Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Universidade Federal de Santa Catarina
Florianópolis, Brazil

August 2013

Contents

Chapter 7. Synthesis of Finite State Machines

A Finite State Machine (FSM) is a powerful tool used to build control circuits. Control circuits are employed in situations where a sequence of operations should be performed, according to a pre-defined execution order. The use of a FSM as a control circuit is better explored in the next chapters. This chapter introduces the basic concepts related to the design of FSMs. At the end of the chapter, the students should be able:

- to understand the concept of FSMs;
- to understand the synthesis process of FSMs in VHDL;
- to understand the concept of counters implemented in VHDL using FSMs;
- to design and to implement in VHDL an FSM based counter.

7.1 Finite State Machines

As already discussed in Chapter 1, a computer system usually consists of a control module (i.e, control unit) and an operations execution module (i.e., datapath), as shown in Figure 1.1. The control module is responsible for coordinating the sequence of activities to be performed by a given system. In digital systems, control signals are generated by sequential circuits. A sequential circuit passes through a series of states, and each state (every time), may provide a certain output. The outputs of each state are the signals used to control the sequence of activities executed by a target system.

In Figure 7.1, a state diagram is used to model an FSM. The state diagram is represented by a directed graph, where the vertices (or nodes) are the FSM states, and the edges (or arcs) represent the transitions from one state to another. As shown in Figure 7.1, an FSM has the following components:

- **States** are represented in the graph by nodes S_0, S_1, S_2 and S_3. The states are responsible, basically, for storing information regarding past changes in the inputs. In the example FSM, the triangle pointing to S_0 indicates that this is the initial state.
- **Transitions** are represented in the graph by the edges. A transition indicates a state change by means of a condition that enables the switch from a state to another.
- **Actions** are activities to be performed when the respective state is reached.

A hardware implementation of an FSM comprises a combinational circuit, and a register. As shown in Figure 7.2, the FSM's current state is stored in the register. The combinational circuit uses the circuit's inputs and the stored "current state", in order to calculate the "next state". The combinational circuit calculates also the FSM's output.

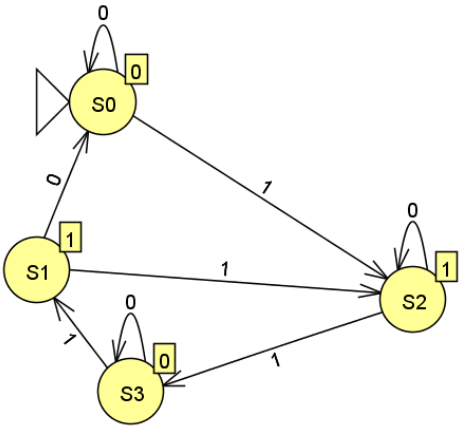

Figure 7.1. Example of a state diagram for an FSM.

An FSM can also be represented through a state transition table. In Table 7.1, it can be observed that when the FSM is in state S_0, a transition (change) to state S_2, takes place only when the circuit's input is '1' ("Inputs" in Figure 7.2). While the circuit's input stays in '0', the FSM will remain in the same state S_0, as shown on the first row of Table 7.1. This action is also shown graphically in the loop with the '0' input in state S_0 in Figure 7.1. The fourth column in Table 7.1 shows the output provided by the FSM for each state. In Figure 7.1, the output for each state is represented by the numeric value in a square at the top right hand corner of the nodes. In Figure 7.2, the output is represented by the signal "Outputs".

The state transition table (Table 7.1) is a direct representation of the state diagram of an FSM.

Table 7.1. State transition table for the FSM shown in Figure 7.1.

Current State	Input	Next State	Output
S_0	0	S_0	0
S_0	1	S_2	0
S_1	0	S_0	1
S_1	1	S_2	1
S_2	0	S_2	1
S_2	1	S_3	1
S_3	0	S_3	0
S_3	1	S_1	0

Considering that the circuit in Figure 7.2 implements the FSM shown in Figure 7.1, the register stores the FSM current state (Table 7.1 first column), and the combinational circuit calculates the "next state" (Table 7.1 third column) using the stored "current state" and the circuit's inputs (Table 7.1 second column). The combinational circuit calculates also the FSM's output (Table 7.1 fourth column).

