Writing VHDL for RTL Synthesis

Stephen A. Edwards, Columbia University

March 6, 2012

The name VHDL is representative of the language itself: it is a two-level acronym that stands for VHSIC Hardware Description Language; VHSIC stands for very high speed integrated circuit. The language is vast, verbose, and was originally designed for modeling digital systems for simulation. As a result, the full definition of the language [**?**] is much larger than what we are concerned with here because many constructs in the language (e.g., variables, arbitrary events, floating-point types, delays) do not have hardware equivalents and hence not synthesizable.

Instead, we focus here on a particular dialect of VHDL dictated in part by the IEEE standard defining RTL synthesis [**?**]. Even within this standard, there are many equivalent ways to do essentially the same thing (e.g., define a process representing edge-sensitive logic). This document presents a particular idiom that works; it does not try to define all possible synthesizable VHDL specifications.

1 Structure

Much like a C program is mainly a series of function definitions, a VHDL specification is mainly a series of entity/architecture definition pairs. An entity is an object with a series of input and output ports that represent wires or busses, and an architecture is the "guts" of an entity, comprising concurrent assignment statements, processes, or instantiations of other entities.

Concurrent assignment statements that use logical expressions to define the values of signals are one of the most common things in architectures. VHDL supports the logical operators `and`, `or`, `nand`, `nor`, `xnor`, `xnor`, and `not`.

```
library ieee;                          -- add this to the IEEE library
use ieee.std_logic_1164.all;           -- includes std_ulogic

entity full_adder is
  port(a, b, c    : in  std_ulogic;
       sum, carry : out std_ulogic);
end full_adder;

architecture imp of full_adder is
begin
  sum   <= (a xor b) xor c;            -- combinational logic
  carry <= (a and b) or (a and c) or (b and c);
end imp;
```

1.1 Components

Once you have defined an entity, the next thing is to instantiate it as a component within another entity's architecture.

The interface of the component must be defined in any architecture that instantiates it. Then, any number of `port map` statements create instances of that component.

Here is how to connect two of the full adders to give a two-bit adder:

```
library ieee;
use ieee.std_logic_1164.all;

entity add2 is
  port (
    A, B : in  std_logic_vector(1 downto 0);
    C    : out std_logic_vector(2 downto 0));
end add2;

architecture imp of add2 is
  component full_adder
    port (
      a, b, c    : in  std_ulogic;
      sum, carry : out std_ulogic);
  end component;

  signal carry : std_ulogic;

begin

  bit0 : full_adder port map (
    a     => A(0),
    b     => B(0),
    c     => '0',
    sum   => C(0),
    carry => carry);

  bit1 : full_adder port map (
    a     => A(1),
    b     => B(1),
    c     => carry,
    sum   => C(1),
    carry => C(2));

end imp;
```

1.2 Multiplexers

The `when`...`else` construct is one way to specify a multiplexer.

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer_4_1 is
  port(in0, in1, in2, in3 : in  std_ulogic_vector(15 downto 0);
       s0, s1             : in  std_ulogic;
       z                  : out std_ulogic_vector(15 downto 0));
end multiplexer_4_1;

architecture imp of multiplexer_4_1 is
begin
  z <= in0 when (s0 = '0' and s1 = '0') else
       in1 when (s0 = '1' and s1 = '0') else
       in2 when (s0 = '0' and s1 = '1') else
       in3 when (s0 = '1' and s1 = '1') else
       "XXXXXXXXXXXXXXXX";
end imp;
```

The `with...select` is another way to describe a multiplexer.

```
architecture usewith of multiplexer_4_1 is
  signal sels : std_ulogic_vector(1 downto 0);  -- Local wires
begin
  sels <= s1 & s0;                              -- vector concatenation

  with sels select
    z <=
    in0                when "00",
    in1                when "01",
    in2                when "10",
    in3                when "11",
    "XXXXXXXXXXXXXXXX" when others;
end usewith;
```

1.3 Decoders

Often, you will want to take a set of bits encoded in one way and represent them in another. For example, the following one-of-eight decoder takes three bits and uses them to enable one of eight.

```
library ieee;
use ieee.std_logic_1164.all;

entity dec1_8 is
port (
  sel : in  std_logic_vector(2 downto 0);
  res : out std_logic_vector(7 downto 0));
end dec1_8;

architecture imp of dec1_8 is
begin
  res <= "00000001" when sel = "000" else
         "00000010" when sel = "001" else
         "00000100" when sel = "010" else
         "00001000" when sel = "011" else
         "00010000" when sel = "100" else
         "00100000" when sel = "101" else
         "01000000" when sel = "110" else
         "10000000";
end imp;
```

1.4 Priority Encoders

A priority encoder returns a binary value that indicates the highest set bit among many. This implementation says the output when none of the bits are set is a “don’t-care,” meaning the synthesis system is free to generate any output it wants for this case.

