Introduction to VHDL

A tutorial derived heavily from:
VHDL Tutorial
Jan Van der Spiegel
University of Pennsylvania
Department of Electrical and Systems Engineering

http://www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html

Shreyas Prasad
ECE 412
Spring 2008
Introduction

- VHDL – VHSIC (Very High Speed IC) Hardware Description Language
- Used to *describe* and *simulate* complex digital circuits
- Sponsored by the U.S. Department of Defense

**Example:** $X = A + B$

**Behavioral**

*functionality of digital system*

**Structural**

*logical components of digital system*

**Physical**

*Define the implementation of digital system on logical device*

**Example:** $X = C \text{ AND } (A \text{ OR } B)$

**Example:** Manually define the interconnects on an FPGA
VHDL Basic Structure

• A VHDL entity contains an interface and an architecture

• Interface section
  – Define inputs/outputs to the entity

• Architecture
  – Define *how* interface is implemented
  – Can use either behavioral or structural design to implement interface
Entity (Interface) Template

**General Declarations** (optional) are used to define constants used by the entity
- Constants can be initialized in this section or simply declared
- Commonly used to adjust parameters such as bus width and timing delays
- Syntax:
  
  ```vhdl
  generic(
    constant_name: type [:= value];
    constant_name: type [:= value] );
  ```

**Entity** `NAME_OF_ENTITY` is [ generic `generic_declarations` ];

**Port**

```vhdl
port (signal_names: mode type;

  signal_names: mode type;

  :;

  signal_names: mode type);
```

**Mode** can be one of the following
- `in` - signal is an input
- `out` - signal is an output whole value can be 'read' only be other entities
- `buffer` - signal is an output whose value can be read by other entities as well as this one
- `inout` - signal is both input and output

**Type** can be one of the following
- `bit` - can have the values 1 or 0
- `bit_vector` - a vector of bits
- `std_logic, std_logic_vector` – has all the capabilities of bit (vector) but can also assume other values such as high-impedance (Z), unknown (X), and uninitialized (U)
- Usually default choice for signals!!
- `boolean` - true/false
- `real` - real values
- `integer` - integral values
- `character` - ASCII characters
- `time` - to indicate time


Example Circuit - Buzzer

entity BUZZER is
  port (DOOR, IGNITION, SBELT: in std_logic;
        WARNING: out std_logic);
end BUZZER;

entity mux4_to_1 is generic (HIGH_BIT: integer := 7);
  port (I0,I1,I2,I3: in std_logic_vector (HIGH_BIT downto 0);
        SEL: in std_logic_vector (1 downto 0);
        OUT1: out std_logic_vector(HIGH_BIT downto 0));
end mux4_to_1;
Architecture Body Format

• Defines how what digital circuit 'does' given the interface provided in the entity declaration

```vhdl
architecture architecture_name of NAME_OF_ENTITY is
    -- Declarations (components, signals, constants, functions, procedures, types)
begin
    -- Statements (assign values to the outputs)
end
```
Behavioral Description (Example)

EXAMPLE #1: BUZZER

architecture behavioral of BUZZER is
-- (No declarations here)
begin
  WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);
end behavioral;

EXAMPLE #2: 2-input XNOR gate

entity XNOR2 is
  port ( A, B: in std_logic;
         Z: out std_logic);
end XNOR2;

architecture behavioral_xnor of XNOR2 is
  -- We now have internal signals, X and Y so we have to declare them
  signal X, Y: std_logic;
begin
  X <= A and B;
  Y <= (not A) and (not B);
  Z <= X or Y;
  -- IMPORTANT: THE ABOVE STATEMENTS OCCUR IN PARALLEL!!
end behavioral_xnor;
Structural Description

- Declare what gates (components) are used
  - Declare internal signals if necessary
- Define how the components, inputs and outputs are connected
  - Each 'connection' represented by following statement:
    - label: component-name port map (signal1, signal2, ...)
  - We can explicitly map the ports (recommended for using components with many I/O):
    - label: component-name port map (port1 => signal1, port 2 => signal2, ...)

