Politecnico di Torino

DIGITAL SYSTEMS ELECTRONICS

04OIHNX - A.A. 2024/2025

Prof. G. Masera

Laboratory assignment no. 5 – FSMs

DUE DATE: 15/04/2025 DELIVERY DATE: 14/04/2025

GROUP 08 - Contributions:

- Daniele Becchero (308299) – 33.3%
- Bohotici Ionut Viorel (300061) – 33.3%
- Simone Viola (310779) – 33.3%

Introduction

The purpose of this laboratory is to explore the operation of Finite State Machines (FSMs) and their implementation using VHDL. The study includes verifying the correctness of the design through simulation in ModelSim, analysing the resulting circuit using RTL Viewer in Quartus Prime, and testing the practical functionality of the FSM on the Altera DE1 board. This report is divided into four sections, each focusing on a different FSM implementation. The sections progress in order of increasing complexity and demonstrate various design approaches.

Section 1: One-hot FSM

This section outlines the VHDL implementation of a finite state machine (FSM) designed to recognize two specific patterns in the input. The circuit activates an LED whenever the preceding four input bits are identical, whether all are '0' or all are '1'. The state diagram of this Moore-type FSM is the one shown in Figure 1.

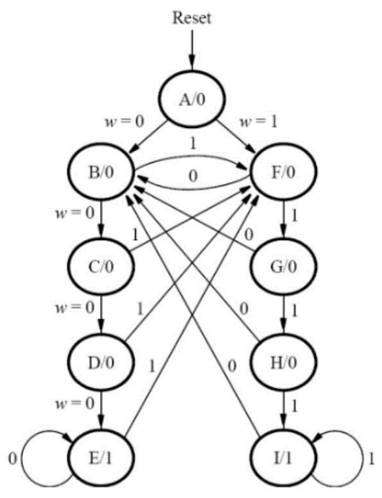


Figure 1 - State diagram for "Pattern recognizer FSM"

In this section, the FSM has been described in VHDL by using the "one-hot" style.

Delivered files

The delivered files for this exercise are three:

- *FSM.vhd*, VHDL code that implements the FSM with one-hot style
- *FSM_tb.vhd*, VHDL code to validate the FSM by a simulation
- *Flipflop.vhd*, VHDL code for the D-type flip-flop

An additional VHDL file have been used in the synthesis phase to physically connect the FSM to the board's devices. The *.sof* file is ready to be used to program the FPGA.

Design entry

The "one-hot" implementation of the FSM involves using one flip-flop for each state in the state diagram and driving each flip-flop by a combinational circuit that can be derived by inspection.

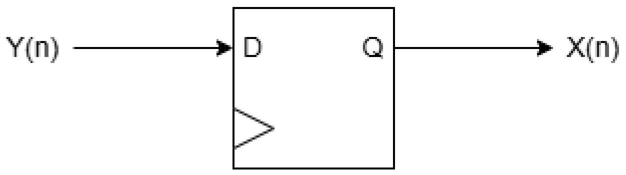


Figure 2 - Single flip-flop arrangement

Each flip-flop input's combinational circuit is described in VHDL as shown. The one-hot encoding table is "Table 1" in the assignment.

```vhdl
Y(0) <= not (RESN);
Y(1) <= not(W) and (X(0) or X(5) or X(6) or X(7) or X(8)) and RESN;
Y(2) <= not(W) and X(1) and RESN;
Y(3) <= not(W) and X(2) and RESN;
Y(4) <= not(W) and (X(3) or X(4)) and RESN;
Y(5) <= W and (X(0) or X(1) or X(2) or X(3) or X(4)) and RESN;
Y(6) <= W and X(5) and RESN;
Y(7) <= W and X(6) and RESN;
Y(8) <= W and (X(7) or X(8)) and RESN;
```

The FSM's output, which is displayed on the red LED named *LED(0)*, is activated when the machine reaches the final states corresponding to the recognition of a sequence (state E for four consecutive zeros and state I for four consecutive ones). Hence, the output combinational circuit is trivial. It has been described as shown.

```vhdl
Z <= X(4) or X(8);
```

Functional simulation

The simulation of this FSM involves sequentially toggling the clock signal at 1 ns interval and modifying the input `w_tb` (the test bench version of the input signal `w`). To ensure the correct behaviour across all possible state transitions, the test bench code is designed to test three different paths through the FSM states, as outlined in Table 1.

Path test	State 1	State 2	State 3	State 4	State 5
#1 (`w` = 0)	A	B	C	D	E
#2 (`w` = 1)	A	F	G	H	I
#3 (mixed)	A	B	F	G	B

Table 1 - Tested combinations for two-processes FSM

The waveform result is shown in the figures below.

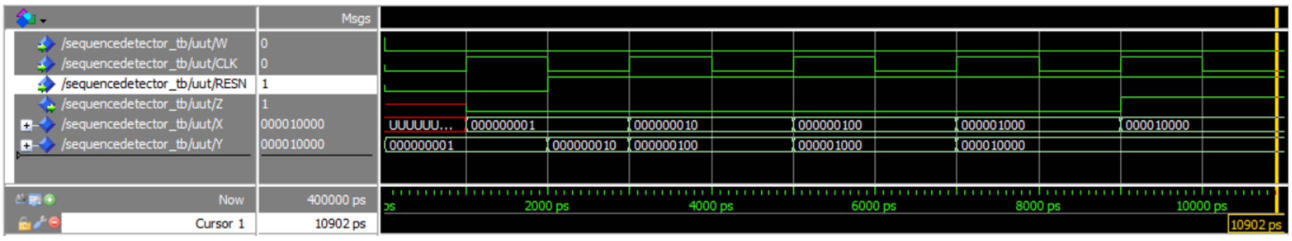

Msgs
/sequencedetector_tb/uut/W 0
/sequencedetector_tb/uut/CLK 0
/sequencedetector_tb/uut/RESN 1
/sequencedetector_tb/uut/Z 1
/sequencedetector_tb/uut/X 000010000
/sequencedetector_tb/uut/Y 000010000
UUUUUU... 000000001 000000010 000000100 000001000 000010000
000000001 000000010 000000100 000001000 000010000
Now 400000 ps
Cursor 1 10902 ps
ps 2000 ps 4000 ps 6000 ps 8000 ps 10000 ps
10902 ps