A possible VHDL implementation for the FSM is shown in Figure 7.3. In that implementation, a transition happens each time an event of a rising clock edge takes place. This event is represented in Figure 7.1 by the graph's edges, and in Figure 7.2 by the *clock* input.

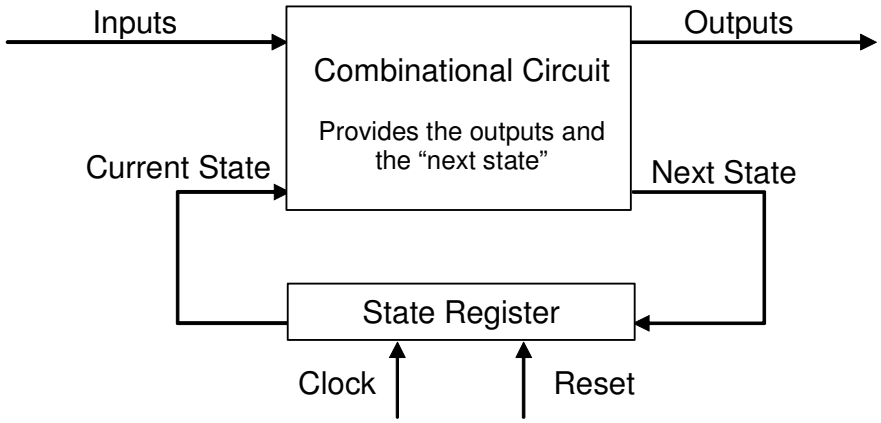

Figure 7.2. Block diagram of an FSM hardware implementation.

7.2 VHDL Synthesis of Finite State Machines

An FSM is, essentially, a sequential circuit. Basically, in an FSM, from the combination of a given sequence of inputs and past states, the next states, and the respective outputs are generated sequentially.

As aforementioned in Chapter 6, a sequential circuit can be implemented in VHDL using the *process* statement. There are several ways to describe an FSM in VHDL, using 1, 2 or even 3 processes.

The FSM studied in Figure 7.1 has been implemented in VHDL using the two processes strategy. The source code for this implementation is shown in Figure 7.3. An important VHDL concept useful for FSMs description is the enumeration type. With the VHDL reserved word *type* the developer can create a new type. In Figure 7.3, line 7, STATES is a new type that can hold the values S0, S1, S2 and S3. On line 8, CS (i.e., current state) and NS (i.e., next state) are two internal signals defined as STATES type. This means that these two signals can be assigned only to other signals or constants of the same type STATES. It is also important to highlight that S0, S1, S2 and S3 are not part of the VHDL language. These are just labels used in the set of possible values to be used in the new enumeration type. During the synthesis process, these labels will be translated into ones and zeros by the EDA tool, in order to generate the final hardware implementation for the abstract VHDL description. The VHDL syntax for the enumeration type is as follows:

type identifier **is** (element1, element2, element3, …);

Where, "type" and "is" are VHDL reserved words, and "identifier" is the name of the new type that is being created. Following the "is" statement, there is an ordered set of values to be used in the VHDL description as the values to be assigned to the signals of type "identifier".