• Declare what gates (components) are used
  - Declare internal signals if necessary
• Define how the components, inputs and outputs are connected
  - Each 'connection' represented by following statement:
    - label: component-name port map (signal1, signal2, ...)
  - We can explicitly map the ports (recommended for using components with many I/O):
    - label: component-name port map (port1 => signal1, port 2 => signal2, ...)
EXAMPLE #1: BUZZER

**architecture structural of BUZZER is**

-- Declarations of components

```vhdl
component AND2
  port ( in1, in2: in std_logic;
         out1: out std_logic );
end component;

component OR2
  port ( in1, in2: in std_logic;
         out1: out std_logic );
end component;

component NOT1
  port ( in1, in2: in std_logic;
         out1: out std_logic );
end component;
```

-- Declarations of internal signals used to connect gates

```vhdl
signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;
```

```vhdl
begin
  U0: NOT1 port map (DOOR, DOOR_NOT);
  U1: NOT1 port map (SBELT, SBELT_NOT);
  U2: AND2 port map (IGNITION, DOOR_NOT, B1);
  U3: AND2 port map (IGNITION, SBELT_NOT, B2);
  U4: OR2 port map (B1, B2, WARNING);
end structural
```
Hierarchical Design

- Use divide-and-conquer method to solving a large design problem
- Make large designs less complex and more human-readable
- Take advantage of logic that is used multiple times
  - Define a component, then consume it multiple times in another component
Example: A 4 Bit Full Adder

```vhdl
library ieee;
use ieee.std_logic_1164.all;
-- define a full-adder
entity FULLADDER is
  port ( a, b, c: in std_logic);
  sum, carry: out std_logic);
end FULLADDER;
architecture fulladder_behav of FULLADDER is
begin
  sum <= (a xor b) xor c;
  carry <= (a and b) or (c and (a xor b));
end fulladder_behav;
```
-- 4-bit adder
library ieee;
use ieee.std_logic_1164.all;

entity FOURBITADD is
  port ( a, b: in std_logic_vector(3 downto 0);
    Cin : in std_logic;
    sum: out std_logic_vector (3 downto 0);
    Cout, V: out std_logic);
end FOURBITADD;

architecture fouradder_structure of FOURBITADD is
  signal c: std_logic_vector (4 downto 0);
  component FULLADDER
    port( a, b, c: in std_logic;
      sum, carry: out std_logic);
  end component;
begin
  FA0: FULLADDER port map (a(0), b(0), Cin, sum(0), c(1));
  FA1: FULLADDER port map (a(1), b(1), C(1), sum(1), c(2));
  FA2: FULLADDER port map (a(2), b(2), C(2), sum(2), c(3));
  FA3: FULLADDER port map (a(3), b(3), C(3), sum(3), c(4));
  V <= c(3) xor c(4);
  Cout <= c(4);
end fouradder_structure;
Libraries and Packages

- VHDL Packages contain declarations of commonly used objects, data types, components, signals, procedures and functions.

- Need to specify a library and a package:

  ```vhdl
  library ieee;  -- Packages located in ieee library
  use ieee.std_logic_1164.all;  -- This package contains the std_logic signal declaration
  
  • The 'all' keyword means we are using the entire package
  ```

- Sample Packages:
  - std_logic_1164 package: standard datatypes
  - std_logic_arith package: arithmetic, conversion, comparison functions for the signed, unsigned, integer, std_ulogic, std_logic, std_logic_vector types
  - std_logic_unsigned
  - std_logic_misc
Defining a Package (Syntax)

-- Package declaration (analogous to entity)
package name_of_package is
  (package declarations)
end package name_of_package;

-- Package body declarations (analogous to architecture keyword)
package body name_of_package is
  (package body declarations)
end package body name_of_package;
Define a Package (example)

-- Package declaration

library ieee;
use ieee.std_logic_1164.all;
package basic_func is
  -- AND2 declaration
  component AND2 generic (DELAY: time :=5ns);
    port (in1, in2: in std_logic; out1: out std_logic);
  end component;
  -- OR2 declaration
  component OR2 generic (DELAY: time :=5ns);
    port (in1, in2: in std_logic; out1: out std_logic);
  end component;
end package basic_func;
Define a Package (example)

-- Package body declaration

library ieee;
use ieee.std_logic_1164.all;
package body basic_func is
-- 2 input AND gate
entity AND2 is
  generic (DELAY: time);
  port (in1, in2: in std_logic; out1: out std_logic);
end AND2;
architecture model_conc of AND2 is
begin
  out1 <= in1 and in2 after DELAY;
end model_conc;

-- 2 input OR gate
entity OR2 is
  generic (DELAY: time);
  port (in1, in2: in std_logic; out1: out std_logic);
end OR2;
architecture model_conc2 of AND2 is
begin
  out1 <= in1 or in2 after DELAY;
end model_conc2;
end package body basic_func;
Numbers in VHDL