Figure 3 - ModelSim waveforms of Path test #1, w = 0

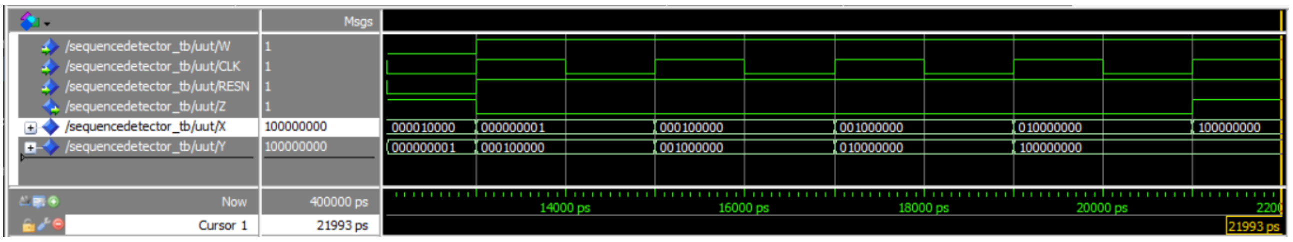

Msgs
/sequencedetector_tb/uut/W 1
/sequencedetector_tb/uut/CLK 1
/sequencedetector_tb/uut/RESN 1
/sequencedetector_tb/uut/Z 1
/sequencedetector_tb/uut/X 100000000
/sequencedetector_tb/uut/Y 100000000
000010000 000000001 000100000 001000000 010000000 100000000
000000001 000100000 001000000 010000000 100000000
Now 400000 ps
Cursor 1 21993 ps
14000 ps
16000 ps
18000 ps
20000 ps
21993 ps

Figure 4 - ModelSim waveforms of Path test #2, w = 1

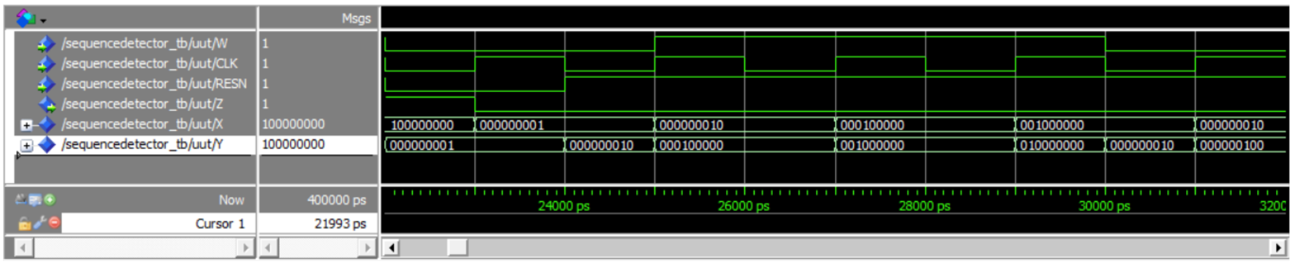


Figure 5 - ModelSim waveforms of Path test #3, w = mixed

This testbench clearly demonstrates that the FSM component is correctly described in VHDL and that the reset mechanism functions as intended to ensure the system always starts in state A.

Synthesis

The VHDL files have been imported in a Quartus prime project with the assignation file for the DE1-SoC board. A new top-level entity has been created to map the connection between the FSM and the DE1 onboard devices. Then, the synthesis tool has generated the file to program the FPGA and it has been uploaded. The video of the practical test is available at this Dropbox link.

Section 2: Modified One-hot FSM

In this section, is required to modify the originally designed one-hot finite state machine to simplify its implementation on an FPGA. The goal is to achieve a coding scheme in which, in the reset state, all flip-flops are 0, taking advantage of the fact that the flip-flops generally include a clear input but not actually a set input.

Delivered files

The delivered files for this exercise are three:

- *FSM.vhd*, VHDL code that implements the FSM with one-hot style
- *FSM_tb.vhd*, VHDL code to validate the FSM by a simulation
- *Flipflop.vhd*, VHDL code for the D-type flip-flop

An additional VHDL file have been used in the synthesis phase to physically connect the FSM to the board's devices. The *.sof* file is ready to be used to program the FPGA.

Design entry

The VHDL description of the circuit is the same used in the previous section, apart from the CC1 sub-circuit. In this part, the VHDL description is the following one. The edited rows have been highlighted with a comment. The one-hot encoding table is "Table 1" in the assignment.

```
-- CC1 sub-circuit
Y(0) <= RESN;          -- !!! EDITED
Y(1) <= not(W) and (not(X(0)) or X(5) or X(6) or X(7) or X(8))
and RESN; -- !!! EDITED
Y(2) <= not(W) and X(1) and RESN;
Y(3) <= not(W) and X(2) and RESN;
Y(4) <= not(W) and (X(3) or X(4)) and RESN;
Y(5) <= W and (not(X(0)) or X(1) or X(2) or X(3) or X(4)) and
RESN;           -- !!! EDITED
Y(6) <= W and X(5) and RESN;
Y(7) <= W and X(6) and RESN;
Y(8) <= W and (X(7) or X(8)) and RESN;
```

The overall circuit and the others VHDL files have not been edited since the previous Section.

Functional simulation

The same testbench of Section 1 have been used to validate this project. The simulation results are available below.

