Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Universidade Federal de Santa Catarina
Florianópolis, Brazil

August 2013

Contents

Chapter 9. More on Processes and Registers

The behavioral description of sequential circuits in VHDL has been discussed in previous chapters. In Chapter 7 and in Chapter 8, the VHDL process statement has been used to describe FSMs. In Chapter 6, the process statement is introduced, and used to describe the behavior of latches, flip-flops and registers. This chapter discusses the use of processes in the implementation of both, combinational and sequential circuits. At the end of the chapter, the students should be able:

- to understand the difference between implicit and explicit processes;
- to design combinational and sequential circuits using implicit and explicit processes;
- to implement shift registers in VHDL.

9.1 Implicit and Explicit Processes

Implicit and explicit processes have been used so far in VHDL examples all over this book. In implicit processes, the reserved word "process" is not used. Examples of implicit process include:

- Simple signal assignments, where any event in the source side of the assignment will trigger the process as, for instance, *SW*, *A*, *B*, and *C* next:
  - LEDR <= SW;
  - F <= (A and B) or C;
- Signal assignments using selection (*with* / *select*) and conditional (*when*) statements;
- Instantiated components (*component* / *portmap*).

Explicit processes are easily identified in a VHDL code by the usage of the reserved word "process". The two VHDL codes in Figure 9.1, when synthesized, will generate the same circuit. The code in the left hand side has three "implicit" processes performing their signal assignment operations in parallel. The code in the right hand side performs the same functionality as the code on the left, but the signal assignment operations are placed inside three "explicit" processes. As shown in Figure 9.3, both implicit and explicit processes implementations listed in Figure 9.1 generate their Y output in parallel to the E and F assignments.

The VHDL code in Figure 9.2 has the same three assignments as the codes in Figure 9.1, but the circuit functionality is slightly different. As shown in Figure 9.4, as the assignments appear in the same process, the Y output is not updated simultaneously to the E and F assignments. The Y output is updated next time the process is triggered and, consequently, it provides previous values of E and F. In Figure 9.4, when the simulation

starts, A = B = C = D = ‘0’, and the Y output is undefined. Next, when A = B = ‘1’, the Y output is ‘0’, representing the previous circuit output when all input signals were ‘0’.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity Implicit is
port (
        A, B, C, D: in std_logic;
        Y         : out std_logic
);
end Implicit;
architecture Arch of Implicit is
   signal E, F: std_logic;
begin
        E <= A and B;
        F <= C and D;
        Y <= E or F;
end Arch;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity Explicit_v1 is
port (
        A, B, C, D : in std_logic;
        Y          : out std_logic
);
end Explicit_v1;
architecture Arch of Explicit_v1 is
    signal E, F: std_logic;
begin
    process (A, B)
    begin
        E <= A and B;
    end process;
    process (C, D)
    begin
        F <= C and D;
    end process;
    process (E, F)
    begin
        Y <= E or F;
    end process;
end Arch;
```

Figure 9.1. Implicit and explicit processes in VHDL.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity Explicit_v2 is
port (
        A, B, C, D : in std_logic;
        Y    : out std_logic
);
end Explicit_v2;
architecture Arch of Explicit_v2 is
   signal E, F: std_logic;
begin
    process (A, B, C, D)
    begin
        E <= A and B;
        F <= C and D;
        Y <= E or F;
    end process;
end Arch;
```

Figure 9.2. Sequential signal assignments in an explicit process.

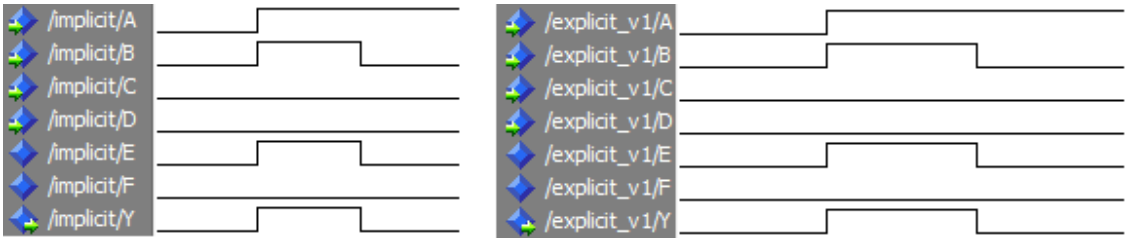


Figure 9.3. Simulation waveforms for the VHDL codes listed in Figure 9.1, showing their equivalence.

