## "FPGA-Programming" Exercise Sheet VII:

### Finite State Machines

This is an exercise in using finite state machines in VHDL.

**As target hardware always choose FPGA chip Cyclone V SoC 5CSEMA5F31C6**

1) *Sequence-recognizing state machine*

In this part, a finite state machine (FSM) should be implemented, that recognizes two specific sequences of applied input symbols, namely four consecutive 1s or four consecutive 0-bits. There is an input $w$ and an output $z$. Whenever $w = 1$ or $w = 0$ for four consecutive clock pulses the value of $z$ has to be 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output $z$ will be equal to 1 after the fourth and fifth pulses. Following figure illustrates the required relationship between $w$ and $z$.

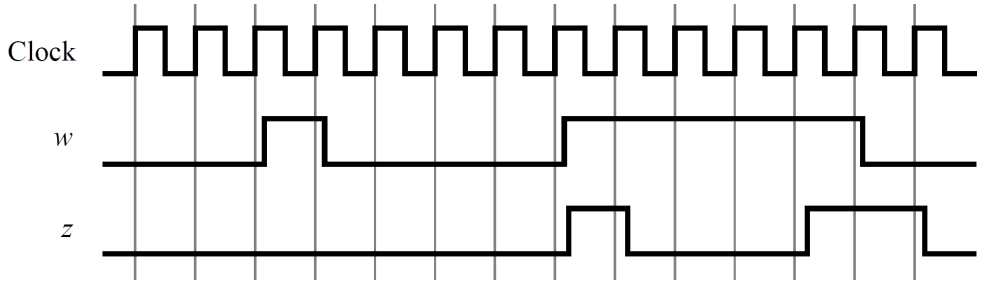


**Figure 1.1:** Timing diagram of the sequence-recognizing state machine

i) Draw the state diagram for this **Moore state machine**. Die states of the state machine are termed $A$ to $I$, in which state $A$ represents the initial state after reset, states $B$ to $E$ based on the reset state symbolize a sequence of 0-bits, and the states $F$ to $I$ based on the reset state symbolize a sequence of 1-bits.

For this part the state machine should be derived manually out of the state diagram including the boolean expressions for each of the state machine flip-flops. In order to realize this state machine, a total of nine flip-flops $DFF0$ to $FDD8$ are necessary as well as the one-hot state assignment given in following table.

| | State Code | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| State | $DFF8$ | $DFF7$ | $DFF6$ | $DFF5$ | $DFF4$ | $DFF3$ | $DFF2$ | $DFF1$ | $DFF0$ |
| $A$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| $B$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| $C$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $D$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $E$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| $F$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| $G$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $H$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $I$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Perform following steps:

ii) Create a new Quartus project for the circuit, named **e_my_bitFSM**

iii) First describe a VHDL entity of a pos-edge triggered D-type flip-flop (**e_flipflop**) with synchronous low-active reset and synchronous low-active set input. Use a *sequential process* with priority encoder.

iv) Describe the top-level entity (**e_my_bitFSM**) which instantiates nine components of the D-type flip-flop. Determine the logic functions to drive the individual input-ports. Only your simple boolean expressions here! Above table with the one-hot state assignments as well as the drawn state diagram facilitate the determination of the logic expressions.

v) Use switch $SW[0]$ as synchronous low-active reset input of the state machine and switch $SW[1]$ as $w$ input. Push-button $KEY[0]$ should be used as manual clock input. Output $z$ should be displayed on $LEDR[9]$ the flip-flop outputs should be displayed on $LEDR[8-0]$.

vi) Import the necessary pin assignments, compile the described state machine and perform a functional simulation by means of two stimuli using an appropriate test bench.

vii) Download the state machine and verify the correct functionality in hardware.

2) *Modified sequence-recognizing state machine*

Based on the sequence-recognizing state machine of part 1), its function should be readjusted according to following modified one-hot state assignments.

| | State Code | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| State | $DFF8$ | $DFF7$ | $DFF6$ | $DFF5$ | $DFF4$ | $DFF3$ | $DFF2$ | $DFF1$ | $DFF0$ |
| $A$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $B$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| $C$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| $D$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| $E$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| $F$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| $G$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| $H$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| $I$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

In practice it is often desirable to set all flip-flop outputs to the value 0 in the reset state. This is accomplished by inverting the state variable $DFF0$. Implement the modified version of the sequence-recognizing state machine.

**Hint:** Only very few changes of the code should be needed!

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_bitFSM2**

ii) First describe a VHDL entity of a pos-edge triggered D-type flip-flop (**e_flipflop**) with synchronous low-active reset. Use a *sequential process* with priority encoder.

iii) Create the top-level entity (**e_my_bitFSM2**, using the one of part 1) as basis. Think about which changes are required and realize them.

iv) Use switch $SW[0]$ as synchronous low-active reset input of the state machine and switch $SW[1]$ as $w$ input. Push-button $KEY[0]$ should be used as manual clock input. Output $z$ should be displayed on $LEDR[9]$ the flip-flop outputs should be displayed on $LEDR[8-0]$.

v) Import the necessary pin assignments, compile the described state machine, download the circuit and verify its correct functionality in hardware.

