ENGINE

Teaching online electronics, microcontrollers and programming in Higher Education

Hardware Implementation of Algorithms

3. State machine in VHDL. Shift register.

Lead Partner: Warsaw University of Technology

Authors: Lukasz Mik

University of Applied Sciences in Tarnow

Erasmus+

GRANT NUMBER: 2020-1-PL01-KA226-HE-095653

Declaration

This laboratory instruction has been prepared in the context of the ENGINE project. Where other published and unpublished source materials have been used, these have been acknowledged.

Copyright

Funding Disclaimer

This project has been funded with support from the European Commission. This report reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

I. State machine in VHDL

Finite state machines consist of a finite set of states and a set of state transitions. They are used in applications where unusual sequences of control output signals are required. An important feature of such an automaton is the ability to assign any sequence of logical states to its output states. One example of the use of such an automaton is a controller for changing traffic lights at an intersection or an unusual code counter.

1. State machine based on constants

The following listing shows the code for an unusual up and down counter.

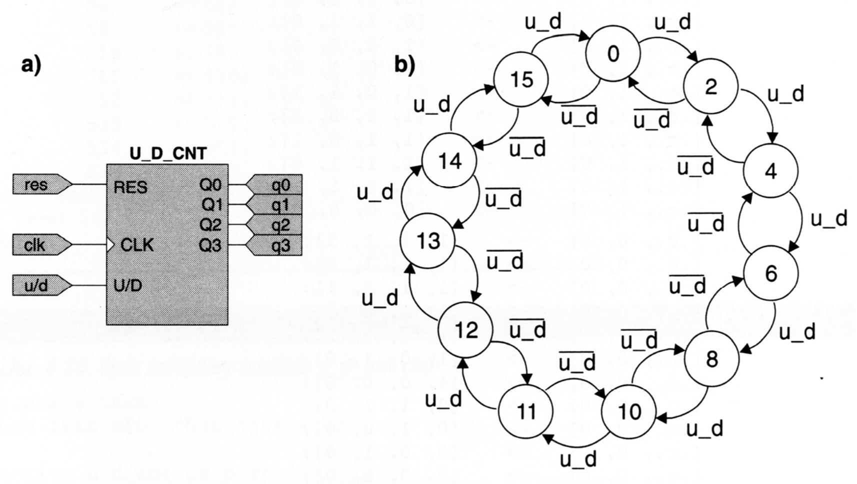
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity u_d_cnt is port (
       clk, res, u_d: in std_logic;
       q: out std_logic_vector(3 downto 0)
       );
end u_d_cnt;

architecture example_arch of u_d_cnt is
      signal state: std_logic_vector(3 downto 0);
      constant S0: std_logic_vector(3 downto 0) := "0000";
      constant S2: std_logic_vector(3 downto 0) := "0010";
      constant S4: std_logic_vector(3 downto 0) := "0100";
      constant S6: std_logic_vector(3 downto 0) := "0110";
      constant S8: std_logic_vector(3 downto 0) := "1000";
      constant S10: std_logic_vector(3 downto 0) := "1010";
      constant S11: std_logic_vector(3 downto 0) := "1011";
      constant S12: std_logic_vector(3 downto 0) := "1100";
      constant S13: std_logic_vector(3 downto 0) := "1101";
      constant S14: std_logic_vector(3 downto 0) := "1110";
      constant S15: std_logic_vector(3 downto 0) := "1111";
begin
process (clk, res) begin
  if (res = '0') then state <= "0000";
      elsif (clk='1' and CLK'event) then
             if u_d = '1' then
                case state is
                    when S0 => state <= S2;
                    when S2 => state <= S4;
                    when S4 => state <= S6;
                    when S6 => state <= S8;
                    when S8 => state <= S10;
                    when S10 => state <= S11;
                    when S11 => state <= S12;
                    when S12 => state <= S13;
                    when S13 => state <= S14;
                    when S14 => state <= S15;
                    when others => state <= S0;
                end case;
             elsif u_d = '0' then
                case state is
```