In Figure 7.3, line 13, an element belonging to the set of enumeration values is directly assigned to the CS signal. In Figure 7.3, line 15, there is an example of assignment of a signal of type STATES to another signal of the same type.

```
1  entity MOORE is
2      port(X, clock, reset: in std_logic;
3           Z              : out std_logic);
4  end;
5
6  architecture TwoProcesses of MOORE is
7      type STATES is (S0, S1, S2, S3);     VHDL Enumeration Type
8      signal CS, NS : STATES;
9  begin
10     process (clock, reset)                1st process: implements a register
11     begin                                 that stores the current state (CS)
12         if reset= '1' then                as a function of the next state (NS)
13             CS  <= S0;
14         elsif clock'event and clock='1' then
15             CS <= NS;
16         end if;
17     end process;
18                                           2nd process: implements the circuit
19     process(CS, X)                        that generates the Z output,
20     begin                                 according to the current state (CS)
21         case CS is                        and the X input
22             when S0 =>  Z <= '0';
23                         if X='0' then NS <= S0; else NS <= S2; end if;
24             when S1 =>  Z <= '1';
25                         if X='0' then NS <= S0; else NS <= S2; end if;
26             when S2 =>  Z <= '1';
27                         if X='0' then NS <= S2; else NS <= S3; end if;
28             when S3 =>  Z <= '0';
29                         if X='0' then NS <= S3; else NS <= S1; end if;
30         end case;
31     end process;
32 end TwoProcesses;
```

Figure 7.3. VHDL implementation of an FSM using two processes.

The VHDL synthesis, using two processes, of the FSM described in Figure 7.1 is straightforward. The process P1 defines the current state (CS) for the processes P2. Process P1 is sensitive to the clock and reset signals. If an asynchronous reset (reset='0') occurs, CS receives the S0 value, otherwise, if a rising clock edge occurs CS is updated with the next state (NS) signal, that is, it is responsible to perform the state transition process. The process P2 updates the next state (NS) signal and defines new values for the output Z signal according to the current state.

In both representations, the graphical (Figure 7.1) and the textual (Figure 7.3) one, it is possible to notice that the circuit's outputs are defined in each of the states. As listed in Table 7.1, for states S0 and S3, the output is '0', and for states S1 and S2, the output is '1'. This type of FSM is known as a Moore machine, as the output values depend only on the current state. Another type of FSM is known as a Mealy machine, where the outputs are defined according not only to the current state, but also to the current inputs.

Using the enumeration type in VHDL designs is important also in the simulation step. As shown in Figure 7.4, the output values (Z) are well identified for each CS state.

All the transitions between states are also well identified, as the names S0, S1, S2 and S3 are a good help for the simulation visualization. In case the enumeration type had not been used in VHDL code, the CS and NS signals would have shown “00”, “01”, “10” and “11” instead, making it a bit more difficult to visualize all the transitions. In FSMs with more states, the situation would be even more complicated.

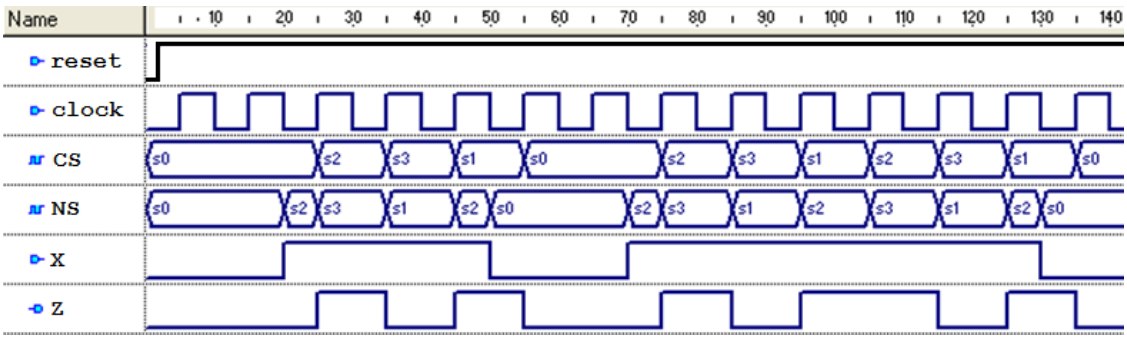

Figure 7.4. Waveform diagram for the example FSM.

Another approach to describe an FSM in VHDL employs three processes instead of two. As shown in Figure 7.5(a), the P1 process updates the current state (CS), and sends this information to P2 and P3. In Figure 7.5(b), P2 computes the next state, but does not perform the transition. The next state (NS) information is sent from P2 to P1, which is the responsible for updating the current state (CS). P3 is the process responsible for signals (outputs) assignments and updates.

