Introduction to Hardware Description Languages - VHDL

Luca Macchiarulo

HDL

- Hardware Description Languages extensively used for:
  - Describing (digital) hardware (formal documentation)
  - Simulating it
  - Verifying it
  - Synthesizing it (first step of modern design flow)
- 2 main options:
  - VHDL
  - Verilog
- “Future” or advanced options:
  - SystemC (for complex systems – C-like syntax)
  - SystemVerilog (as SystemC with Verilog-like syntax)
  - VHDL-AMS (Analog-Mixed Signal structure to simulate analog continuous time systems)

VHDL

- VHSIC HDL (=Very High Speed Integrated Circuit HDL)
  - originally DoD project (explains why syntax so close to ADA)
  - IEEE standardized since 1987
    - Most important standardization: 1993
    - currently IEEE2008 (after other 2 minor revisions)
    - Most tools support 1993 – minor differences for synthesis
- Commonly used in academia in Europe, 50 % in US, not so much in Japan
- Full standard very complex – synthesizable version quite minimal (and de facto very similar to Verilog)

Verilog

- Originally company proprietary (Gateway Design Automation then bought by Cadence)
  - IEEE standard from 1995
  - Further revisions 2001/2005 then systemVerilog 2009
- Originally simpler than VHDL, and much closer to common syntax (C-like) – adequate for syntesis
- Historically popular in Japan, and a little over 50% in US

Which one to choose?

- Individual preference – frankly mostly historical (first language learned...)
- But important to have an idea of both
  - I personally write VHDL from scratch, but need to use/ modify Verilog code from others
  - Tools today (almost) seamlessly work with both (careful with interface/naming), so no need to rewrite
- After this week you might have your favorite

More information

- Countless monographs on either of them
  - Chu – FPGA prototyping by VHDL examples Xilinx Spartan-3 version (ebook at UH library)
  - Same as above for Verilog
  - Ashenden – Designer's Guide to VHDL (and also the old but free VHDL Cookbook)
  - Sadeo - The complete Verilog book (ebook at UH library)
- Good Internet coverage
  - VHDL: old but good VHDL FAQ at the University of Hamburg
  - Verilog: http://www.asic-world.com/verilog/index.html
- Xilinx documentation:
  - Simulation and synthesis guide (chapters 4 and 5 is all about HDLs): http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/sim.pdf

Purpose of present week

- Concentrate on <u>synthesizable code</u>
  - Write in such a way that synthesizers will understand what to implement
  - Have in mind HW rather than an algorithm
    - If you think about a classical CS algorithm you almost certainly will not write synthesizable code and certainly code that is very hard to debug
    - Think of multiple object operating in parallel
      - Objects will be components (VHDL) or modules (Verilog)
      - Use master components (State Machines) to organize their communication
      - Check the intended behavior by simulation
  - It is helpful using templates that are guaranteed to work
- Useful non-synthesizable features:
  - Time behavior (after, wait, #)
  - Files
  - Integer, real numbers
- Tricky differences with simulations/post synthesis simulations:
  - “U” signals
  - Sensitivity lists

VHDL Core Elements

- VHDL: Entity and Architecture:

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

Library and Package declarations:
All useful types and functions –
NEEDS TO PRECEDE EVERY
ENTITY THAT USES IT

```
entity simple is
port( a, b, c : in std_logic;
y : out std_logic);
end simple;
```

Entity declaration –
Contains the interface of the component
– inputs and outputs – and possibly some
passing parameters

```
architecture Behavioral of simple is

begin
y<= (a or b) and c;
end Behavioral;
```

Architecture –
Contains the description of how the output
depend on the inputs – there can be multiple
Architecture for the same entity,
but if you just write one the compiler will know
automatically which one to use

Another example – 32 bit adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity simple_adder is
port(a, b: in STD_LOGIC_VECTOR(31 downto 0);
   y: out STD_LOGIC_VECTOR(31 downto 0));
end simple_adder;
architecture Behavioral of simple_adder is
begin
      y <= a + b;
end Behavioral;
```

Packages for arithmetic operations on unsigned numbers

ONLY through them we can write – note that the result has the same size as the operands – if you need carry make sure to extend one of them by one bit

Combinational Logic

- In VHDL simple combinational logic (logic that is not registered or controlled by a clock) can be expressed by a simple **signal assignment operator <=**
  - y <= not a;
- Implicitly, all the signal used in the right hand side expression are used to decide when the expression is evaluated (technically, they are in the “sensitivity list” of the assignment)
- From the point of view of synthesis, this means that the hardware will not contain any FF or latch

Bitwise operators

- The operation can be repeated on all bits in a std_logic_vector:

```
y<= not a;
```

is the same (if y and a are std_logic_vector(2 downto 0)) as

```
y(0)<= not a(0);
y(1)<= not a(1);
y(2)<= not a(2);
```

Conditional Assignment

- Two ways of assigning out of a choice based on another signal:

```
y <= d0 when s = “00” else
     d1 when s = “01” else
     d2 when s = “10” else
     d3;
```

```
with s select y <=d0 when s = “00” else
                        d1 when “01”,
                    d2 when “10”,
                    d3 when others;
```

Internal signal

- Normally in a complex architecture you need intermediate values. They are declared in the architecture before the keyword **begin:**

```
architecture beh of simple
signal p : std_logic;
begin
        p<=a or b;
        y<=p and c;
end beh;
```

Note that all the architecture is still combinational – can be seen as 2 gates connected together – the synthesizer will try to fit it in the smaller number of logic blocks in the target technology (for FPGA, LUTs)

Expressions and precedence

- Inversion: not

concatenation

- Multiplicative: * / mod rem
- Additive: + - &
- Rotate and shift: rol, ror, srl, sll, sra, sla
- Comparison: = /= < <= >= >
- Logical: and or nand nor xor

No xnor!

Numbers

- Integer and real numbers are written normally (130, 0.12)
  - There is no reason to use real numbers but in testbenches for time expressions – they are most clearly non-synthesizable
  - Integers should be used sparingly, as they can take a lot of space (typically 32 bits) but they are ok if used as constants in expressions
- Binary numbers need quotes:
  - “10110010”
  - X”B2”
- Single bits require single quotes '0', '1'

High impedence, uninitialized and invalid

- std_logic is more complex than bit. Besides '0' and '1' it has:
  - 'u' : undefined (at start of simulation before any assignment)
  - 'z' : high impedence (to model tristate buffers)
  - 'x' : invalid (for example when 2 drivers try to force conflicting values like '0' and '1')
  - ...and many more ('H', 'L', 'W', '-')
- Useful in simulation, ignored in synthesis
- But good to know as they explain strange warnings (not all options covered...)

Bit swizzling

- It is easy to break up and build a new vector:

  y(5 downto 3) <= a(2 downto 1) & '0';

- The strange “downto” makes sure numeric vectors are read correctly with constants:
  - Y(3 downto 0) <= “1100”; – assigns the (decimal) number 12 to the 4 Lsbits of Y

Delays

- For simulation (but NOT for synthesis) it is possible to fix the delays in assignments:
  - y<=a after 2 ns;
  - Z <= b + c after 10 ns;
- Synthesis does not use them at all, and if you want to set a timing constraint, you need to follow a different route (see the next days).

Structural modelling

- How to describe very complex systems?
  - By having a hierarchy of modules, in which one is used in a higher module
  - An entity/architecture pair that is used in a higher level module is called a **component**
  - The specific match of a component to an entity/architecture is supposed to be explicit (using **configuration**s)
  - Implicit configurations are possible, if we are lazy, but the trick is making sure we use the same name for the component and entity and their signals as well

Example – leaf module

```
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Mux2 is
port(
a,b : in std_logic;
s : in std_logic;
y : out std_logic);
end Mux2;

architecture Behavioral of Mux2 is

begin

y <= b when s = '1' else a;

end Behavioral;

```

Example – hierarchy top

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

entity Mux4 is
port(
a,b,c,d : in std_logic;
s : in std_logic_vector(1 downto 0);
y : out std_logic);
end Mux4;

architecture Behavioral of Mux4 is
signal m0, m1 : std_logic;

component Mux2 is
port(
a,b : in std_logic;
s : in std_logic;
y : out std_logic);
end component;

begin

Mux_I:  Mux2 port map(
a => a,
b => b,
s => s(0),
y => m0
);

Mux_II:  Mux2 port map(
a => c,
b => d,
s => s(0),
y => m1
);

Mux_y:  Mux2 port map(
a => m0,
b => m1,
s => s(1),
y => y
);
```

Same as entity !

