Lecture 3: Writing VHDL Code for Synthesis

The objective of this supplemental material is to provide guidelines for writing VHDL code for synthesis.

1. Introduction

The quality of a synthesized design, in terms of area, performance, etc., depends directly on the VHDL description of the design. Generally, two different VHDL descriptions of the same design may result in two different final implemented circuits. Also, the final implemented design depends on what software tools you use. Adopting a VHDL programming style, which ensures best synthesized designs, is desirable. During this lab you will learn how to write VHDL constructs that are efficiently synthesized.

2. Synthesis tools

Usually, there are several ways to express the functionality of a design. For example, the following VHDL code describes an edge triggered D-flip-flop in four different ways:

```
-- version 1
process (clk) is
begin
if rising_edge(clk) then
      q <= d;
end if;
end process

-- version 2
process is
begin
wait until rising_edge(clk);
      q <= d;
end process

-- version 3
q <= d when rising_edge(clk) else q;

-- version 4
b: block (rising_edge(clk) and not clk'stable) is
begin
      q <= guarded d;
end block b;
```

It is unlikely that all the above descriptions will be synthesizeable by the same tool. This depends on how the tool is constructed, i.e., what are the "expectations" of the tool for certain expressions. That is why it is recommended that you 1) read the documentation of your particular tool and 2) possibly change your programming style to conform to the particular requirements specific to your tool.

3. Potentially synthesizable

VHDL can be utilized to describe design for the purposes of 1) simulation and 2) synthesis. There are constructs (closer to the C programming constructs) included in the VHDL language which are intended for simulation and which cannot be synthesized into hardware. File operations and assertion statements are such kind of constructs. Such constructs should be used for creating testbenches but not for the synthesizable sections of a model.

There are constructs which are potentially synthesizable but are not handled correctly by some synthesis tools. If you use these constructs, the synthesized hardware will produce different results from the simulated model. For example, suppose you want to describe a registered comparator with two data inputs, a and b, a clock input, clk, and a data output, q. The device stores the result of comparing a and b on each rising edge of clk. The following are two possible ways to describe this circuit:

```
-- version 1
process (clk) is
      variable d : std_logic;
begin
      if a=b then
            d:='1';
      else
            d:='0';
      end if;
      if rising_edge(clk) then
            q <= d;
      end if;
end process

-- version 2
process (clk) is
begin
if rising_edge(clk) then
      if a=b then
            q <= '1';
      else
            q <= '0';
      end if;
end if;
end process
```

When you simulate the first version it works correctly. When you synthesize it you have to take into account the fact that the process is resumed on both rising and falling edges of clk. The variable d is updated in both cases and in this way it is a function of clk. Some tools treat this as illegal and fail to synthesize the device. Others proceed to synthesize the device, but may not produce a correct circuit. Version 2 is a better description since it reflects accurately your intention that the comparison is performed only on rising edges of clk. In this case the process does not contain any unnecessary implied state.

4. "Doing it right" vs. "Doing it wrong"

a) "Doing it right"

```
y <= a or b; -- simple gate, easy to synthesize

y <= a when x = '1' else b; -- simple multiplexer, no process
                            -- statement necessary!
```

Extend VHDL code already written to describe new blocks. Example which uses the description of a flip-flop to specify a counter:

```
-- flipflop description
ff2 : process (reset, clk) is
begin
      if reset = '1' then
            q <= '0';
      elsif rising_edge(clk) then
            if x = '1' then
                  q <= 'a';
            else
                  q <= 'b';
            end if;
      end if;
end process ff2;

-- flipflop extended to form a counter
constant terminal_count : integer := 2**6-1;
subtype counter_range is integer range 0 to terminal_count;
signal count : counter_range;
...
counter6: process (reset, clk) is
begin
      if reset = '0' then
            count <= '0';
      elsif rising_edge(clk) then
            if count < terminal_count then
                  count <= count + 1;
            else
                  count <= '0';
            end if;
      end if;
end process counter6;
```

Describe Finite State Machines (FSMs) using two-processes model:

```
architecture behavioral of an_FSM is
type state_type is (S0, S1, S3, S4);
signal state, next_state : state_type;
begin

combinational_part: process (input, state) is
begin
case state is
    when S0 =>
```

```
      if input = '1' then
            output <= '1';
            next_state <= S0;
      else
            output <= '1';
            next_state <= S1;
      end if;
    when S1 =>
      if input = '1' then
            output <= '0';
            next_state <= S1;
      else
            output <= '0';
            next_state <= S0;
      end if;
      when S2 =>
      if input = '1' then
            output <= '0';
            next_state <= S1;
      else
            output <= '0';
            next_state <= S3;
      end if;
    when S3 =>
      if input = '1' then
            output <= '0';
            next_state <= S3;
      else
            output <= '1';
            next_state <= S0;
      end if;
end case;
end process combinational_part;

state_register: process (reset, clk) is
begin
if reset = '0' then
      state <= S0;
elsif rising_edge(clk) then
      state <= next_state;
end if;
end process state_register;

end architecture behavioral;
```