```
                        when S0 => state <= S15;
                        when S2 => state <= S0;
                        when S4 => state <= S2;
                        when S6 => state <= S4;
                        when S8 => state <= S6;
                        when S10 => state <= S8;
                        when S11 => state <= S10;
                        when S12 => state <= S11;
                        when S13 => state <= S12;
                        when S14 => state <= S13;
                        when S15 => state <= S14;
                        when others => state <= S0;
                    end case;
                end if;
            end if;
 q <= state;
end process;
end example_arch;
```

The counting direction depends on the ***u_d*** input state. The counter counts in a cycle: 0, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15, 0, 2 ... (up) ... or 0, 15, 14, 13, 12, 11, 10 8, 6, 4, 2, 0, 15, 14 … (down). The counter states are changed under the influence of clock pulses fed to the ***clk*** input, and resetting is provided by the ***res*** asynchronous reset input. The graphic symbol of the project (a) and the corresponding transition graph (b) are shown in the figure below.

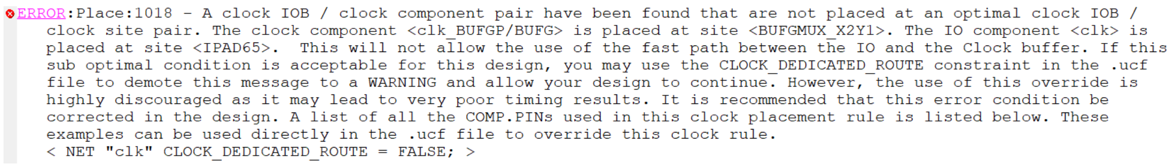


In case of defining a sequence of output states of the automaton, it is necessary to specify all states occurring in it, and for each current state to determine its subsequent states. When we use the ***case*** instruction, the condition for the correctness of the description is to define all possible values of the selection variable (in the example it is the ***state*** signal).

The automaton system in the example consists of 4 flip-flops that can take 16 states. As can be seen from the description, in the normal work cycle of the automaton only 11 states were used, and the remaining 5 are unused. The correct logical synthesis of the VHDL description will be possible only when the next states after the occurrence of one of the states outside the normal cycle are explicitly defined, which can be done in the simplest way with the command `when others => state <= S0;`.

- ***Implementation of a state machine with a button as the clock source***

Launch *ISE Design Suite 14.7* from the desktop shortcut or from *Start* → *Programs* → *Xilinx Design Tools* → *64-bit Project Navigator*. If there are any projects open, close them all by selecting *File* → *Close Project*. Next, create a new project named ***u_d_cnt*** and add the source files from the tutorial: ***u_d_cnt.vhd*** and ***u_d_cnt.ucf***.
Then synthesize the project and try to generate a **.bit* (bitstream) configuration file.
When trying to implement the project in the target system, an error will occur that will prevent the bitstream file from being generated..

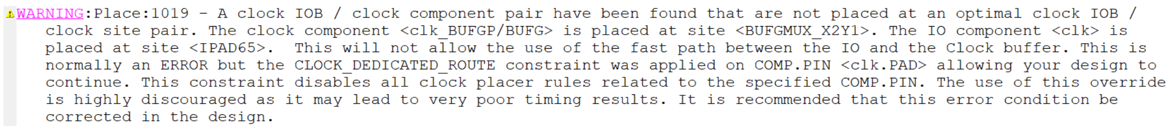


```
ERROR:Place:1018 - A clock IOB / clock component pair have been found that are not placed at an optimal clock IOB /
   clock site pair. The clock component <clk_BUFGP/BUFG> is placed at site <BUFGMUX_X2Y1>. The IO component <clk> is
   placed at site <IPAD65>.  This will not allow the use of the fast path between the IO and the Clock buffer. If this
   sub optimal condition is acceptable for this design, you may use the CLOCK_DEDICATED_ROUTE constraint in the .ucf
   file to demote this message to a WARNING and allow your design to continue. However, the use of this override is
   highly discouraged as it may lead to very poor timing results. It is recommended that this error condition be
   corrected in the design. A list of all the COMP.PINs used in this clock placement rule is listed below. These
   examples can be used directly in the .ucf file to override this clock rule.
   < NET "clk" CLOCK_DEDICATED_ROUTE = FALSE; >
```

This is because the button is not connected to the GCLK global clock line, which the program will detect during implementation. Therefore, in the **.ucf* file, a fragment code should be added that will skip checking this rule for the ***clk*** line.

```
NET "clk" LOC = P80   | PULLUP   | IOSTANDARD = LVCMOS33 | SLEW = SLOW
NET "clk" CLOCK_DEDICATED_ROUTE = FALSE;
```

The fragment that should be added to the description of the clock line has been marked in green. After such a procedure, instead of an error during implementation, only a warning will appear.