In Figure 7.6 there is a VHDL implementation for a three processes case study. The P1 process is triggered by the clock falling edge (i.e. clk’event and clk = ‘0’), performing the state transitions (line 7, CS <= NS). P2 process is triggered by CS and the X input signal. According to the input (X), P2 defines the next state of the FSM, which will be assigned to CS by P1 in the next clock falling edge. The P3 process is triggered by the rising clock edge (i.e. clk’event and clk = ‘1’), and it is responsible for providing the FSM outputs, according to the current state (CS).

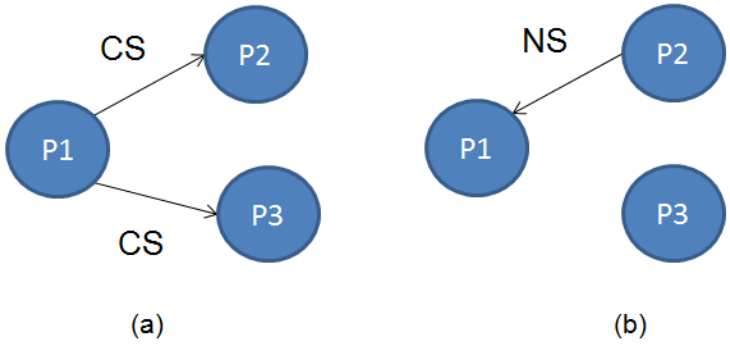

Figure 7.5. Three processes FSM diagram. (a) P1 process updates the current state; (b) P2 process defines the next state.

```
1  P1: process(clk)
2  begin
3     if falling_edge(clk) then
4        if rst = '0' then
5           CS <= S0;
6        else
7           CS <= NS;
8        end if;
9     end if;
10 end process;
11
12 P3: process(clk)
13 begin
14    if rising_edge(clk) then
15       case CS is
16          when S0 =>
17             Z <= '0'
18          when S1 =>
19             Z <= '0'
20          when S2 =>
21             Z <= '1';
22          when others =>
23       end case;
24    end if;
25 end process;
```

```
P2: process(CS, X)
begin
   case CS is
      when S0 =>
         NS <= S1;
      when S1 =>
         if X = '1' then
            NS <= S2;
         else
            NS <= S1;
         end if;
      when S2 =>
          NS <= S1;
      when others =>
   end case;
end process;
```

Figure 7.6. A VHDL implementation for a three processes FSM.

In the two processes FSM, one of the processes is used to model the state register, defining the next state, and the other process updates the next state and provides the FSM outputs.

In the three processes FSM, one process models the state register defining the next state, a second process updates the next state, and a third process provides the FSM output.

In a single process FSM implementation, all these activities are performed by the same process. Figure 7.7 shows a VHDL implementation for the Figure 7.1 FSM, using only one process. In this case, when the reset signal is '1', the initial state is set to S0. When the reset signal is '0', and there is a rising clock edge event, then the Z output is provided according to the current state (CS), and the next state (NS) is defined according to the X input.

```
1  entity MOORE is
2     port(X, clock, reset: in std_logic;
3          Z              : out std_logic);
4  end;
5
6  architecture OneProcess of MOORE is
7     type STATES is (S0, S1, S2, S3);
8     signal CS: STATES;
9  begin
10    process(clock, reset)
11    begin
12       if reset= '1' then
13          CS  <= S0;
14       elsif clock'event and clock='1' then
15          case CS is
16             when S0 => Z <= '0';
17                        if X='0' then CS <= S0; else CS <= S2; end if;
18             when S1 => Z <= '1';
19                        if X='0' then CS <= S0; else CS <= S2; end if;
20             when S2 => Z <= '1';
21                        if X='0' then CS <= S2; else CS <= S3; end if;
22             when S3 => Z <= '0';
23                        if X='0' then CS <= S3; else CS <= S1; end if;
24          end case;
25       end if;
26    end process;
27 end OneProcess;
```

Figure 7.7. VHDL implementation of an FSM using one process.