With this type of description the register which holds the current state is independent of the logic that determines the next state and the outputs. Synthesis tools work better with the state machine specified in this way. In the above example, the four states can be encoded using, two, three or four bits. The best encoding depends on the synthesis target library, the required speed of the circuit, and the circuit area available. You can force the above machine to use **"one hot" state encoding** by modifying the state definition as follows:

```
-- forcing one hot encoding
subtype state_type is std_logic_vector (3 downto 0);
constant S0 : state_type := "0001";
constant S1 : state_type := "0010";
constant S2 : state_type := "0100";
constant S3 : state_type := "1000";
```

b) "Doing it wrong"

```
-- Wrong:
y <= a + b + c + d; -- will be synthesized as three stage circuit!
-- Correct:
y <= (a + b) + (c + d); -- will be synthesized as two stage circuit!

-- Wrong:
y <= a or b or c and d; -- wrong if you want (a+b)+(cd)!
                        -- Recall the operator associativity.
-- Correct:
y <= (a or b) or (c and d);
```

Incomplete definitions:

```
-- Wrong version because there is uncertainty about what is x when a = 1
-- and about what is z when NOT(a = 1).
if (a = '1' ) then
      z <= f();
elsif (clk'event and clk = '1') then
      x <= g();
end if;

-- Correct:
if (a = '1' ) then
      z <= f(); x <= x;
elsif (clk'event and clk = '1') then
      x <= g(); z <= z;
end if;

-- Wrong construct; because there is an else after a clocked if. Try
-- to draw a schematic and figure it out what is wrong.
if (clk'event and clk = '1') then
      x <= f();
      y <= g();
else
      z <= h();
end if;
```

Avoid putting too much on the sensitivity list of processes! Usually we put on the sensitivity list clocks, resetting signals and inputs. Do not include in the sensitivity lists output signals!

5. Distinguishing when to use signals and when to use variables

The behavior of signals and variables can be completely different. Variables can only be used to store data only temporarily in a process or subprogram.

```
-- Undesired construct:
signal int : std_logic;
begin
      process(a, b, c, d, int) is
      begin
            int <= a and b and c;
            q <= int and d;
      end process;
end;
```

It is undesired because we assign the signal int inside the process and then use it to assign q inside the same process. Because int is updated only after a delta delay, in the current step int has still the old (incorrect) value. To get around this, int has to be on the sensitivity list, and thus the process will be activated again. But according to the previous guideline, you have to avoid overloaded sensitivity list! A better option to write the above construct is:

```
-- Better construct:
begin
      process(a, b, c, d) is
      variable int : std_logic;
      begin
            int := a and b and c;
            q <= int and d;
      end process;
end;
```

This will present also the advantage of a faster simulation because, now, the process will be executed only once! The advantage of the version using int declared as signal is that int can be used as a waveform in the simulator. This is not possible if int is declared as a variable because no **time** is linked to a variable. This makes it harder to debug the variable example then the signal example!
Rule: Use variables only when you want to store a value temporarily!

6. Others

Declaring vectors:

```
-- NOT recommended:
signal a : std_logic_vector (0 to 3);
-- Recommended: (because the MSB will be always the one with the highest index)
signal a : std_logic_vector (3 downto 0);
```

Counter synthesis:
The following is the description of a counter without resetting line:

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity COUNTER is
port ( CLK : in std_ulogic;
      Q : out integer range 0 to 15 );
end COUNTER;

architecture my_cool_arch of COUNTER is
      signal COUNT : integer range 0 to 15 ;
begin
process (CLK)
begin
      if (CLK'event and CLK = '1') then
```

```
            if (COUNT >= 9) then
                COUNT <= 0;
            else
                COUNT <= COUNT + 1;
            end if;
      end if;
end process;
Q <= COUNT;
end my_cool_arch;
```

Note the range assignment in the port declaration of the output. Here, only integers between 0 and 15 are allowed - which means that 4 bits are sufficient for the binary representation of the output port. The port Q is replaced by the synthesis tool with a 4 bit signal (ultimately all types are transformed by means of the synthesis tools to std_logic types).
Note also that Q, as an output port, can be only written and it cannot be read within the architecture declaration (unless its mode is changed to buffer). Therefore, a signal COUNT must be declared within the architecture to be able to query COUNT >= 9. The result of the counting is finally transferred to Q in a concurrent signal assignment, Q<=COUNT, as an additional process. That means that each change of COUNT triggers the assignment Q<=COUNT. Thus, the internal IF assignment describes the combinatorial circuit before the FF. The number of FFs is derived from the width of the signal, which receive an assignment inside the outer IF assignment. In this example, the width is four for signal COUNT (because of its range 0 to 15).
Finally note that only signal CLK is on the sensitivity list.

7. Conclusion

The discussion in this supplemental material is not meant to be an exhaustive list of how VHDL code should be written for synthesis. Rather, the purpose is to provide a rough idea about what writing code for synthesis means. Remarkably, the intent is to make you aware of possible situations where the VHDL description performs correctly during simulation but it does **not** after synthesis and implementation (or even worse: code is **not** synthesizeable in the first place)!