```
WARNING:Place:1019 - A clock IOB / clock component pair have been found that are not placed at an optimal clock IOB /
   clock site pair. The clock component <clk_BUFGP/BUFG> is placed at site <BUFGMUX_X2Y1>. The IO component <clk> is
   placed at site <IPAD65>.  This will not allow the use of the fast path between the IO and the Clock buffer. This is
   normally an ERROR but the CLOCK_DEDICATED_ROUTE constraint was applied on COMP.PIN <clk.PAD> allowing your design to
   continue. This constraint disables all clock placer rules related to the specified COMP.PIN. The use of this override
   is highly discouraged as it may lead to very poor timing results. It is recommended that this error condition be
   corrected in the design.
```

Please note that for fast clocks we only use dedicated lines, i.e. GCLK.

After generating the *u_d_cnt.bit* configuration file, you must program the target system using the ElbertV2Config application, selecting the appropriate COM serial port beforehand.

When testing the project on the Elbert V2 board, keep in mind that the SW1 - SW3 buttons short the FPGA input pins to ground. By default, they have a logical 1 state (internal pull-up resistors are on).

You may notice that pressing SW1 often jumps several states forward. This is caused by vibration of the mechanical contacts. So we will convert our project to a form in which the source of the clock signal will be a crystal oscillator on a printed circuit board. Its fundamental frequency will be divided so as to obtain a clock with a frequency of 1 Hz inside the architecture.

- ***Implementation of a state machine with a crystal oscillator as the clock source***

Inside the architecture, before the *begin* keyword, we define an additional signal called *clk_1Hz***.**

```
signal clk_1Hz : std_logic := '0';
```

In the previous architecture description, we change the *clk* clock to *clk_1Hz* in the sensitivity list.

```
process (clk_1Hz, res)
```

At the beginning of the architecture description (after the *begin* keyword), we add a process in which we only specify the *clk* signal in the sensitivity list. The task of this process will be to generate a clock with a frequency of 1 Hz.

```
process(clk)
variable counter : integer:=0;
begin
if rising_edge(clk) then
      if counter < 6000000 then
            counter := counter+1;
      else
            counter := 0;
            clk_1Hz <= not clk_1Hz;
      end if;
end if;
end process;
```

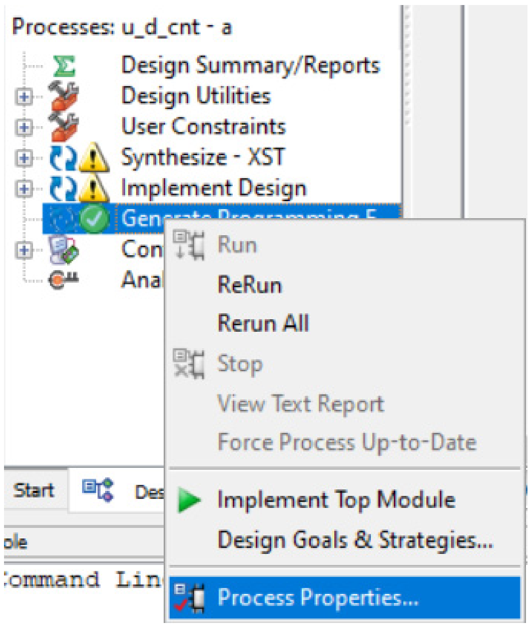
In the constraints file (*.ucf) the line responsible for the clk clock should be changed. This time we indicate the FPGA pin connected to the 12 MHz crystal oscillator. We also remove the line that ignores pin type checking.

