

- Multiplication operators
  - *, /, mod, rem

- Miscellaneous operators
  - **, abs, not
In all the testbenches we created so far, the test stimuli were coded inside each testbench.

Hence, if we need to change the test stimuli we need to modify the model or create a new model.

Input and output files can be used to get around this problem.
File Definition and Declaration

- A file class needs to be defined before it can be used.

```pln
file_type_defn <= type file_type_name is file of type_mark ;
```

```pln
type integer _file is file of integer ;
```

- Once defined, a file object can be declared.

```pln
file_decl <= file id {, ...} : subtype_indication
[ [ open file_open_kind ] is string_expr ];
```

```pln
type file_open_kind is
(read_mode, write_mode, append_mode);
```

```pln
file table: integer _file open read_mode is "table.dat" ;
```
File reading

- Given a file definition, VHDL implicitly provides the following subprograms:

```vhdl
type file_type is file of element_type;

procedure read ( file f: file_type; value : out element_type;
    length : out natural);

function endfile ( file f: file_type ) return boolean;
```

If the length of the element is greater than the length of the actual data on the file, it is placed left justified in the element.
p1: process is
  type bit_vector_file is file of bit_vectors;
  file vectors: bit_vector_file open read_mode is "vec.dat";
  variable next_vector : bit_vector (63 downto 0);
  variable actual_len: natural;
begin
  while not endfile(vectors) loop
    read (vectors,next_vector,actual_len);
    if actual_len > next_vector’length then
      report "vector too long";
    else
      for bit_index in 1 to actual_len loop
        ....
      end loop;
    end if;
  end loop;
  wait;
end process;
File writing

- Given a file definition, VHDL implicitly provides the following subprograms:

```vhdl
    type file_type is file of element_type;

    procedure write ( file f: file_type; value : in element_type);
```
Problem Description

- Write a process description that writes the data of integer type from an input signal to a file.
- Assume that the input signal “s1” is an “in” port of the top level entity.
- Assume the file name to be “out.dat”.
Example

P1: process (s1) is
    type integer_file is file of integer;
    file out_file: integer_file open write_mode is
        “out.dat”;
begin
    write (out_file,s1);
end;