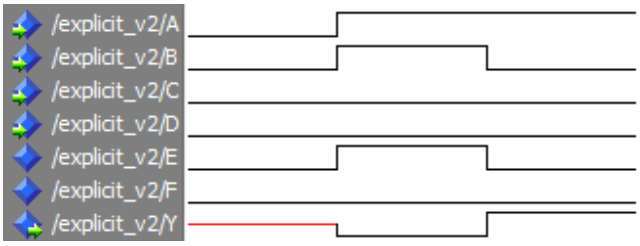
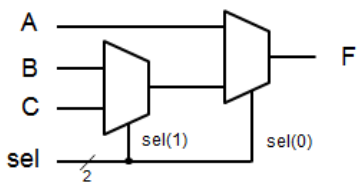


Figure 9.4. Simulation waveform for the VHDL code listed in Figure 9.2.

In Figure 9.5, a multiplexer with priority is implemented using implicit (left hand side) and explicit (right hand side) processes. In VHDL, the IF statement can be used only in explicit processes. The synthesized multiplexer has a priority, as when *sel(0)* is '0', the *F* output is assigned the value in the *A* input, no matters the *sel(1)* value. The *C* input has the lowest assignment priority.

```
architecture implicit of y is
begin

   F <= A when sel(0) = '0' else
        B when sel(1) = '0' else
        C;

end implicit;
```

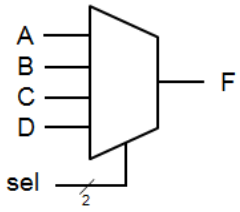



```
architecture explicit of y is
begin

 process (A, B, C, sel)
 begin
     if sel(0) = '0' then
       F <= A;
     elsif sel(1) = '0' then
       F <= B;
     else
       F <= C;
     end if;
  end process;

end explicit;
```

Figure 9.5. Conditional assignment with priority using the "comparison" mechanism, implemented with implicit and explicit processes.

In Figure 9.6, the selection mechanism is used in order to implement a no priority multiplexer. VHDL implicit processes use the *with/select/when* construction to synthesize no priority multiplexers. In explicit process, this mechanism is synthesized by using the *case/when* construction. In Figure 9.6, as well as in Figure 9.5, both the implicit and the explicit process implementation, generate the same piece of hardware.

```
architecture implicit of y is
begin
    with sel select
    F <= A when "00",
         B when "01",
         C when "10",
         D when others;
end implicit;
```




```
architecture explicit of y is
begin
  process (sel, A, B, C, D)
  begin
      case sel is
          when "00" =>
                F <= A;
          when "01" =>
                F <= B;
          when "10" =>
                F <= C;
          when others =>
                F <= D;
      end case;
   end process;
end explicit;
```

Figure 9.6. Conditional assignment with no priority using the "selection" mechanism, implemented with implicit and explicit processes.

9.2 Designing a Shift Register

A shift register is a sequential circuit used to shift bits in a word. In a shift left register, the bits are moved to the left, and the most significant bit is lost, as shown in Figure 9.7. In a shift right register, the bits are moved to the right and the less significant bit is lost. In a circuit that rotates the bits to the left, the most significant bit of a word is moved to the less significant bit position. In a rotate right circuit, the bits are moved to the right, and the less significant bit is moved to the most significant position.

The VHDL behavioral description of registers discussed in previous chapters can be easily modified in order to implement shift and rotate registers. It is important to notice that registers are implemented in VHDL using explicit processes.

However, instead of reusing and adapting VHDL code from previous chapters, in this case study a template provided by Quartus II is used for the implementation of a shift register. To insert templates in a design, the target VHDL file must be opened in Quartus II text editor. Next, select the menu *Edit –> Insert Template…* and *VHDL –> Shift Registers –> Basic Shift Registers*, as shown in Figure 9.8.