```
library ieee;
use ieee.std_logic_1164.all;

entity priority is
  port (
    sel  : in  std_logic_vector(7 downto 0);
    code : out std_logic_vector(2 downto 0));
end priority;

architecture imp of priority is
begin
  code <= "000" when sel(0) = '1' else
          "001" when sel(1) = '1' else
          "010" when sel(2) = '1' else
          "011" when sel(3) = '1' else
          "100" when sel(4) = '1' else
          "101" when sel(5) = '1' else
          "110" when sel(6) = '1' else
          "111" when sel(7) = '1' else
          "---";                        -- output is a "don't care"
end imp;
```

1.5 Arithmetic Units

VHDL has extensive support for arithmetic. Here is an unsigned 8-bit adder with carry in and out. By default VHDL's + operator returns a result that is the same width as its arguments, so it is necessary to zero-extend them to get the ninth (carry) bit out. One way to do this is to convert the arguments to integers, add them, then convert them back.

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

entity adder is
  port (
    A, B : in  std_logic_vector(7 downto 0);
    CI   : in  std_logic;
    SUM  : out std_logic_vector(7 downto 0);
    CO   : out std_logic);
end adder;

architecture imp of adder is
signal tmp : std_logic_vector(8 downto 0);
begin
  tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B) +
                                conv_integer(CI)), 9);
  SUM <= tmp(7 downto 0);
  CO  <= tmp(8);
end imp;
```

A very primitive ALU might perform either addition or subtraction:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity alu is
  port (
    A, B : in  std_logic_vector(7 downto 0);
    ADD  : in  std_logic;
    RES  : out std_logic_vector(7 downto 0));
end alu;

architecture imp of alu is
begin
  RES <= A + B when ADD = '1' else
         A - B;
end imp;
```

VHDL provides the usual arithmetic comparison operators. Note that signed and unsigned versions behave differently.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity comparator is
  port (
    A, B : in  std_logic_vector(7 downto 0);
    GE   : out std_logic);
end comparator;

architecture imp of comparator is
begin
  GE <= '1' when A >= B else '0';
end imp;
```

Multiplication and division is possible, but is very costly in area and can be very slow.

1.6 Generate statements

To get an unusual array, say that for a 4-bit ripple-carry adder, use a `generate` construct, which expands its body into multiple gates when synthesized.

```
library ieee;
use ieee.std_logic_1164.all;

entity rippleadder is
  port (a, b : in  std_ulogic_vector(3 downto 0);
        cin  : in  std_ulogic;
        sum  : out std_ulogic_vector(3 downto 0);
        cout : out std_ulogic);
end rippleadder;

architecture imp of rippleadder is
  signal c : std_ulogic_vector(4 downto 0);
begin
  c(0) <= cin;
  G1: for m in 0 to 3 generate
    sum(m) <= a(m) xor b(m) xor c(m);
    c(m+1) <= (a(m) and b(m)) or (b(m) and c(m)) or (a(m) and c(m));
  end generate G1;
  cout <= c(4);
end imp;
```

2 State-holding Elements

Although there are many ways to express something that behaves like a flip-flop in VHDL, this is guaranteed to synthesize as you would like

```
library ieee;
use ieee.std_logic_1164.all;

entity flipflop is
  port (Clk, D : in  std_ulogic;
        Q      : out std_ulogic);
end flipflop;

architecture imp of flipflop is
begin
  process (Clk)                        -- Process made sensitive to Clk
  begin
    if (Clk'event and Clk = '1') then  -- Rising edge
      Q <= D;
    end if;
  end process P1;
end imp;
```

Often, you want a synchronous reset on the flip-flop.

```
library ieee;
use ieee.std_logic_1164.all;

entity flipflop_reset is
  port (Clk, Reset, D : in  std_ulogic;
        Q             : out std_ulogic);
end flipflop_reset;

architecture imp of flipflop_reset is
begin
  P1: process (Clk)
  begin
    if (Clk'event and Clk = '1') then
      if (Reset = '1') then Q <= '0';
      else Q <= D;
      end if;
    end if;
  end process P1;
end imp;
```

2.1 Counters

Counters are often useful for delays, dividing clocks, and many other uses. Here is code for a four-bit unsigned up counter with a synchronous reset:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
   port(
     Clk, Reset : in  std_logic;
     Q          : out std_logic_vector(3 downto 0)
   );
end counter;

architecture imp of counter is
  signal count : std_logic_vector(3 downto 0);
  begin
    process (Clk)
    begin
      if (Clk'event and Clk = '1') then
        if (Reset = '1') then
           count <= "0000";
        else
           count <= count + 1;
        end if;
      end if;
    end process;

  Q <= count;

end imp;
```

2.2 Shift Registers

Here is code for an eight-bit shift register with serial in and out.