In the two and the three processes approaches, one of the processes provides combinatorial outputs, as there is no clock signal on its sensitivity list. This can be an interesting solution, in case the designer needs to have combinatorial outputs. In the one process implementation, there is no combinatorial output. On the other hand, the one process approach, historically, is a better choice regarding faster simulation times, and also it is easier to debug as there is no combinatorial processes involved. However, designers usually find the two processes approach more readable, as it is an appropriate behavioral description of the FSM classical hardware architecture shown in Figure 7.2.

7.3 FSM Case Study: Designing a Counter

A counter is a good example of a sequential circuit, and it has interesting features suitable for FSM modeling. A counter has an output that provides expected values for each of its states, according to the input clock pulses. The output values are provided in a sequence that can be ascending, descending, or out of order. In any case, when the last value of the counting is reached, the sequence can be restarted, according to the design requirements. There are several types of counters, and among the most used ones there are the up counters, and the down counters. In Figure 7.8 there is the VHDL description of a ring counter, performing an up counting.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity ring_counter is
5     port (CLK    : in std_logic;
6           RST    : in std_logic;
7           Z      : out std_logic_vector(7 downto 0)
8           );
9  end ring_counter;
10
11 architecture beh of ring_counter is
12    signal aux : std_logic_vector(7 downto 0);
13 begin
14    process(CLK, RST)
15    begin
16       if (RST = '0') then
17          aux <= "00000001";
18       elsif (CLK'event and CLK = '1') then
19          aux(7 downto 1) <= aux(6 downto 0);
20          aux(0) <= aux(7);
21       end if;
22    end process;
23    Z <= aux;
24 end beh;
```

Figure 7.8. A ring counter described in VHDL.

The ring counter is initialized on line 17, and at each rising clock edge, the counter value is rotated to the left. On line 20, the most significant bit is copied to the less significant bit position, closing the “ring”.

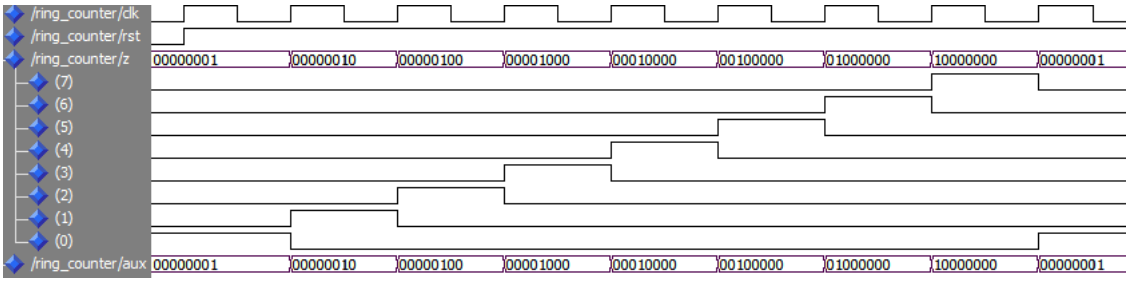

Figure 7.9. Simulation results for the ring counter.

The simulation results for this counter are shown in Figure 7.9. After the reset (RST) signal goes up, the ring counter is load with the start value “00000001”. The bit in

the less significant position holds a '1', which is rotated passing through all the eight bit positions, and going back to the original position after eight clock pulses. The pattern provided on the Z output by this ring counter is also known as the "one hot" code. As shown in Figure 7.9, at each clock pulse only one of the eight output bits is '1' ("one hot"). This is a circular up counter as the provided Z outputs, in decimal notation, are: 1, 2, 4, 8, 16, 32, 64, 128, 1, 2, 4, …

Another example of counter is shown in Figure 7.10. In this example, a counter is used to generate a delay. This sort of circuit is very useful, for instance, in applications where data need to be shown in a display. In Figure 6.14, the calculation result stored in the registers is shown in the 7-segment displays each time the KEY(1) push-button (Enter) is pressed. A similar application may require the stored data to be shown on the displays periodically as, for example, every one second.