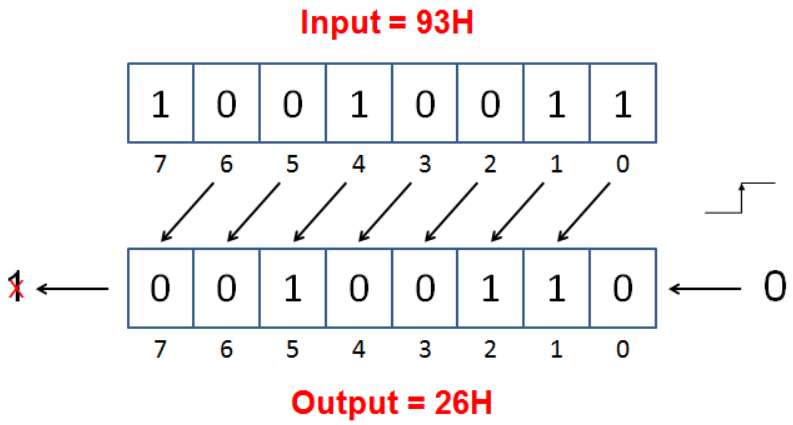


Figure 9.7. Example of shift left operation.

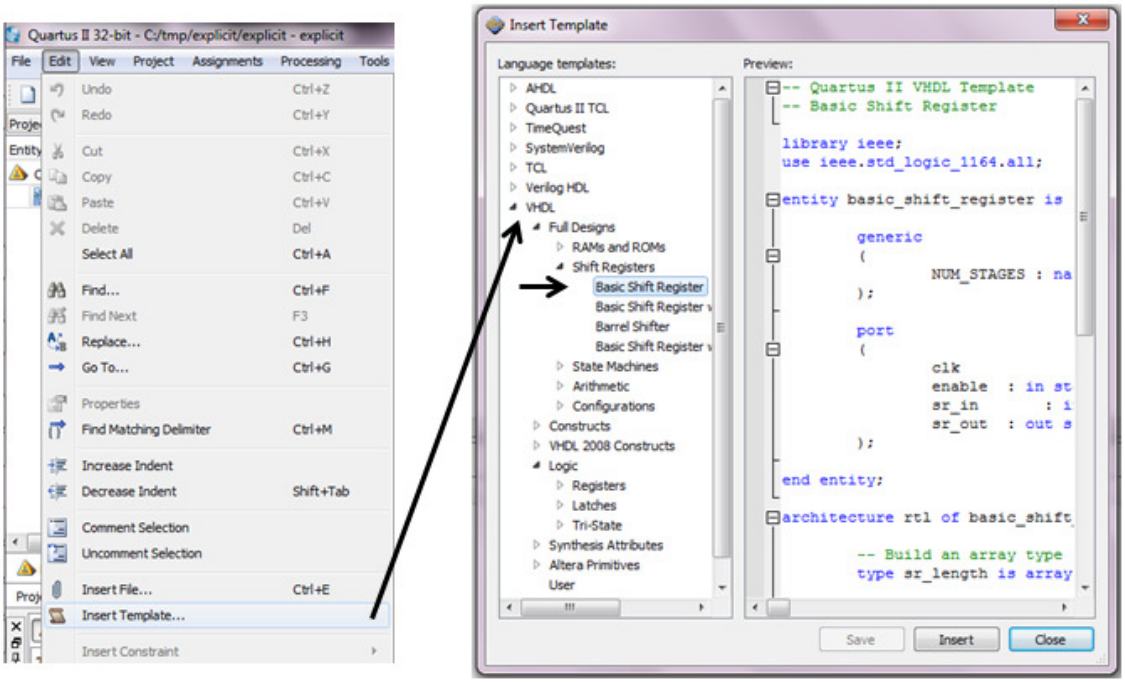


Figure 9.8. Using Quartus II to insert VHDL templates in a design.

