

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 synthesis
- 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 synthesizable code
 - 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;
```

Packages for arithmetic operations on unsigned numbers



```
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;
```

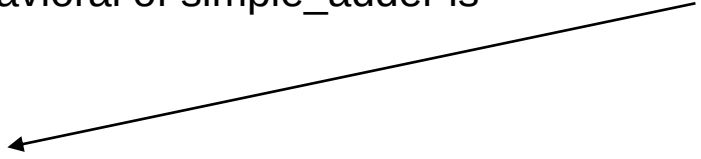
```
architecture Behavioral of simple_adder is
```

```
begin
```

```
    y <= a + b;
```

```
end Behavioral;
```

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
```

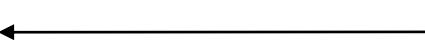
```
    p<=a or b;
```

```
    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
- 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 impedance, 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 impedance (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:
 $y(5 \text{ downto } 3) \leq a(2 \text{ downto } 1) \& '0';$
- The strange “downto” makes sure numeric vectors are read correctly with constants:
 - $Y(3 \text{ downto } 0) \leq "1100";$ – 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:
 - $y \leq a$ after 2 ns;
 - $Z \leq b + c$ 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 **configurations**)
 - 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

```
12  --
13  -- Dependencies:
14  --
15  -- Revision:
16  -- Revision 0.01 - File Created
17  -- Additional Comments:
18  --
19  -----
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22
23  -- Uncomment the following library declaration if using
24  -- arithmetic functions with Signed or Unsigned values
25  --use IEEE.NUMERIC_STD.ALL;
26
27  -- Uncomment the following library declaration if instantiating
28  -- any Xilinx primitives in this code.
29  --library UNISIM;
30  --use UNISIM.VComponents.all;
31
32  entity Mux2 is
33  port(
34  a,b : in std_logic;
35  s : in std_logic;
36  y : out std_logic);
37  end Mux2;
38
39  architecture Behavioral of Mux2 is
40
41  begin
42
43  y <= b when s = '1' else a;
44
45  end Behavioral;
46
```

Example – hierarchy top

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 entity Mux4 is
24 port(
25   a,b,c,d : in std_logic;
26   s : in std_logic_vector(1 downto 0);
27   y : out std_logic);
28 end Mux4;
29
30 architecture Behavioral of Mux4 is
31   signal m0, m1 : std_logic;
32
33   component Mux2 is
34   port(
35     a,b : in std_logic;
36     s : in std_logic;
37     y : out std_logic);
38   end component;
39
40   begin
41
42     Mux_I:  Mux2 port map(
43       a => a,
44       b => b,
45       s => s(0),
46       y => m0
47     );
48
49     Mux_II:  Mux2 port map(
50       a => c,
51       b => d,
52       s => s(0),
53       y => m1
54     );
55
56     Mux_y:  Mux2 port map(
57       a => m0,
58       b => m1,
59       s => s(1),
60       y => y
61     );
62
```

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 **processes**
- 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

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23
24 entity D_FF is
25 port(
26   clk : in std_logic;
27   d : in std_logic;
28   q : out std_logic);
29 end D_FF;
30
31 architecture Behavioral of D_FF is
32
33 begin
34
35   process(clk)
36   begin
37     if rising_edge(clk) then -- rising_edge(clk) is a shortcut for (clk'event and clk = '1')
38       q<=d;
39     end if;
40   end process;
41
42 end Behavioral;
```

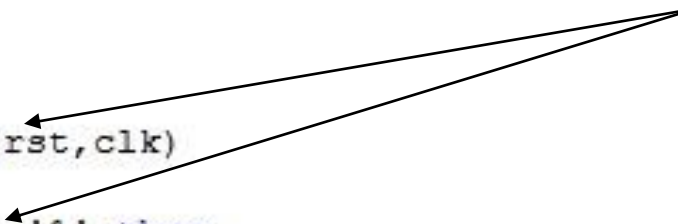
Sensitivity list: only when there is a change in the listed signals the process is active

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

Resettable registers

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23
24 entity Reg_res is
25 port(
26   clk,rst : in std_logic;
27   d : in std_logic_vector(7 downto 0);
28   q : out std_logic_vector(7 downto 0));
29 end Reg_res;
30
31 architecture Behavioral of Reg_res is
32
33 begin
34
35   process(rst,clk)
36   begin
37     if rst = '1' then
38       q<= (others => '0');
39     elsif rising_edge(clk) then
40       q<=d;
41     end if;
42   end process;
43
44 end Behavioral;
```