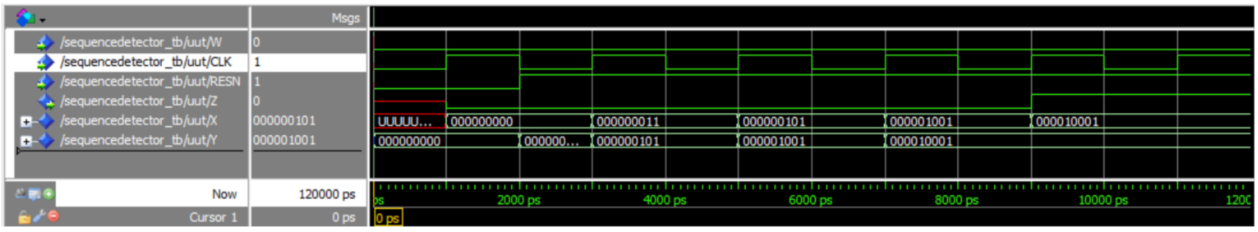

Msgs
/sequencedetector_tb/uut/W 0
/sequencedetector_tb/uut/CLK 1
/sequencedetector_tb/uut/RESN 1
/sequencedetector_tb/uut/Z 0
/sequencedetector_tb/uut/X 000000101
/sequencedetector_tb/uut/Y 000001001
UUUU... 000000000 000000011 000000101 000001001 000010001
000000000 000000... 000000101 000001001 000010001
Now 120000 ps
Cursor 1 0 ps
ps 2000 ps 4000 ps 6000 ps 8000 ps 10000 ps 1200
0 ps

Figure 6 - ModelSim waveforms of Path test #1, w = 0

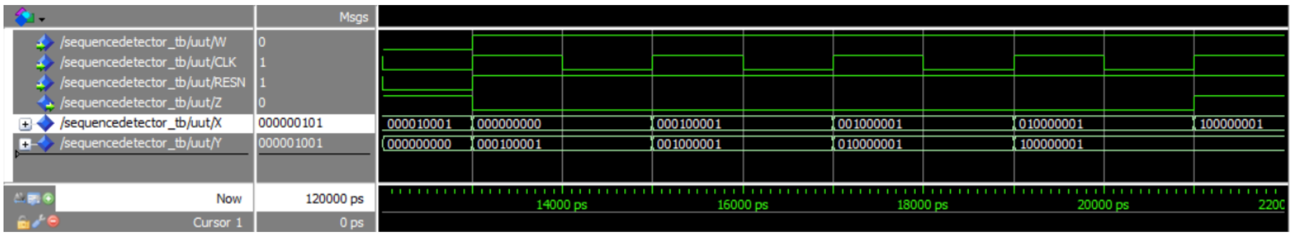

Msgs
/sequencedetector_tb/uut/W 0
/sequencedetector_tb/uut/CLK 1
/sequencedetector_tb/uut/RESN 1
/sequencedetector_tb/uut/Z 0
/sequencedetector_tb/uut/X 000000101
/sequencedetector_tb/uut/Y 000001001
000010001 000000000 000100001 001000001 010000001 100000001
000000000 000100001 001000001 010000001 100000001
Now 120000 ps
Cursor 1 0 ps
14000 ps
16000 ps
18000 ps
20000 ps
220

Figure 7 - ModelSim waveforms of Path test #2, w = 1

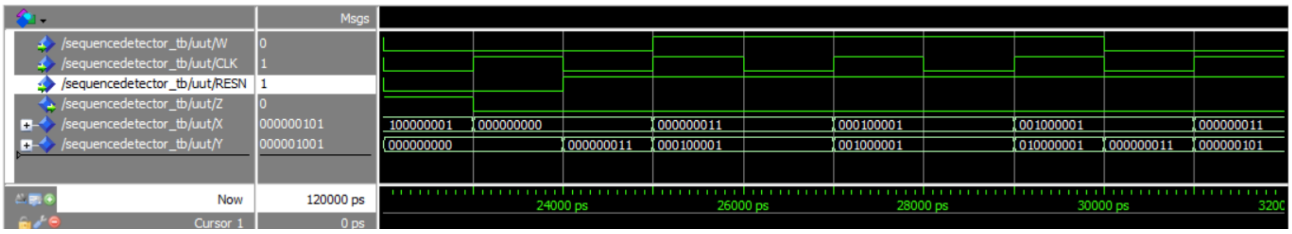


Figure 8 - ModelSim waveforms of Path test #3, w = mixed

The simulation results are exactly equivalent to the one of the previous sections, apart from the internal signals.

Synthesis

The VHDL files have been imported in a Quartus prime project with the assignation file for the DE1-SoC board. A new top-level entity has been created to map the connection between the FSM and the DE1 onboard devices. Then, the synthesis tool has generated the file to program the FPGA and it has been uploaded. The video of the practical test is available at this Dropbox link.

Section 3: Two-process FSM

As already shown in the previous section, a FSM consists of two combinational logic blocks and a memory element. In this section, we clearly defined each of these components using three distinct `PROCESS` blocks.

Delivered files

Two VHDL files have been delivered for the description of the FSM and its simulation:

- *two_process_FSM.vhd*: VHDL code for the implementation of the FSM using a "two process" approach
- *two_process_FSM_tb.vhd*: VHDL code for the test bench used to validate the functionality of the FSM

Design entry

In this section, a different VHDL coding style is used to implement the same FSM from the previous sections. The corresponding state diagram is shown in Figure 1. Unlike the previous implementation, which involved manually deriving logical expressions for each state flip-flop, this method uses a structured VHDL template that models the FSM using two `PROCESS` blocks as illustrated in the code template in the assignment instructions, as shown below:

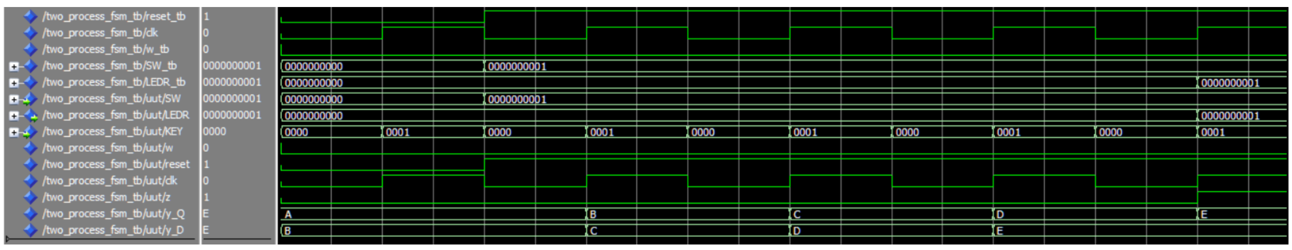
- The first `PROCESS` describes the state table using a `CASE` statement;
- the second `PROCESS` handles the update of the flip-flops that encode the state.
- Additionally, a third `PROCESS` block is included to define the output `z`.

Since this approach does not rely on a structural method using separate components, the entire implementation was carried out within a single VHDL code, named *two_process_FSM.vhd*.

Functional simulation

To verify the functionality of the Finite State Machine (FSM) implemented using two processes, a VHDL testbench must be simulated in ModelSim. This has the same structure as the one used in previous sections.

In the following figures, the signal waveforms observed during the testing of the three different paths are shown.



/two_process_fsm_tb/reset_tb 1
/two_process_fsm_tb/clk 0
/two_process_fsm_tb/w_tb 0
/two_process_fsm_tb/SW_tb 0000000001
/two_process_fsm_tb/LEDR_tb 0000000001
/two_process_fsm_tb/uut/SW 0000000001
/two_process_fsm_tb/uut/LEDR 0000000001
/two_process_fsm_tb/uut/KEY 0000
/two_process_fsm_tb/uut/w 0
/two_process_fsm_tb/uut/reset 1
/two_process_fsm_tb/uut/clk 0
/two_process_fsm_tb/uut/z 1
/two_process_fsm_tb/uut/y_Q E
/two_process_fsm_tb/uut/y_D E
0000000000
0000000001
0000
0001
A
B
C
D
E

Figure 9 - ModelSim waveforms of Path test #1, w = 0

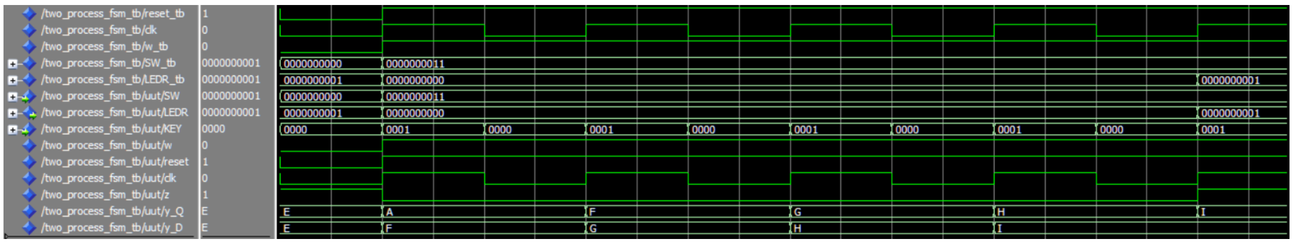

/two_process_fsm_tb/reset_tb
/two_process_fsm_tb/clk
/two_process_fsm_tb/w_tb
/two_process_fsm_tb/SW_tb
/two_process_fsm_tb/LEDR_tb
/two_process_fsm_tb/uut/SW
/two_process_fsm_tb/uut/LEDR
/two_process_fsm_tb/uut/KEY
/two_process_fsm_tb/uut/w
/two_process_fsm_tb/uut/reset
/two_process_fsm_tb/uut/clk
/two_process_fsm_tb/uut/z
/two_process_fsm_tb/uut/y_Q
/two_process_fsm_tb/uut/y_D

Figure 10 - ModelSim waveforms of Path test #2, w = 1

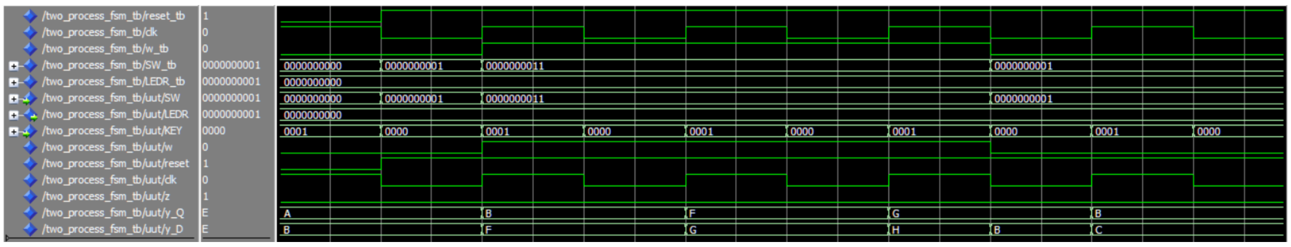

/two_process_fsm_tb/reset_tb
/two_process_fsm_tb/clk
/two_process_fsm_tb/w_tb
/two_process_fsm_tb/SW_tb
/two_process_fsm_tb/LEDR_tb
/two_process_fsm_tb/uut/SW
/two_process_fsm_tb/uut/LEDR
/two_process_fsm_tb/uut/KEY
/two_process_fsm_tb/uut/w
/two_process_fsm_tb/uut/reset
/two_process_fsm_tb/uut/clk
/two_process_fsm_tb/uut/z
/two_process_fsm_tb/uut/y_Q
/two_process_fsm_tb/uut/y_D

Figure 11 - ModelSim waveforms of Path test #3, w = mixed

As expected, every tested combination worked correctly, and all the signals were consistent with those required by the implementation. This series of simulations allows us to test our code in different scenarios. After completing the simulation phase in ModelSim and verifying the correct operation of the implementation, we now move on to the synthesis phase.