In the proposed case study, an N bits input word is shifted left by one bit at each rising clock edge, and send to an output signal. This functionality is implemented by a shift register with enable and asynchronous reset. The original shift register template provided by Quartus II is changed, in order to implement the proposed functionality. The list of changes are as follows:

- A reset signal has been added;
- The “array” type was removed;
- The new input data is N bits long (instead of a single bit);
- After shifting, the less significant bit receives '0', and not the input bit.

The modified version of the shift register template is listed in Figure 9.9, and it has the following features:

- On line 4, the "generic" statement is used to make it easier to adapt this shift register implementation to different word sizes. For instance, in this example an 8 bits shift register is created, as N is replaced by 8 all over the code (see lines 8, 9, 13, and 22).
- The input (*sr_in*) and output (*sr_out*) data are 8 bits long (see lines 8 and 9).
- Each time the *reset* signal is not '0', *enable* is '1', and in the event of a rising clock edge, the 7 most significant bits of the internal signal *sr* receive the 7 less significant bits of the input data *sr_in* (see line 22). Also, the less significant bit of *sr* receives '0' (see line 23). The result is the input data sr_in shifted 1 bit to the left, as shown in Figure 9.7.
- The circuit output is provided on line 27, when the internal signal *sr* is assigned to the output signal *sr_out*.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity basic_shift_register is
4     generic(N : natural := 8);
5     port (clk    : in std_logic;
6           enable : in std_logic;
7           reset  : in std_logic;
8           sr_in  : in std_logic_vector((N - 1) downto 0);
9           sr_out : out std_logic_vector((N - 1) downto 0)
10    );
11 end entity;
12 architecture rtl of basic_shift_register is
13    signal sr: std_logic_vector ((N-1) downto 0);