```
NET "clk"   LOC = P129  | IOSTANDARD = LVCMOS33 | PERIOD = 12MHz;
```

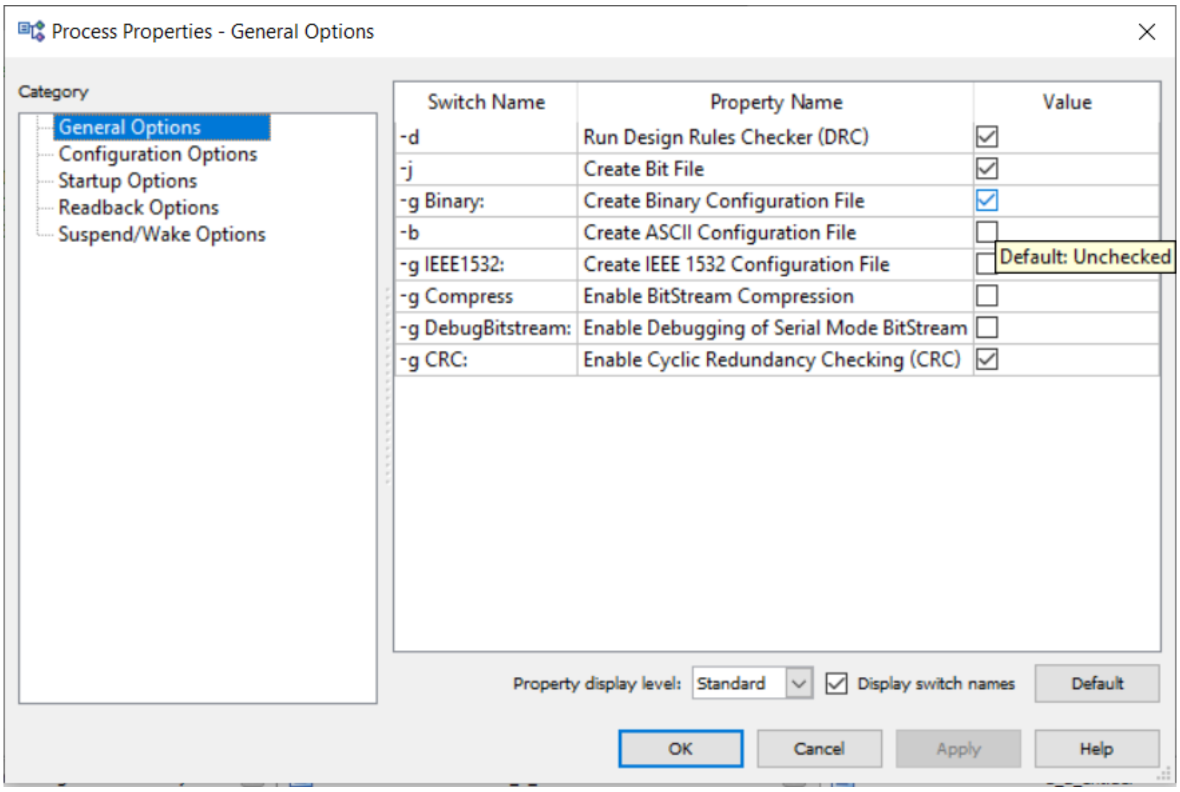
The functions of the SW2 and SW3 buttons remain unchanged.

In case of problems with errors resulting from editing the original project, use the source files named *u_d_cnt_clk_1Hz* that are included with the manual.

The *ElbertV2Config* program, which is used to configure the FPGA, works with various types of files generated by the Webpack ISE. So far, we've only used bit files. We will modify the WebPack settings so that it is possible to generate bin files in addition to bit files. In the Processes window, right-click on the Generate Programming File option, and then select the Process Properties... option from the drop-down context menu.



In the dialog box that appears, select the *Create Binary Configuration File* option.



Switch Name	Property Name	Value
-d	Run Design Rules Checker (DRC)	☑
-j	Create Bit File	☑
-g Binary:	Create Binary Configuration File	☑
-b	Create ASCII Configuration File	☐
-g IEEE1532:	Create IEEE 1532 Configuration File	☐
-g Compress	Enable BitStream Compression	☐
-g DebugBitstream:	Enable Debugging of Serial Mode BitStream	☐
-g CRC:	Enable Cyclic Redundancy Checking (CRC)	☑

After confirming the changes, we generate the files for programming the FPGA again. In the *ElbertV2Config*, this time we indicate the **.bin* file and program (configure) the target system.

In the presented example, the state machine was defined using the state signal and constants named S0, S2, S4, S6, S8, S10, S11, S12, S13, S15 and S15 with defined values. You can achieve the same effect by defining a new variable type with its values explicitly listed. In addition to numbers, these values can also be strings.

2. State machine based on defined type with enumarated values.

We define a signal and assign a new data type to it as follows:

```
type STATE_TYPE is (s0, s2, s4, s6, s8, s10, s11, s12, s13, s14, s15);
signal state   : STATE_TYPE;
```

Each state of the state signal must be assigned an appropriate value at the output **q**. For this purpose, we will use an additional process whose sensitivity list will only include the state signal. The processes responsible for the state machine are presented in the following listings.