```
process(CLK, RST)       -- Considering a 27MHz clock, the process is triggered
begin                   -- 27 million times per second.
   if RST = '0' then
     CS  <= S0;
   elsif CLK'event and CLK = '1' then
     case CS is
       when S1 =>   ...              -- Do something (e.g. write to LCD, ...).
       when D1 =>                    -- This state starts the delay counting (1s).
         delay <= ( others => '0' );
         CS <= D2;
       when D2 =>                    -- State that generates the chosen delay
         delay <= delay + 1;         -- "delay" was set to zero in D1.
         CS <= D3;
       when D3 =>              -- Tests if the final counting has been reached
         if delay >= x"800000" then  -- 8,388,608 / 27,000,000 = 0.3 * 3 = 1 s.
           CS <= S1;  -- When the upper limit is reached, leaves the delay
         else         -- loop, and goes back to the application.
           CS <= D4;  -- Remains in the counting loop to generate the delay.
         end if;
       when D4 =>     -- Continues the delay counting.
         CS <= D2;
     end case;
   end if;
end process;
```

Figure 7.10. A counter used for delay generation.

The FSM in Figure 7.10 is used to generate a 1 s delay, considering an input clock of 27 MHz. The S1 state is included just to indicate the place where the 7-segment code could be inserted. Usually, after the data are sent to the display (in S1, S2, …, states), the FSM flow is diverged to the D1 state, which is the initial state of the delay routine. In this state, the delay counter is set to an initial value (zero in this example). Next, in the following rising clock edge, the counter (delay) is incremented (D2 state). In the next state (D3), there is a test inorder to check if the final value for the counting has been reached. If the test results in true, then 1 s has passed, and the FSM diverges back to the application (S1 state). Otherwise, if the test results in false, the FSM needs to keep counting on