14 begin
15    process (clk, reset)
16    begin
17       if (reset = '0') then
18          sr <= (others => '0');
19       elsif (rising_edge(clk)) then
20          if (enable = '1') then
21             -- Shift left 1 bit each clock
22             sr((N-1) downto 1) <= sr_in((N-2) downto 0);
23             sr(0) <= '0';
24          end if;
25       end if;
26    end process;
27    sr_out <= sr;
28 end rtl;
```

Figure 9.9. Modified Quartus II shift register template.

The synthesized shift register component is shown in Figure 9.10.

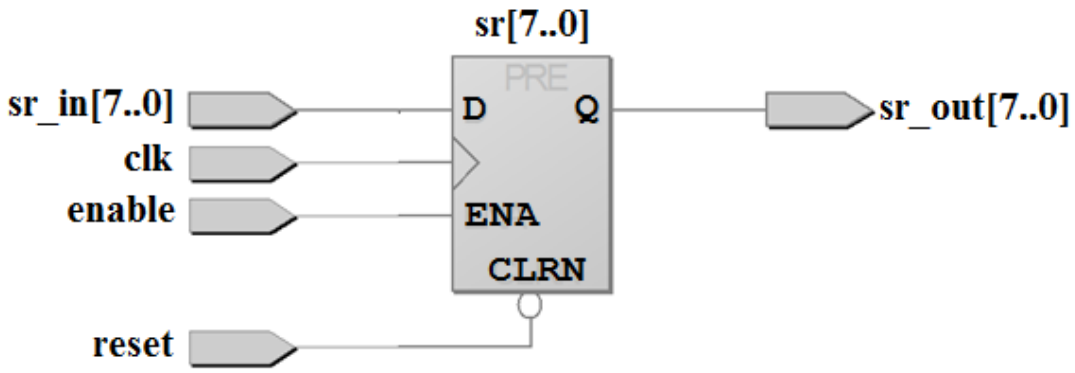


Figure 9.10. Shift register synthesized by Quartus II from VHDL listed in Figure 9.9.

9.3 Laboratory Assignment

The laboratory objectives are:

- to fix the concepts of shift register design in VHDL;
- to insert multiplication and division by 2 operations to the calculator case study.

Multiplication and division by 2 are simple operations to be performed in the base 2 numeral system. To multiply a binary number by 2, just shift the number 1 bit to the left and fill the less significant bit with '0'. The division by 2 is performed in a similar way, but shifting the number 1 bit to the right, and inserting a '0' in the most significant bit position. The operation represented in Figure 9.7 is a multiply by 2, which resulted in an overflow as the target register is only 8 bits long, and the result is 9 bits long.

The component implemented in the shift register case study, and listed in Figure 9.9, performs a multiply by 2 operation.

Laboratory Session

The tasks to be performed in this laboratory session are as follows:

- Using a shift register, design a component to perform division by 2 operations (*shift right*).
- Using a shift register, design a circuit to perform multiplication by 2 operations (*shift left*).
- Perform the simulation of both components, before inserting them in the calculator.

- Make the appropriate changes in the calculator implemented in previous chapters, in order to include the multiplication and division components:
  - The new multiplication and division components, as well as the not operation component, have also just one input. The FSM must be changed in order to deal with this situation.
  - Replace the XOR component by the new divide by 2 component.
  - Replace the NOT component by the new multiply by 2 component.

Figure 9.11 shows the block diagram of the new version for the calculator. The new components need a clock and a reset signals, but there is no need for an enable input. For instance, the component described in Figure 9.9 shifts its input value 1 bit left. Thus, when the user provides a value to be multiplied by 2, and selects the multiplication operation, the mux output will be the input value shifted 1 bit left. At each rising clock edge, the multiplication component shifts its input 1 bit left and, while the input value remains the same, the output value will be also remain unchanged (but shifted 1 bit left in relation to the input value).

So, the shift register listed in Figure 9.9 can be used to implement the multiply by 2 component, but without lines 20 and 24 as the enable input is not used in Figure 9.11.

The division by 2 component can also be easily implemented from Figure 9.9, but now the input value must be shifted to the right, and a '0' should be inserted in the most significant position bit.

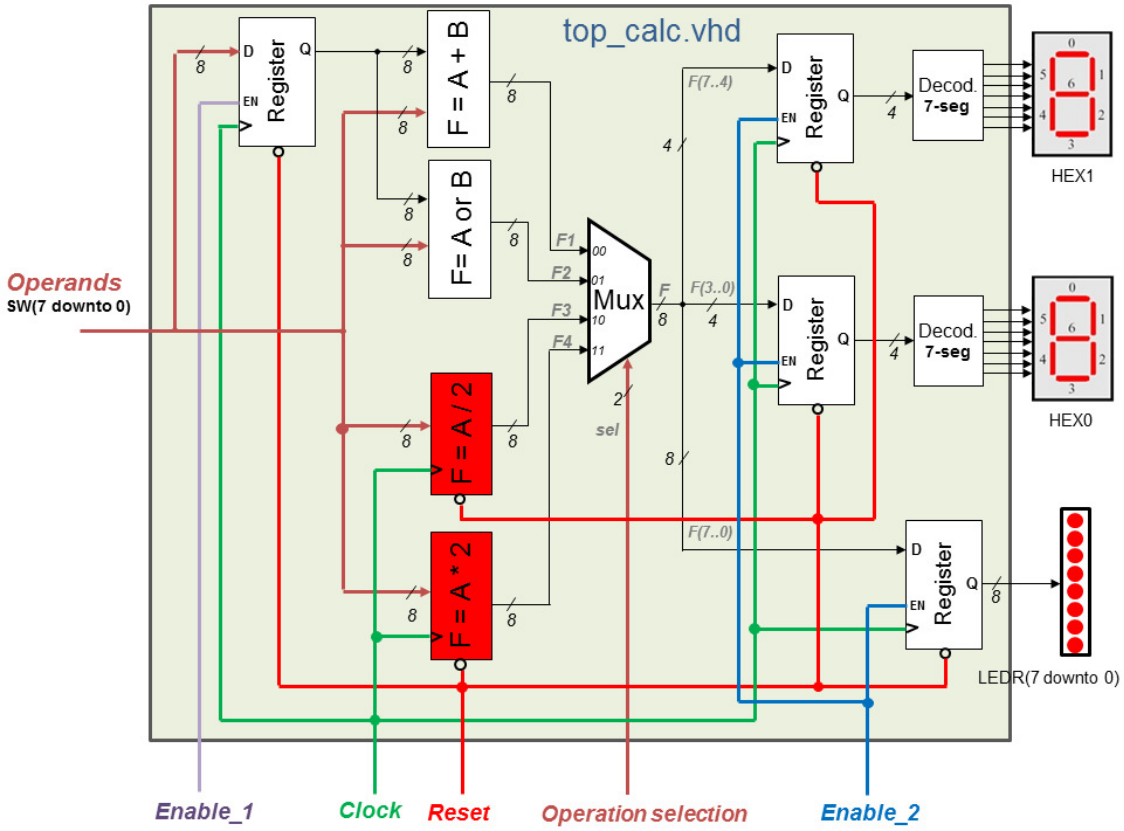


Figure 9.11. Inserting the divide and multiply components to the calculator.

Figure 9.12 shows the full block diagram of the calculator, including the controller FSM. As there is no need for enable signals in the new shift registers, the FSM can be used with minor changes. Basically, the FSM should be changed in order to manage two 1-input components, instead of just one 1-input component (NOT in the previous calculator version).

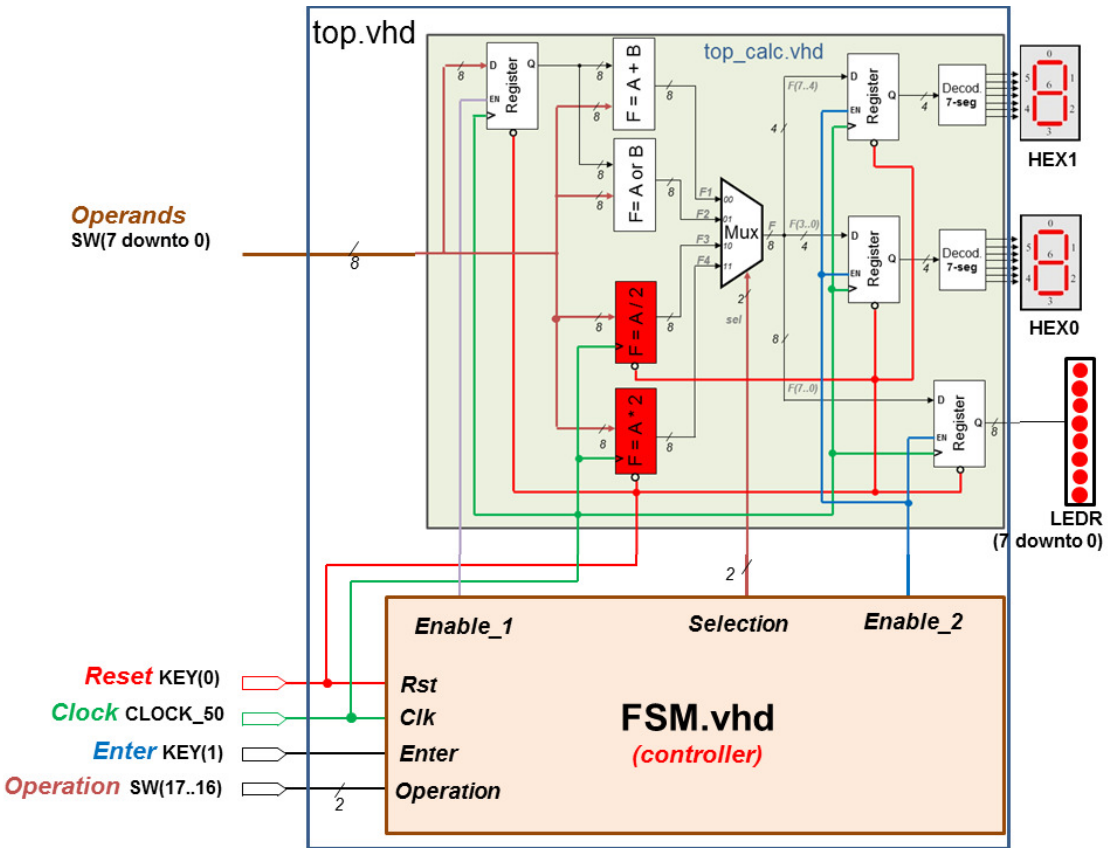


Figure 9.12. The complete calculator design with the new components and the controller FSM.