```
library ieee;
use ieee.std_logic_1164.all;

entity shifter is
  port (
    Clk : in  std_logic;
    SI  : in  std_logic;
    SO  : out std_logic);
end shifter;

architecture impl of shifter is
  signal tmp : std_logic_vector(7 downto 0);
begin
  process (Clk)
  begin
    if (Clk'event and Clk = '1') then
      for i in 0 to 6 loop -- Static loop, expanded at compile time
        tmp(i+1) <= tmp(i);
      end loop;
      tmp(0) <= SI;
    end if;
  end process;

  SO <= tmp(7);
end impl;
```

2.3 RAMs

While large amounts of memory should be stored off-chip, small RAMs (say 32×4 bits) can be implemented directly. Here's how:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_32_4 is
  port (
    Clk  : in  std_logic;
    WE   : in  std_logic;                      -- Write enable
    EN   : in  std_logic;                      -- Read enable
    addr : in  std_logic_vector(4 downto 0);
    di   : in  std_logic_vector(3 downto 0);   -- Data in
    do   : out std_logic_vector(3 downto 0));  -- Data out
end ram_32_4;

architecture imp of ram_32_4 is
  type ram_type is array(31 downto 0) of std_logic_vector(3 downto 0);
  signal RAM : ram_type;
begin
process (Clk)
begin
  if (Clk'event and Clk = '1') then
    if (en = '1') then
      if (we = '1') then
        RAM(conv_integer(addr)) <= di;
        do <= di;
      else
        do <= RAM(conv_integer(addr));
      end if;
    end if;
  end if;
end process;
end imp;
```

Occasionally, an initialized ROM is the most natural way to compute a certain function or store some data. Here is what a synchronous ROM looks like:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rom_32_4 is
  port (
    Clk  : in  std_logic;
    en   : in  std_logic;              -- Read enable
    addr : in  std_logic_vector(4 downto 0);
    data : out std_logic_vector(3 downto 0));
end rom_32_4;

architecture imp of rom_32_4 is
type rom_type is array (31 downto 0) of std_logic_vector(3 downto 0);
constant ROM : rom_type :=
  ("0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000",
   "1001", "1010", "1011", "1100", "1101", "1110", "1111", "0001",
   "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001",
   "1010", "1011", "1100", "1101", "1110", "1111", "0000", "0010");
begin

process (Clk)
begin
  if (Clk'event and Clk = '1') then
    if (en = '1') then
      data <= ROM(conv_integer(addr));
    end if;
  end if;
end process;

end imp;
```

2.4 Finite-State Machines

Write a finite state machine as an entity containing two processes: a sequential process with if statement sensitive to the edge of the clock and a combinational process sensitive to all the inputs of the machine.

```
library ieee;
use ieee.std_logic_1164.all;

entity tlc is
  port (
    clk            : in  std_ulogic;
    reset          : in  std_ulogic;
    cars           : in  std_ulogic;
    short          : in  std_ulogic;
    long           : in  std_ulogic;
    highway_yellow : out std_ulogic;
    highway_red    : out std_ulogic;
    farm_yellow    : out std_ulogic;
    farm_red       : out std_ulogic;
    start_timer    : out std_ulogic);
end tlc;

architecture imp of tlc is
signal current_state, next_state : std_ulogic_vector(1 downto 0);
constant HG : std_ulogic_vector := "00";
constant HY : std_ulogic_vector := "01";
constant FY : std_ulogic_vector := "10";
constant FG : std_ulogic_vector := "11";
begin

P1: process (clk)
begin
  if (clk'event and clk = '1') then
    current_state <= next_state;
  end if;
end process P1;

P2: process (current_state, reset, cars, short, long)
begin
  if (reset = '1') then
    next_state <= HG;
    start_timer <= '1';
  else
    case current_state is
      when HG =>
        highway_yellow <= '0';
        highway_red    <= '0';
        farm_yellow    <= '0';
        farm_red       <= '1';
        if (cars = '1' and long = '1') then
          next_state  <= HY;
          start_timer <= '1';
        else
          next_state  <= HG;
          start_timer <= '0';
        end if;

      when HY =>
        highway_yellow <= '1';
        highway_red    <= '0';
        farm_yellow    <= '0';
        farm_red       <= '1';
        if (short = '1') then
          next_state  <= FG;
          start_timer <= '1';
        else
          next_state  <= HY;
          start_timer <= '0';
        end if;
```

```
      when FG =>
        highway_yellow <= '0';
        highway_red    <= '1';
        farm_yellow    <= '0';
        farm_red       <= '0';
        if (cars = '0' or long = '1') then
          next_state  <= FY;
          start_timer <= '1';
        else
          next_state  <= FG;
          start_timer <= '0';
        end if;

      when FY =>
        highway_yellow <= '0';
        highway_red    <= '1';
        farm_yellow    <= '1';
        farm_red       <= '0';
        if (short = '1') then
          next_state  <= HG;
          start_timer <= '1';
        else
          next_state  <= FY;
          start_timer <= '0';
        end if;

      when others =>
        next_state <= "XX";
        start_timer <= 'X';
        highway_yellow <= 'X';
        highway_red    <= 'X';
        farm_yellow    <= 'X';
        farm_red       <= 'X';
    end case;
  end if;
end process P2;

end imp;
```

Acknowledgements

Ken Shepard's handouts for his EECS E4340 class formed a basis for these examples.

References