each rising clock edge, and the FSM goes to the D4 state. In this example, the final test value was calculated considering the 27 MHz input clock (i.e. 27 million pulses per second), and the FSM has to count 8,388,608 times in order to waste, approximately 1/3 s of time. As shown in Figure 7.10, the FSM needs three clock pulses to perform a complete round, i.e., increment in D2, test in D3, and do nothing in D4. The D4 state has been included in this FSM with the only goal of spending one extra clock pulse. Therefore, 3 clock pulses times 1/3 s is equal to the required 1 s delay.

7.4 Laboratory Assignment

In previous chapters, several components have been glued together in order to build a basic calculator. In the next chapters, an FSM will be used in order to control the calculator operations.

In this chapter, a counter is used as a case study. Counters represent an important type of digital circuits, and FSMs can be used to model their behavior and functionality.

The laboratory objectives are:

- to design an FSM based digital circuit;
- to design a counter in VHDL.

Laboratory Session

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

- Design and implement in VHDL an one process FSM, for the generation of the ASCII characters ‘A’ to ‘Z’;
- The characters should be presented on the green LEDs (binary format) and on two 7-segments displays (hexadecimal format), as shown in Figure 7.11;
- To write in the 7-segment displays, it is necessary to use two instances of the *decod7seg.vhd* component, described in Chapter 5 (see Figure 5.22);
- The FSM has an asynchronous reset, KEY(0) button, used to initialize a counter with the first character in the sequente (‘A’ = 41H);
- On the rising clock edge (27 MHz), the counter should be incremented, generating the next ASCII table character (‘B’, ‘C’, ‘D’, …);
- The FSM should have a small number of states, but sufficient to increment the counter and check whether the final counting (’Z’ = 5AH) has been reached;
- When the final counting is reached, the FSM should be reset to the initial state (beginning of the counting);
- In order to activate the DE2 27 MHz clock signal, the TD_RESET output has to be set to ‘1’;
- In Figure 7.12 it is shown a suggestion of VHDL description for the top component, without the 7-segment decoders. In order to include the 7-segment decoders, create an F signal in the top file to connect the FSM component to the 7-segment decoders and the LEDG output.

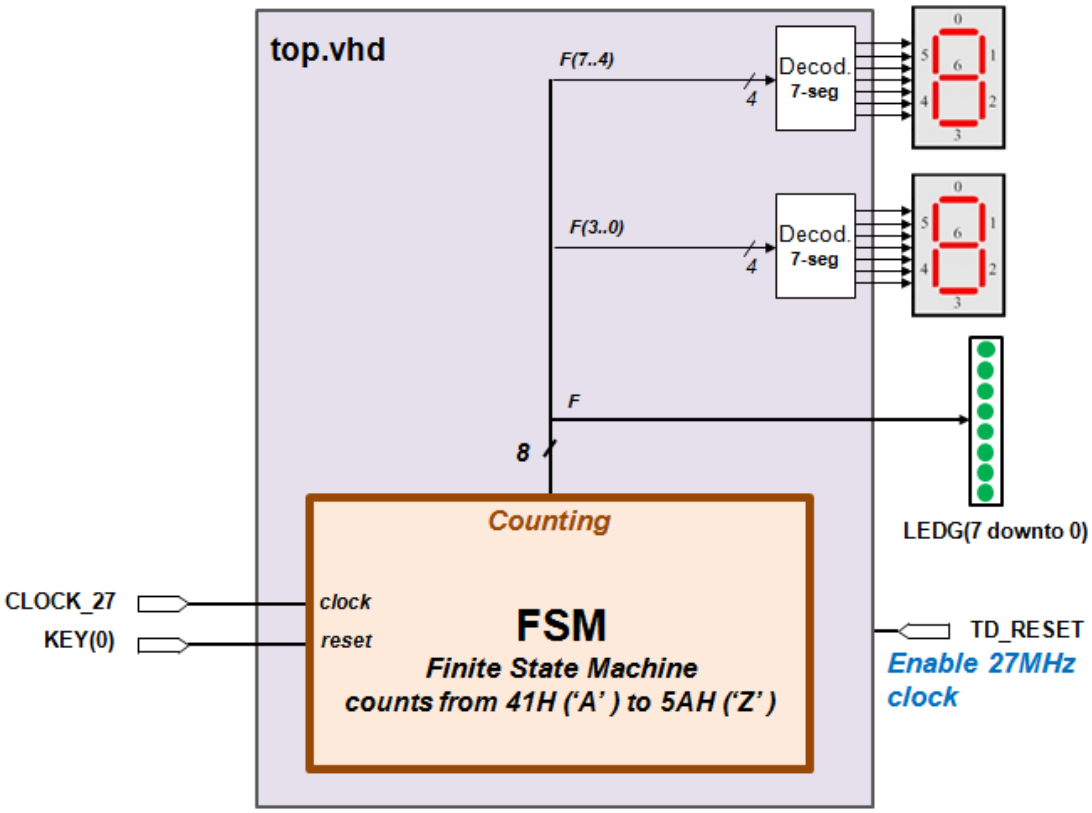

Figure 7.11. 'A' to 'Z' ASCII counter block diagram.