Synthesis

After creating a new Quartus project and modifying the VHDL code to use toggle switch `SW0` on the Altera DE1 board as an active-low synchronous reset input for the FSM, `SW1` was used as the `w` input, pushbutton `KEY0` as the manually applied clock input, and `LEDR0` as the output `z`. To assign all these pins on the FPGA, the DE1_SoC.qsf assignment file was imported, as indicated in the board's user manual. Next, as specified in the assignment instructions, the state assignment style was set to "Minimal Bits". After compiling the code, the resulting circuit generated by Quartus that implements the VHDL description can be observed using the RTL Viewer Tool, as shown in Figure 12 below:

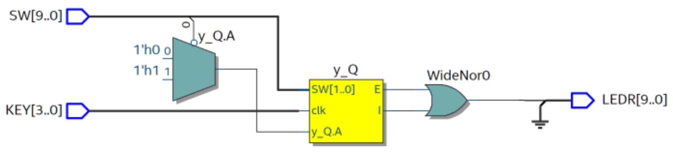


Figure 12 - Register transfer level viewer

The equivalent circuit, with all flip-flops visible, can be examined in more detail using the *"Technology Map Viewer Tool"*, as shown in Figure 13 below:

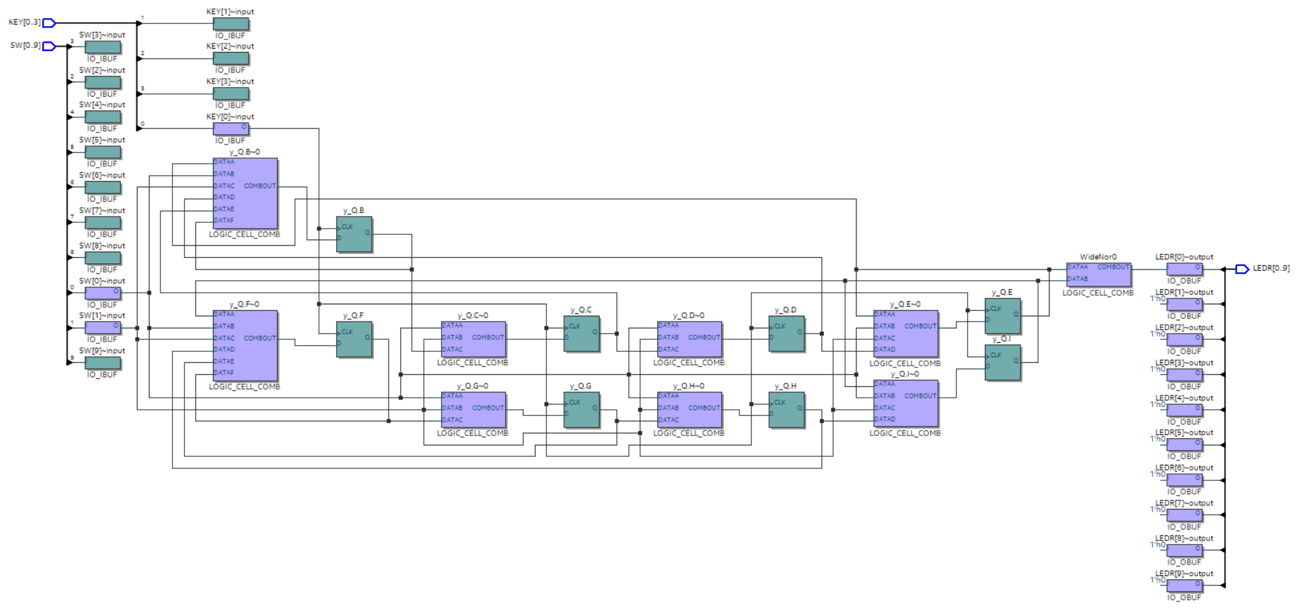


Figure 13 - Technology map viewer

The circuits just shown are consistent with the intended VHDL implementation and meet the expected structure. By now clicking on the main block in Figure 14, we can finally observe the resulting state diagram and compare it with the one specified in the assignment.

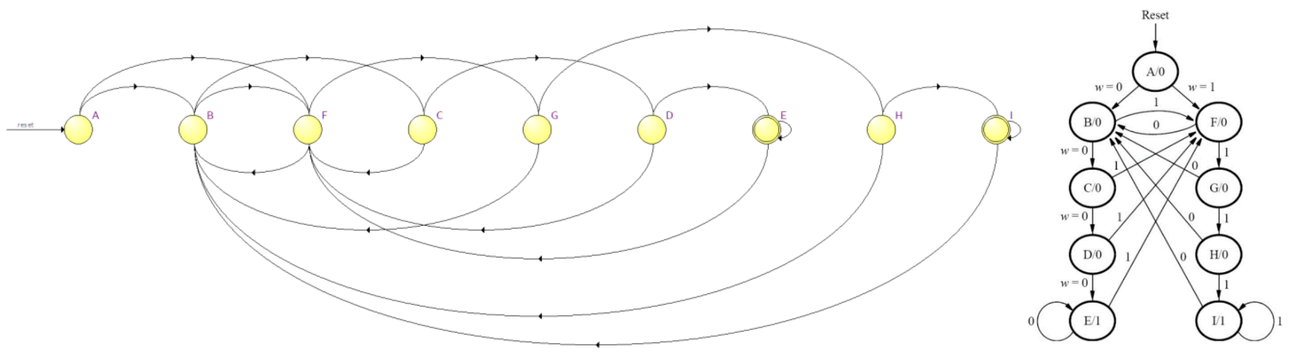
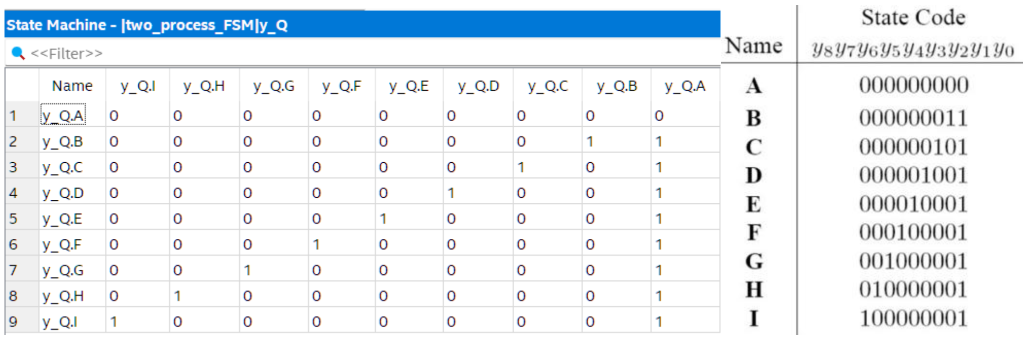