Assignments of formal => actual argument:
c is a signal in Mux4, a is its name in Mux2

Side note – strange/confusing conventions

- VHDL has some subtle conventions and symbols that easily cause subtle syntax errors if not correct:
  - Semicolons: just before a closed parenthesis (in a port map, for example), no semicolons:
    - port(a, b, c : in std_logic;y : out std_logic);
  - Test for equality is = not ==
  - Test for inequality is /= not !=
  - There are 2 “arrow” signs: <= is used in signal assignments, => is used for port maps and case statements (see below)
  - If statements ALWAYS require **then** and **end if;** - but no need for parenthesis to bracket the expression
  - Alternative if is **elsif**, not else if nor elseif
  - Comments are introduced by – not //

Sequential Logic

- Most designs use sequential logic – logic that has memory of the past, and typically employs a clock signal to time the computation or data transfer
- In VHDL this needs to be described using **process**es
- The key for the synthesizer to recognize sequential logic is the fact that the assignments are not continuous – there are conditions in which nothing happens now – see the D-FF of the following page

D-FF

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity D_FF is
port(
clk : in std_logic;
d : in std_logic;
q : out std_logic);
end D_FF;

architecture Behavioral of D_FF is

begin

process(clk)
begin
if rising_edge(clk) then -- rising_edge(clk) is a shortcut for (clk'event and clk = '1')
   q<=d;
end if;
end process;

end Behavioral;
```

Sensitivity list: only when there is a change in the listed signals the process is active

This is happening only at the rising edge of clk, all other times q remembers the old value

Resettable registers

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


entity Reg_res is
port(
clk,rst : in std_logic;
d : in std_logic_vector(7 downto 0);
q : out std_logic_vector(7 downto 0));
end Reg_res;

architecture Behavioral of Reg_res is

begin

process(rst,clk)
begin
if rst = '1' then
   q<= (others => '0');
elsif rising_edge(clk) then
   q<=d;
end if;
end process;

end Behavioral;
```

Asynchronous reset:
It operates independently of clk

Synchronizer

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

entity Syncrhonizer is
port(
clk: in std_logic;
d : in std_logic_vector(7 downto 0);
q : out std_logic_vector(7 downto 0));
end Syncrhonizer;

architecture Behavioral of Syncrhonizer is

signal n : std_logic_vector(7 downto 0);
begin


process(clk)
begin
if rising_edge(clk) then
    n<=d;
    q<=n;
end if;
end process;

end Behavioral;
```

IMPORTANT!
The assignments are performed using the d and n values BEFORE the clock transition so this introduces an extra clock delay!

Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity counter is
port(
clk,rst : in std_logic;
q : out std_logic_vector(7 downto 0));
end counter;

architecture Behavioral of counter is

signal int_q :  std_logic_vector(7 downto 0);

begin


process(clk)
begin
if rising_edge(clk) then
   if rst = '1' then
      int_q<= (others => '0');
   else
      int_q<= int_q + 1;
   end if;
end if;
end process;

q<=int_q;

end Behavioral;
```

This is required as VHDL
does not allow out signals to be read
inside an architecture

Latch

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

entity Latch is
port(
load : in std_logic;
d : in std_logic;
q : out std_logic);
end Latch;

architecture Behavioral of Latch is

begin

process(load,d)
begin
if load = '1' then
   q <= d;
end if;
end process;

end Behavioral;
```

Note the sensitivity list

No “rising_edge”

NOTE: Xilinx discourages the use of latches and asynchronous resets – mostly for timing closure issues, but they are occasionally handy (to record one time flags, for example)

Combinational processes

- Processes can be also used to describe complex combinational logic, not only sequential:
  - The trick is to make sure that ALL possible changes in the used signals are actually taken into consideration
  - This requires having all “input” signals in the sensitivity list (if not, the system might work but the simulation might not match the synthesis)
  - And also making sure that all “outputs” are assigned a value in the process no matter the value of any signal

Example of wrong “combinational” process

```
process(a)
begin
    if a = “00” then
        y <= '0';
    elsif a = “01” then
        y <=b;
    elsif a = “10” then
        Y<=b;
    end if;
end process;
```

b missing from sensitivity list

Ooops – what happens if a = “11”?

Corrected “combinational” process