```
entity Top is
    port (  LEDG: out std_logic_vector(7 downto 0);
            KEY: in std_logic_vector(3 downto 0);
            TD_RESET: out std_logic;
            CLOCK_27: in std_logic
          );
end Top;
architecture top_beh of Top is
   component ASCII_counter  -- This is the FSM component
      port (
              ASCIIvalue: out std_logic_vector(7 downto 0);
              clock: in std_logic;
              reset: in std_logic
      );
begin
   TD_RESET <= '1';

    L0: ASCII_counter port map ( LEDG, CLOCK_27, KEY(0) );

end top_beh;
```

TD_RESET should be '1' in order to switch on the DE2 CLOCK_27 signal

Figure 7.12. Suggestion of top file for the ASCII counter design without the 7-segment decoders.

Note: Alternatively, instead of the 27 MHz clock, the rising clock edge can be generated by a push button (KEY). **Warning**! Beware of the debounce problem, since a single press in a mechanical switch may result in multiple presses in a digital circuit. A clock signal is a much more precise way to control the counting sequence.

In this laboratory session, all the usual activities should be performed: project creation; VHDL synthesis; VHDL simulation; and FPGA prototyping.

For the FPGA prototyping, it is important to include in the FSM some sort of the delay procedure as, for instance, the one shown in Figure 7.10. Without a delay routine, the user will not be able to watch the counting presented on the LEDs and displays. However, for the simulation, the delay routine should be set to a smaller final counting value, otherwise the process would take too long. For instance, a 20 seconds simulation time in an i7 quad core processor (hyper threading, so 8 cores), running at 2.93 GHz and with 8 GB of RAM, took 19 hours and 28 minutes to be performed.

Figure 7.13 shows the ASCII counter simulation waveform. In the CS.Init state, after the KEY(0) button is released, the initial counting value is set to 41H, which is the 'A' character in the ASCII table (see LEDG). In the CS.Test state, the FSM checks if the upper counting limit 5AH ('Z') has been reached. Next, the delay states are performed, starting by the CS.D1 state, where the delay counting signal is initialized. The CS.D2, CS.D3, and CS.D4 states are performed 8,388,608, as discussed before in Figure 7.10.

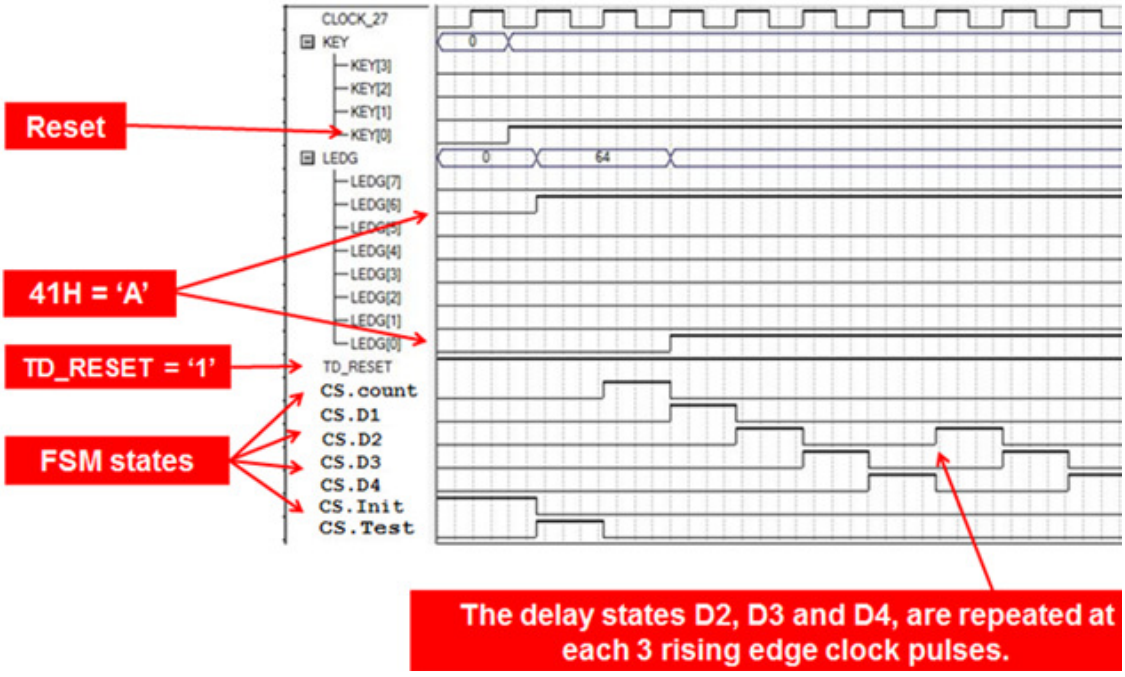

Figure 7.13. Simulation results showing the delay states of the FSM.