Figure 14 - Comparison between the state diagram from the VHDL two-process implementation (left) and the intended state diagram (right)

By comparing each arc and each state, we can see that the two state diagrams are equivalent.

Changing now the setting for “State Machine Processing” from “Minimal Bits” to “One-Hot”, after having recompiled the code, it is possible to observe the resulting State Machine table from the Compilation Report and comparing it to the “Table 2” provided in the assignment.



State Machine - |two_process_FSM|y_Q

<<Filter>>

	Name	y_Q.I	y_Q.H	y_Q.G	y_Q.F	y_Q.E	y_Q.D	y_Q.C	y_Q.B	y_Q.A
1	y_Q.A	0	0	0	0	0	0	0	0	0
2	y_Q.B	0	0	0	0	0	0	0	1	1
3	y_Q.C	0	0	0	0	0	0	1	0	1
4	y_Q.D	0	0	0	0	0	1	0	0	1
5	y_Q.E	0	0	0	0	1	0	0	0	1
6	y_Q.F	0	0	0	1	0	0	0	0	1
7	y_Q.G	0	0	1	0	0	0	0	0	1
8	y_Q.H	0	1	0	0	0	0	0	0	1
9	y_Q.I	1	0	0	0	0	0	0	0	1

Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000000
B	000000011
C	000000101
D	000001001
E	000010001
F	000100001
G	001000001
H	010000001
I	100000001

Figure 15 - Comparison between the State Machine table from the VHDL two-process implementation with one-hot encoding (left) and the intended state machine table (right)

Also in this case, we can observe that the tables in Figure 15 are equivalent, providing further confirmation that the implementation is correct.

Section 4: “HELLO” FSM

This section is about the design of a circuit to scroll the word "HELLO", followed by a blank space, across six 7-segment displays (HEX5–0). The word must move from right to left at one-second intervals, restarting from the rightmost display after scrolling off the leftmost display to create a continuous loop. The design should utilize six 7-bit registers in a pipeline configuration, where each register's output connects to the input of the next. Each register must drive directly the corresponding 7-segment display.

Delivered files

Twelve VHDL files have been delivered for this project.

- *HELLO_FSM.vhd*, VHDL file of the main entity
- *HELLO_FSM_tb.vhd,* VHDL file of the testbench
- *FSM.vhd,* VHDL file of the finite state machine
- *shiftRegister.vhd,* VHDL file of the 7-bit wide shift register with intermediate outputs
- *regn.vhd,* VHDL file of a single N-bit wide register
- *pulse1Hz,* VHDL file of the pulse generator at a rate of 1 Hz starting from the clock signal of the circuit
- *moduloCounter.vhd,* VHDL file of the modulo counter used to generate the pulse
- *elementaryCounter.vhd,* VHDL file of the elementary building block of the counter
- *mux.vhd,* VHDL file of a 2-to-1 multiplexer
- *flipflop.vhd,* VHDL file of one elementary D-type flip-flop
- *comparator.vhd, VHDL file of a comparator used in the pulse generator*
- *mux7bit.vhd*, VHDL file of the multiplexer (same as *mux.vhd*, but 7-bit wide)

An additional VHDL file have been delivered as the new top-level entity, named *device_HELLO_FSM.vhd,* which includes the connections between the `HELLO_FSM` entity and the DE1 onboard devices. The *.sof* file is the output file generated by Quartus for the 5CSEMA5F31C6N board, and it already includes the pin assignment file *DE1-SoC.qsf*.

Design entry

The circuit operates based on a shift register with intermediate outputs that directly drive the displays. The system is made by six 7-bit registers configured to enable the shift operation between them. During startup, the input data for the registers is provided by the finite state machine (FSM), which sequentially loads each letter encoding into the first register. This data is then shifted progressively to the subsequent registers. Once all letters have been loaded into the registers, the FSM manages a multiplexer to enable the rotation effect. The entire circuit is clocked by a 50 MHz clock signal, though the letters move at a rate of one second.