Asynchronous reset:
It operates independently of clk



Synchronizer

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 entity Synchronizer is
24 port(
25   clk: in std_logic;
26   d : in std_logic_vector(7 downto 0);
27   q : out std_logic_vector(7 downto 0));
28 end Synchronizer;
29
30 architecture Behavioral of Synchronizer is
31
32   signal n : std_logic_vector(7 downto 0);
33   begin
34
35
36   process(clk)
37   begin
38     if rising_edge(clk) then |
39       n<=d;
40       q<=n;
41     end if;
42   end process;
43
44   end Behavioral;
```

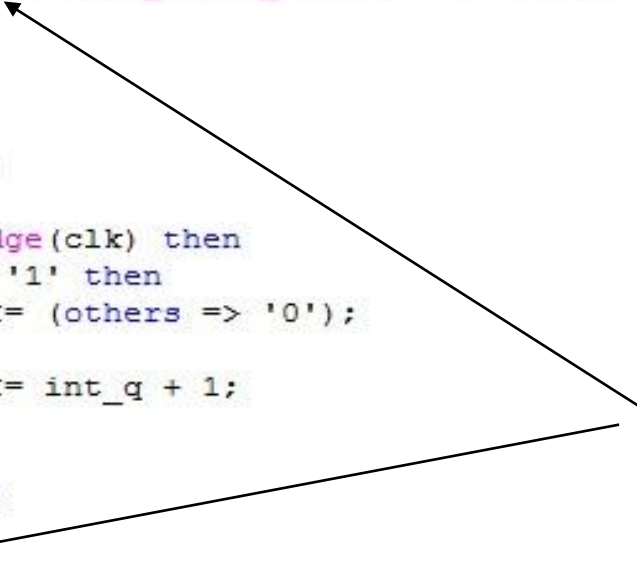
IMPORTANT!

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

Counter

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_arith.ALL;
23 use IEEE.STD_LOGIC_unsigned.ALL;
24 |
25 entity counter is
26 port(
27   clk,rst : in std_logic;
28   q : out std_logic_vector(7 downto 0));
29 end counter;
30
31 architecture Behavioral of counter is
32
33   signal int_q : std_logic_vector(7 downto 0);
34
35   begin
36
37
38   process(clk)
39   begin
40     if rising_edge(clk) then
41       if rst = '1' then
42         int_q<= (others => '0');
43       else
44         int_q<= int_q + 1;
45       end if;
46     end if;
47   end process;
48
49   q<=int_q;
50
51 end Behavioral;
```

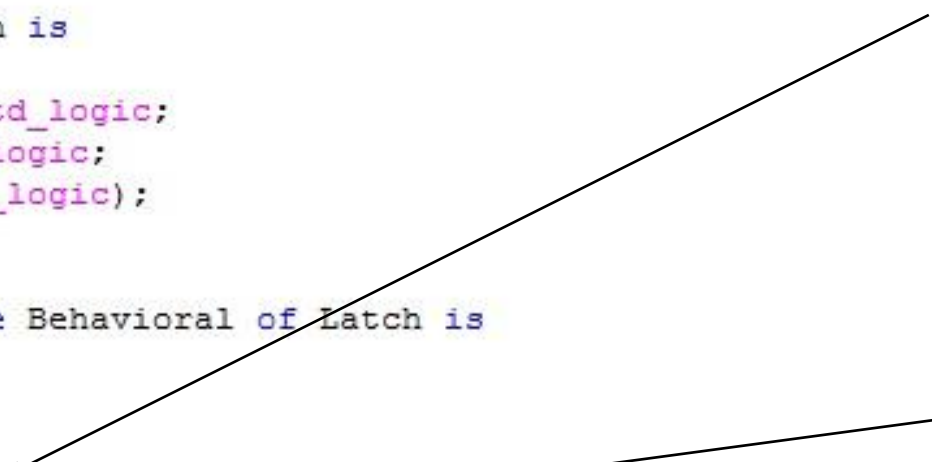
This is required as VHDL
does not allow out signals to be read
inside an architecture