• Default number system is *decimal*

• Exponents:
  - 256e3 , 256E3 , 256e+3 = 256000
  - 1.2e-1 = 0.12

• Numbers in non-decimal bases:
  - Base 2: 2#10010# (Decimal 18)
  - Base 16: 16#1e# (Decimal 30)
Data Objects

- Objects types include signals, variables, and constants
  - So far we have only seen signals of type in or out
  - Variables and constants used to model circuit behavior
    - `constant` list_of_name_of_constant: type ;
      - constant RISE_FALL_TME: time := 2 ns;
      - constant DELAY1: time := 4 ns;
      - constant RISE_TIME, FALL_TIME: time:= 1 ns;
      - constant DATA_BUS: integer:= 16;
    - `variable` list_of_variable_names: type [ := initial value] ;
      - variable CNTR_BIT: bit :=0;
      - variable VAR1: boolean :=FALSE;
      - variable SUM: integer range 0 to 256 :=16;
      - variable STS_BIT: bit_vector (7 downto 0);
  - Variable Assignment: Variable_name := expression;
Variables vs. Signals

- Variable assignment occurs instantly
- Signal assignment occurs after a fixed delay
  - Delay specified by the 'after' keyword followed by a time
  - If not specified, assignment occurs after a delta delay
- Consequences:
  - Variable assignment occurs SEQUENTIALLY
  - Signal assignment occurs at delay offset
    - If multiple assignments occur at a single delay, they occur IN PARALLEL
- Example:

  ```vhdl
  var1 := var2;
  var2 := var1 + var3;
  var3 := var2;
  RESULT <= var1 + var2 + var3;
  ```

  - RESULT is updated only after the three variables have been sequentially updated (new values are used)

  ```vhdl
  sig1 <= sig2;
  sig2 <= sig1 + sig3;
  sig3 := sig2;
  RESULT <= var1 + var2 + var3
  ```

  - All three signals are computed simultaneously using old values of each signal
Common Operators

<table>
<thead>
<tr>
<th>Logical</th>
<th>and</th>
<th>or</th>
<th>nand</th>
<th>nor</th>
<th>xor</th>
<th>xnor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>=</td>
<td>/=</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Shift</td>
<td>sll</td>
<td>srl</td>
<td>sla</td>
<td>sra</td>
<td>rol</td>
<td>ror</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>-</td>
<td>&amp; (concatenation)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplying</td>
<td>*</td>
<td>/</td>
<td>mod</td>
<td>rem (remainder)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>** (exponential)</td>
<td>abs</td>
<td>not</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Remember to include the appropriate libraries
- Examples
  - variable NUM1 :bit_vector := “10010110”;
    NUM1 srl -2; -- Shift left by -2 or shift right by 2
  - MYARRAY (15 downto 0) <= “1111_1111” & MDATA (2 to 9);
    -- Concatination of 2 8-bit vectors
Behavioral Modeling: Sequential Statements

*Process* construct used to describe circuit behavior in terms of sequential statements

- The body is processed only when a signal in the sensitivity list changes
- If sensitivity list is absent a *wait* statement must be included (to ensure that the process will eventually halt)
- A sensitivity list and wait statement cannot be used in the same process construct
- Format:

```
[process_label:] process [ (sensitivity_list) ] [is]
[ process_declarations]
begin
list of sequential statements such as:
signal assignments
variable assignments
case statement
exit statement
if statement
loop statement
next statement
null statement -> useful in case statements
procedure call
wait statement
end process [process_label];
```

Cannot be used outside a *process* construct!
Sequential Statements - Loops

- The `next` statement advances to the next iteration (like the `continue` statement in C)
- The `exit` statement terminates the loop (like the `break` statement in C)
- Iteration scheme can be one of the following:
  - `loop`
  - `while condition loop`
  - `for identifier in range loop`
    - Identifier is a variable index, no need to declare this separately
    - Examples of ranges include `1 to 15` or `5 downto 2`.
- Loop Format:

```
[ loop_label : ]iteration_scheme loop
  sequential statements
  [next [label] [when condition];
  [exit [label] [when condition];
end loop [loop_label];
```
Sequential Statements (Example 1)