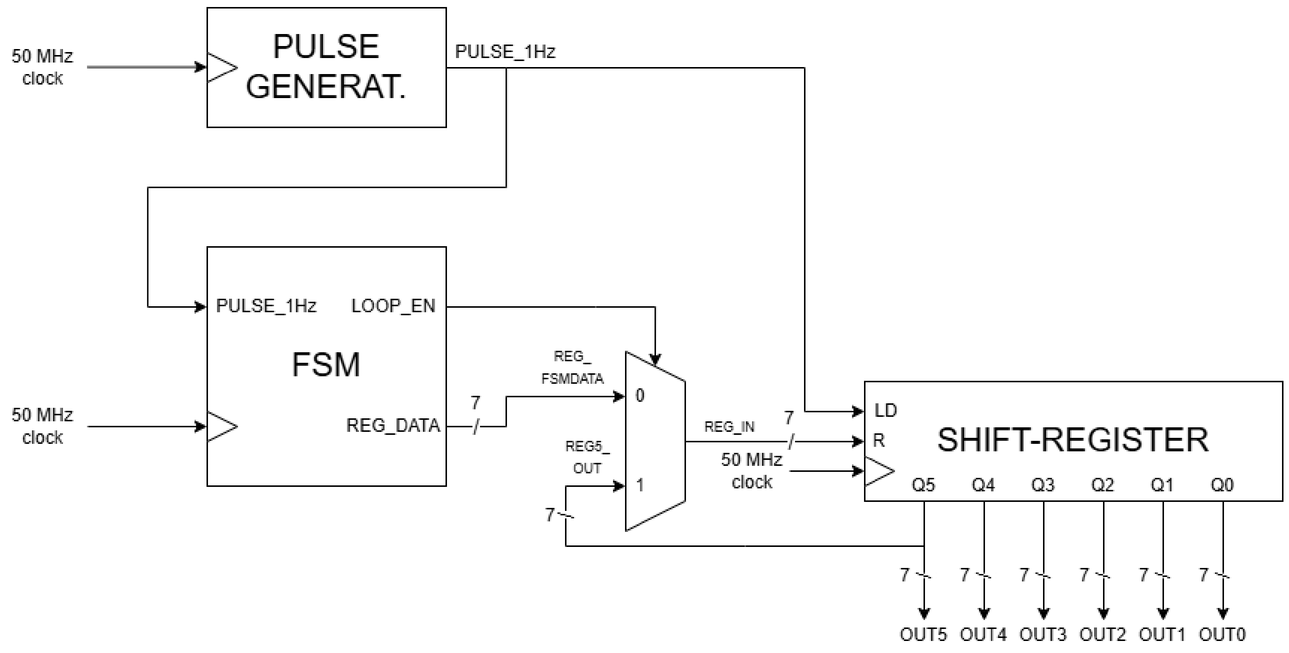


Figure 16 - HELLO FSM internal architecture

FSM

The FSM state diagram is shown below. At startup, the circuit is reset, and it enters the start state. After the startup phase, the FSM transitions through a sequence of states, loading each letter's encoding into the shift register, starting from 'H' and ending with the blank character. Once the final character has been loaded, the circuit enters the loop state, where it remains indefinitely.

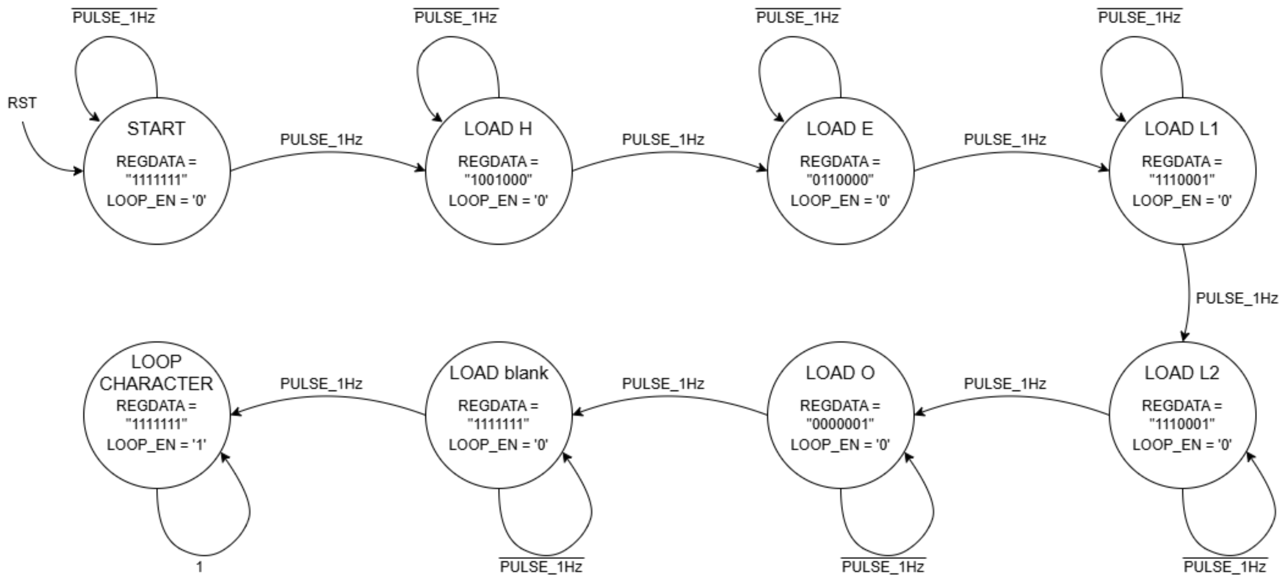

RST
START REGDATA = "1111111" LOOP_EN = '0'
LOAD H REGDATA = "1001000" LOOP_EN = '0'
LOAD E REGDATA = "0110000" LOOP_EN = '0'
LOAD L1 REGDATA = "1110001" LOOP_EN = '0'
LOAD L2 REGDATA = "1110001" LOOP_EN = '0'
LOAD O REGDATA = "0000001" LOOP_EN = '0'
LOAD blank REGDATA = "1111111" LOOP_EN = '0'
LOOP CHARACTER REGDATA = "1111111" LOOP_EN = '1'
PULSE_1Hz
$\overline{\text{PULSE\_1Hz}}$
1

Figure 17 - State diagram for HELLO FSM

Shift register

The architecture of the shift register is shown below. There are five 7-bit wide registers connected in cascade so that the output of the first register is connected to the input of the second and so on. The load command is driven by the 1 Hz pulse, so that the register will shift it data each second. Notice that the clock signal is the 50 MHz clock, but it is not shown in the schematic.