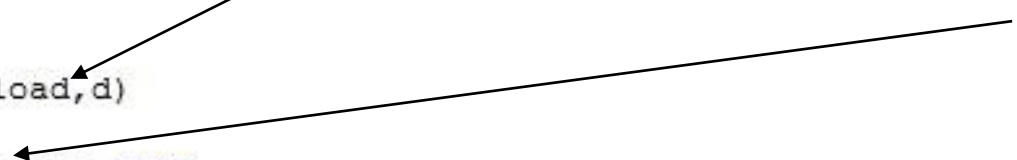
Latch

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 |
23 entity Latch is
24 port(
25   load : in std_logic;
26   d : in std_logic;
27   q : out std_logic);
28 end Latch;
29
30 architecture Behavioral of Latch is
31
32 begin
33
34   process(load,d)
35   begin
36     if load = '1' then
37       q <= d;
38     end if;
39   end process;
40
41 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;
```

Diagram illustrating the corrected “combinational” process. The process is defined as `process(a,b)`. The sensitivity list is `a` and `b`. The process body contains an `if` statement with three branches: `if a = "00" then`, `elsif a = "01" then`, and `elsif a = "10" then`. The `else` branch is present. The process is terminated by `end process;`. Annotations indicate that the sensitivity list must include all signals used in the process body, including `b`, which is used in the assignment `y <= b;` in the `elsif a = "01" then` branch and the `else` branch.

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 being 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

```
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 entity FSM is
24 port(
25   clk : in std_logic;
26   a : in std_logic;
27   o : out std_logic
28 );
29 end FSM;
30
31 architecture Behavioral of FSM is
32   type state_t is (S0, S1, S2);
33   signal next_state, current_state : state_t;
34 begin
35   --- state updating
36   process(clk)
37   begin
38     if rising_edge(clk) then
39       current_state <= next_state;
40     end if;
41   end process;
42
43   --next state and output updating
44   process(current_state, a)
45   begin
46     case current_state is
47       when S0 => o <= '1';
48                 if a='1' then next_state <= S1; else next_state <= S0; end if;
49       when S1 => o <= '0';
50                 if a='1' then next_state <= S2; else next_state <= S1; end if;
51       when S2 => o <= '0';
52                 if a='1' then next_state <= S0; else next_state <= S2; end if;
53     end case;
54   end process;
55
56 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

```
58  -- Clock period definitions
59  constant clk_period : time := 10 ns;
60
61  BEGIN
62
63  -- Instantiate the Unit Under Test (UUT)
64  uut: FSM PORT MAP (
65      clk => clk,
66      a => a,
67      o => o
68  );
69
70  -- Clock process definitions
71  clk_process :process
72  begin
73      clk <= '0';
74      wait for clk_period/2;
75      clk <= '1';
76      wait for clk_period/2;
77  end process;
78
79  -- Stimulus process
80  stim_proc: process
81  begin
82      -- hold reset state for 100 ns.
83      a <= '0';
84      wait for 100 ns;
85      wait for clk_period*10;
86      -- insert stimulus here
87      a <= '1';
88      wait for clk_period;
89      a <= '0';
90      wait for clk_period;
91      a <= '1';
92      wait for clk_period;
93      a <= '0';
94      wait for clk_period;
95      a <= '1';
96      wait for clk_period;
97      wait;
98  end process;
99
100 END;
```

Instance of “Unit Under Test”

Sequence of inputs

No further change from here