```
process (clk_1Hz, res) begin
  if res = '0' then state <= S0;
      elsif rising_edge(clk_1Hz) then
             if u_d = '1' then
              case state is
                   when S0 => state <= S2;
                   when S2 => state <= S4;
                   when S4 => state <= S6;
                   when S6 => state <= S8;
                   when S8 => state <= S10;
                   when S10 => state <= S11;
                   when S11 => state <= S12;
                   when S12 => state <= S13;
                   when S13 => state <= S14;
                   when S14 => state <= S15;
                   when others => state <= S0;
              end case;
             elsif u_d = '0' then
              case state is
                   when S0 => state <= S15;
                   when S2 => state <= S0;
                   when S4 => state <= S2;
                   when S6 => state <= S4;
                   when S8 => state <= S6;
                   when S10 => state <= S8;
                   when S11 => state <= S10;
                   when S12 => state <= S11;
                   when S13 => state <= S12;
                   when S14 => state <= S13;
                   when S15 => state <= S14;
                   when others => state <= S0;
              end case;
             end if;
      end if;
end process;
```

```
process (state)
begin
   case state is
      when S0 => q <= "0000";
      when S2 => q <= "0010";
      when S4 => q <= "0100";
      when S6 => q <= "0110";
      when S8 => q <= "1000";
      when S10 => q <= "1010";
      when S11 => q <= "1011";
      when S12 => q <= "1100";
      when S13 => q <= "1101";
      when S14 => q <= "1110";
      when S15 => q <= "1111";
      when others => q <= "0000";
   end case;
end process;
```

An important feature of the VHDL language and its interpreters is that line breaks do not affect its interpretation and synthesis. For example, writing when S0 => q <= "0000"; is interpreted in the same way as:

```
when S0 =>
   q <= "0000";
```

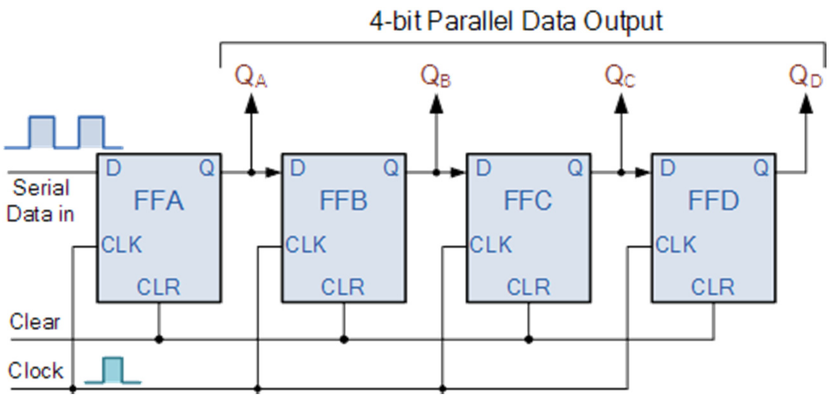
After synthesizing the project and generating the configuration file, it is necessary to check the correctness of its operation..

II. Shift register

Shift registers are used to store and move data in digital systems, or to convert data from serial to parallel and vice versa. They are created by means of series-connected flip-flops, most often type D. The length of the register in bits corresponds to the number of flip-flops in this register.

- ***Implementation of a 4-bit shift register with serial and parallel output.***

An example 4-bit shift register with serial input and parallel output is shown in the figure below.



If we wanted to use such a shift register as SISO (with serial input and serial output), then the output of the register would be the output of the last flip-flop, i.e. QD.

An example of a shift register implementation is presented in the following code snippet in VHDL.

```vhdl
process (clk_1Hz)
begin
  if (clk_1Hz'event and clk_1Hz = '1') then
             s_reg(2 downto 0) <= s_reg(3 downto 1);
             s_reg(3) <= din;
  end if;
end process;
```

On each rising edge of the clk_1Hz clock, the 3 oldest bits are shifted 1 position to the right (`s_reg(2 downto 0) <= s_reg(3 downto 1))` with the simultaneous assignment of the bit from the *din* input to the highest bit in the register. Instruction `led <= s_reg` assigns of states from the parallel register output to the output pins of the FPGA connected to the LEDs (D5 to D8). Instruction `q <= s_reg(0)` assigns the serial output of the register to the LED (D1). The source code files *shift_reg.vhd* and *shift_reg.ucf* are provided with the tutorial.

TASKS:

1. Create test vectors for the shift register design, then simulate its operation using the ISim tool in ISE Webpack.

2. Based on the examples you worked through in this exercise, create a shift register design that will operate as a state machine.

References

- P. Zbysiński, J. Pasierbiński – *Układy programowalne – pierwsze kroki.* Wydawnictwo BTC, Warszawa 2004

- J. Majewski, P. Zbysiński – *Układy FPGA w przykładach.* Wydawnictwo BTC, Legionowo 2007.

- Electronics Tutorial – *Sequential Logic: The Shift Register* https://www.electronics-tutorials.ws/sequential/seq_5.html