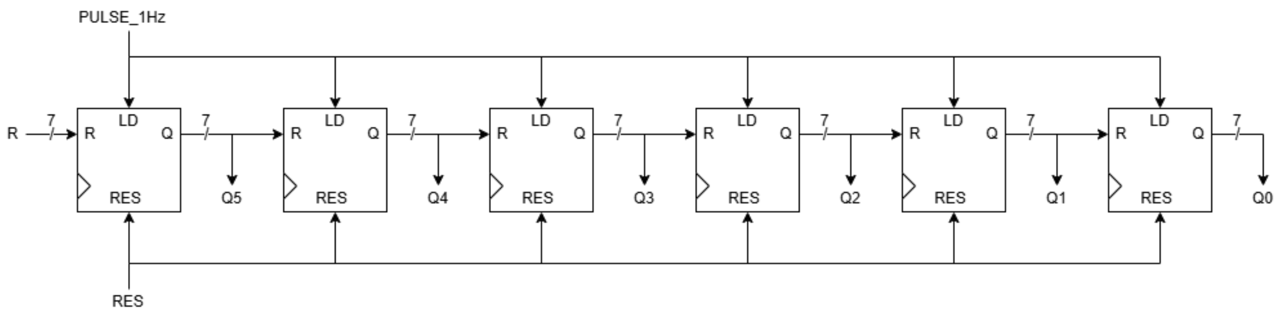


Figure 18 - Internal architecture of the shift register with intermediate outputs

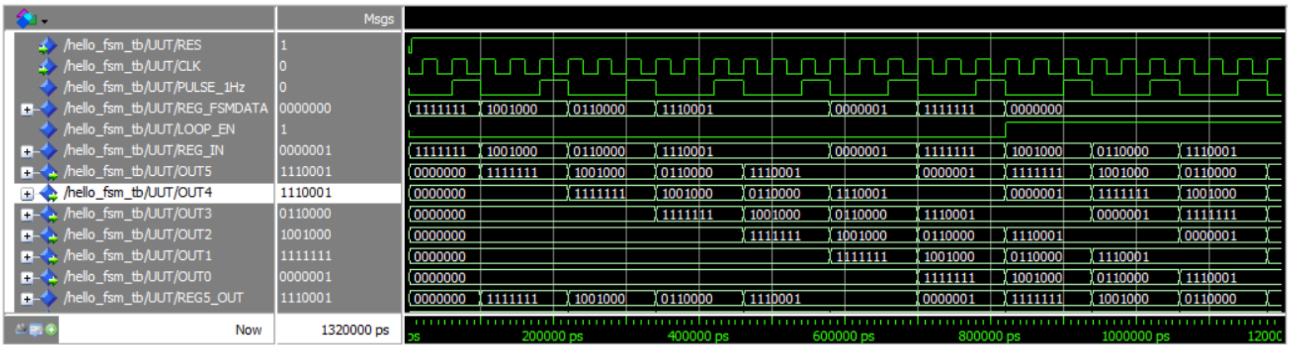
1 Hz pulse generator

The pulse generator is based on a 50-million-modulo counter and a comparator, using the same approach for the "Flashing digit" project, in previous lab experience. The counter increments its value with each clock cycle, while the comparator checks for 49999999 (50000000 – 1) to set its output high. This design produces a pulse with a frequency of 1 Hz.

For a detailed explanation of the modulo counter and the comparator, refer to the previous lab report, Section 4, as these components are identical.

Functional simulation

The functional simulation has been validated with a test bench. First, the test bench reset the system, then it generates an internal clock signal, and the six outputs are going to shift in a loop. Notice that the modulo of the modulo counter has been set as 3 to speed up the simulation time. Hence, the output is going to shift every 3 clock cycles.



	Msgs
/hello_fsm_tb/UUT/RES	1
/hello_fsm_tb/UUT/CLK	0
/hello_fsm_tb/UUT/PULSE_1Hz	0
/hello_fsm_tb/UUT/REG_FSMDATA	0000000
/hello_fsm_tb/UUT/LOOP_EN	1
/hello_fsm_tb/UUT/REG_IN	0000001
/hello_fsm_tb/UUT/OUT5	1110001
/hello_fsm_tb/UUT/OUT4	1110001
/hello_fsm_tb/UUT/OUT3	0110000
/hello_fsm_tb/UUT/OUT2	1001000
/hello_fsm_tb/UUT/OUT1	1111111
/hello_fsm_tb/UUT/OUT0	0000001
/hello_fsm_tb/UUT/REG5_OUT	1110001
Now	1320000 ps

Figure 19 - ModelSim results of the simulation of the HELLO FSM

Synthesis

The VHDL files described before have been included in a new Quartus project in order to test the circuit on the DE1-SoC board. A new top-level entity has been created to include the "HELLO_FSM" described above and the connections between the entity and the DE1-SoC onboard devices. The video of the practical test is available at this Dropbox link.

Conclusions

The topics covered in this report have proven particularly useful for further investigating the implementation of Finite State Machines (FSMs) in VHDL. In particular:

- In Sections 5.1, 5.2 and 5.3, different implementations were explored, using various approaches to realize the same FSMs. This was especially helpful in understanding how many different ways an FSM can be implemented and in analysing the simulations from different perspectives.
- In Section 5.4, building on the experience gained in the previous sections, a more design-oriented approach was required to implement an entire circuit to scroll the word "HELLO" on the 7-segment displays based on a state machine.

Overall, the completion of this laboratory experience has improved skills related to FSMs by testing both practical and simulation skills. The results obtained from each implementation are consistent with expectations.

Sources and notes

For all the VHDL files provided, an online tool has been used to format the code to ensure coherence between all the different files. The tool is free to use, and it is available at this link: VHDL Beautifier.

The VHDL files were developed using course materials produced by Prof. G. Masera (available on the course webpage) and the following eBooks recommended by the professor:

1. Mealy, Bryan, and Fabrizio Tappero. *Free Range VHDL*. 2016.
2. Ashenden, Peter J. *The VHDL Cookbook*. 1990.