Note that entity outputs are not restricted to 'bits' or 'std_logic'!

```vhdl
entity COUNT31 is
    port ( CLK: in std_logic;
            COUNT: out integer);
end COUNT31;
architecture behav_COUNT of COUNT31 is
begin
    P_COUNT: process
    variable intern_value: integer :=0;
    begin
        COUNT <= intern_value;
        loop
            wait until CLK='1';
            intern_value:=(intern_value + 1) mod 32;
            COUNT <= intern_value;
        end loop;
    end process P_COUNT;
end behav_COUNT;
```

- COUNT31 is a 5-bit counter that counts from 0 to 31
- Counter increments on the rising edge of clock
Sequential Statements (Example 2)

- Example: D Flip Flip with Asynchronous clear

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity DFF_CLEAR is
  port (CLK, CLEAR, D : in std_logic;
        Q : out std_logic);
end DFF_CLEAR;

architecture BEHAV_DFF of DFF_CLEAR is
begin
  DFF_PROCESS: process (CLK, CLEAR)
  begin
    if (CLEAR = '1') then
      Q <= '0';
    elsif (CLK'event and CLK = '1') then
      Q <= D;
    end if;
  end process;
end BEHAV_DFF;
```

Evaluates to True only on CLK's rising edge

This block is processed only when CLK or CLEAR change value
Sequential Statements
(Example 3 – A Mealy Machine)
Sequential Statements
(Example 3 – A Mealy Machine)

declaration

entity myvhdl is
  port (CLK, RST, X: in STD_LOGIC;
        Z: out STD_LOGIC);
end;

description

architecture myvhdl_arch of myvhdl is
  -- SYMBOLIC ENCODED state machine: Sreg0
  type Sreg0_type is (S1, S2, S3, S4);
  signal Sreg0: Sreg0_type;
begin
  -- concurrent signal assignments
  Sreg0_machine: process (CLK)
  begin
    if CLK'event and CLK = '1' then
      if RST='1' then
        Sreg0 <= S1;
      else
        case Sreg0 is
          when S1 =>
            if X='0' then
              Sreg0 <= S1;
            elsif X='1' then
              Sreg0 <= S2;
            end if;
          when S4 =>
            if X='0' then
              Sreg0 <= S3;
            elsif X='1' then
              Sreg0 <= S2;
            end if;
          when others =>
            null;
        end case;
      end if;
    end process;
  -- signal assignment statements for combinatorial outputs
  Z_assignment:
  Z <= '0' when (Sreg0 = S1 and X='0') else
   '0' when (Sreg0 = S1 and X='1') else
   '0' when (Sreg0 = S2 and X='1') else
   '1' when (Sreg0 = S4 and X='1') else
   '1';
end myvhdl_arch;
Dataflow Modeling: Concurrent Statements

- Dataflow modeling refers to using concurrent statements to describe behavior
- We have already seen several examples of simple concurrent statements (signal assignments, etc)
- We can also use control statements in dataflow modelling
  - Use the when ... else construct
  - Format:
    
    Target_signal <= expression when Boolean_condition else expression when Boolean_condition else : expression;
  - Example (4 to 1 MUX)

```vhdl
entity MUX_4_1_funcTab is
  port (A, B, C, D: in std_logic;
        SEL: in std_logic_vector (1 downto 0);
        Z: out std_logic);
end MUX_4_1_funcTab;
architecture concurr_MUX41 of MUX_4_1_funcTab is
begin
  Z <= A when SEL = "00" else 
  B when SEL = "01" else 
  C when SEL = "10" else 
  D;
end concurr_MUX41;
```
Dataflow Modeling: Example

entity FullAdd_Conc is
  port (A, B, C: in std_logic;
         sum, cout: out std_logic);
end FullAdd_Conc;

architecture FullAdd_Conc of FullAdd_Conc is
  --define internal signal: vector INS of the input signals
  signal INS: std_logic_vector (2 downto 0);

begin
  --define the components of vector INS of the input signals
  INS(2) <= A;
  INS(1) <= B;
  INS(0) <= C;

  with INS select
  begin
    (sum, cout) <= std_logic_vector’(“00”) when “000”,
                   std_logic_vector’(“10”) when “001”,
                   std_logic_vector’(“10”) when “010”,
                   std_logic_vector’(“01”) when “011”,
                   std_logic_vector’(“10”) when “100”,
                   std_logic_vector’(“01”) when “101”,
                   std_logic_vector’(“01”) when “110”,
                   std_logic_vector’(“11”) when “111”,
                   std_logic_vector’(“11”) when others;
  end FullAdd_Conc;
end;