```
process(a,b)
begin
    if a = “00” then
        y <= '0';
    elsif a = “01” then
        y <=b;
    elsif a = “10” then
        y<=b;
    else
        y <=b;
    end if;
end process;
```

All used signals in sensitivity list

Else guarantees all cases are covered

Statements allowed inside a process

- Variable assignments:
  - variables are declared in the declarative part of the process (not the architecture – they are like local variables)
  - variables can be used as signals in expressions
  - variables are assigned using the variable assignment operator **:=** that has immediate effect – the result of an assignment can be used in successive parts of the same process

Example of variable use

```
process(clk)
    variable v1,v2: std_logic_vector(7 downto 0);
begin
    if rising_edge(clk) then
        v1:= a + b;
        v2 := c + d;
        y<= v1+v2;
    end if;
end process;
```

At every clock cycle, y will be updated
with the current value of a+b+c+d. What would have
Happened if v1 and v2 were signals?

Statements allowed inside a process

- Case statements:

```
case a is
    when "00" => y<='0';
    when "01" => y<=b;
    when "10" => y<=b;
    when others => y<=b;
end case;
```

Statements allowed inside a process

- Some statements are very useful for simulation but are meaningless and should not be used for synthesis:
  - Wait for xx ns → used to introduce a finite delay between one statement and the next – no code with this will be synthesizable (we'll see it in testbenches)
  - Assert <condition> report “blah blah” severity error
    - Useful to check satisfaction of logic constraints

Finite State Machines (FSM)

- Finite State Machines are typically key sequential components for any design – coordinate action among other components
- Abstractly defined by a number of **states** and possible **transitions** between states, a transition baing chosen based on current inputs
- Actions (outputs) are assigned to either the states (**Moore** machine) or the transitions (**Mealy** machine)
- Less abstractly, in Moore machines states depend only on the current state, in Mealy machines on state and current inputs.

FSM descriptions

- FSM can be described in various ways in VHDL. The most common ways are trying to mimic the HW implementation of FSMs:
  - 2 processes, 1 updating the state at the clock tick, 1 computing the future state and outputs
  - 3 processes, 1 updating the state at the clock tick, 1 computing the future state, 1 computing the output (can easily distinguish Moore and Mealy machines).
  - A single process, updating a single state signal inside a clock "if" statement – only for Moore machines
- It typically uses:
  - An enumerative type to define the states symbolically
  - case statements to distinguish between the current state

FSM example – 2 processes

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

entity FSM is
port(
clk : in std_logic;
a : in std_logic;
o : out std_logic
);
end FSM;

architecture Behavioral of FSM is
type state_t is (S0, S1, S2);
signal next_state, current_state : state_t;
begin
--- state updating
process(clk)
begin
if rising_edge(clk) then
   current_state <= next_state;
end if;
end process;

--next state and output updating
process(current_state, a)
begin
case current_state is
   when S0 => o <= '1';
              if a='1' then next_state <= S1; else next_state <= S0; end if;
   when S1 => o <= '0';
              if a='1' then next_state <= S2; else next_state <= S1; end if;
   when S2 => o <= '0';
              if a='1' then next_state <= S0; else next_state <= S2; end if;
end case;
end process;

end Behavioral;
```

This process needs to be completely combinational – next_state needs to be assigned for any possible input

Testbenches

- Once a module is written and passes syntax checks, how do we test it behaves as expected?
  - Simulation of course, but how do we feed the inputs?
  - Building a **testbench**: a VHDL code that can be simulated and that provides all appropriate inputs and relative timing
- A testbench:
  - Is an empty entity (no inputs or outputs – and no port!)
  - uses wait constructs to add delays and synchronize the inputs with clocks

Example

Testbench for the FSM

```
    -- Clock period definitions
    constant clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: FSM PORT MAP (
            clk => clk,
            a => a,
            o => o
          );

    -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        a <= '0';
        wait for 100 ns;
        wait for clk_period*10;
        -- insert stimulus here
        a <= '1';
        wait for clk_period;
        a <= '0';
        wait for clk_period;
        a <= '1';
        wait for clk_period;
        a <= '0';
        wait for clk_period;
        a <= '1';
        wait for clk_period;
        wait;
    end process;

END;
```

Instance of “Unit Under Test”

Sequence of inputs

No further change from here