3) *Finite State Machine with high-level syntax*

In this part a different style of VHDL code for the FSM of part 1) should be used.

In this version of the code the logic equations should not be derived manually for each of the flip-flops. Instead, the state table of the FSM should be described using a VHDL **case** statement in a process block. Another process block should be used to instantiate the state flip-flops. To implement the FSM four state flip-flops and a binary code should be used, as shown in following table.

| | State Code | | | |
|---|---|---|---|---|
| State | $DFF3$ | $DFF2$ | $DFF1$ | $DFF0$ |
| $A$ | 0 | 0 | 0 | 0 |
| $B$ | 0 | 0 | 0 | 1 |
| $C$ | 0 | 0 | 1 | 0 |
| $D$ | 0 | 0 | 1 | 1 |
| $E$ | 0 | 1 | 0 | 0 |
| $F$ | 0 | 1 | 0 | 1 |
| $G$ | 0 | 1 | 1 | 0 |
| $H$ | 0 | 1 | 1 | 1 |
| $I$ | 1 | 0 | 0 | 0 |

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_FSM**.

ii) Develop the top-level entity **e_my_FSM**. In the declarations part, declare the states **A** to **I** by using the VHDL enumeration type declaration **type - is**.

iii) Describe a *sequential process* with low-active synchronous reset for the state change.

iv) Describe a *combinatorial process* for the state transitions.

v) Describe a *combinatorial process* for displaying the current state on $LEDR[8-0]$. State **A** should be displayed as "000000001", state **B** as "000000010, and so on.

vi) Describe the remaining logic with *concurrent statements*. Use the same input and output pins as in part 1) and part 2).

vii) Before compilation it is necessary to instruct the synthesis tool of Quartus to use exactly the state coding of the table above. If this will not be done, the synthesis tool automatically implements a self-selected state coding. This setting can be performed via **Assignments** → **Settings** → **Compiler Settings** → **Advanced Settings (Synthesis)** where the setting **State Machine Processing** has to be set to **User-Encoded**. By using following syntax above state coding can be realized:

```
-- Synthesis Attribute Declarations
attribute syn_encoding: string;
attribute syn_encoding if t_fsm_states : type is "0000 0001 0010
    0011 0100 0101 0110 0111 1000";
```

viii) Import the necessary pin assignments and compile the project. Inspect the resulting circuit by using Quartus **RTL Viewer**. Examine the state diagram and compare it with the designed state diagram of part 1). In order to review the state coding open the **Analysis and Synthesis** section of the **Compilation Report** and press on **State Machines**.

ix) Download the finite state machine and tests its functionality.

x) Open **Assignments** → **Settings** → **Compiler Settings** → **Advanced Settings (Synthesis)** again and change the **State Machine Processing** setting to **One-Hot**. Comment the attribute section of the VHDL code. Recompile the state machine, open the **Analysis and Synthesis** of the **Compilation Report**, click on **State Machines** and compare the state coding with the one of part 2).

4) *Finite State Machine using shift register*

The simple sequence-recognizing state machine of parts 1) - 3) can also be realized in a straightforward manner by using shift registers instead of the formal description process of part 3). This should be demonstrated in this part.

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_shiftregFSM**.

ii) Create the top-level entity (**e_my_shiftregFSM**). In the code, describe two $4$-bit long shift register, one for the recognition of a sequence of $0$-bit values, one for the recognition of $1$-bit values. Describe the functionality of both registers in a *sequential process* with synchronous low-active reset.

iii) Add an appropriate logic equation for the assignment of the output $z$. Use switch $SW[0]$ as synchronous low-active reset input of the finite state machine and shwitch $SW[1]$ as input $w$. Push-button $KEY[0]$ should be used as manual clock input. The output $z$ should be displayed on $LEDR[9]$, the state of the shift register for the recognition of the sequence of $0$-bit values should be displayed on $LEDR[3-0]$, the state of the shift register for the recognition of the sequence of $1$-bit values should be displayed on $LEDR[7-4]$.

iv) Import the necessary pin assignments, compile the described state machine, download it and verify the correct functionality in hardware.

v) Constitute if the state machine also could be based on only one $4$-bit long shift register.

5) *Coffee Maker finite state machine*

In this part a coffee maker should be described in VHDL code as FSM. The coffee maker should be able to produce four different products, which are coffee, cappuccino, espresso and hot chocolate. The finite state machine has the states IDLE, GRIND, BREW, CHOC_POWDER, PUMP_WATER, PUMP_MILK and DONE. The finite state machine furthermore has an asynchronous reset input (SW(9)). The reset state and the operational states of the state machine will be displayed on the red LEDs (LEDR). Furthermore the choice, the reset state as well as the idle state should be shown on the 7-segment displays (HEX5 – HEX0). Following figure clarifies the requirements:

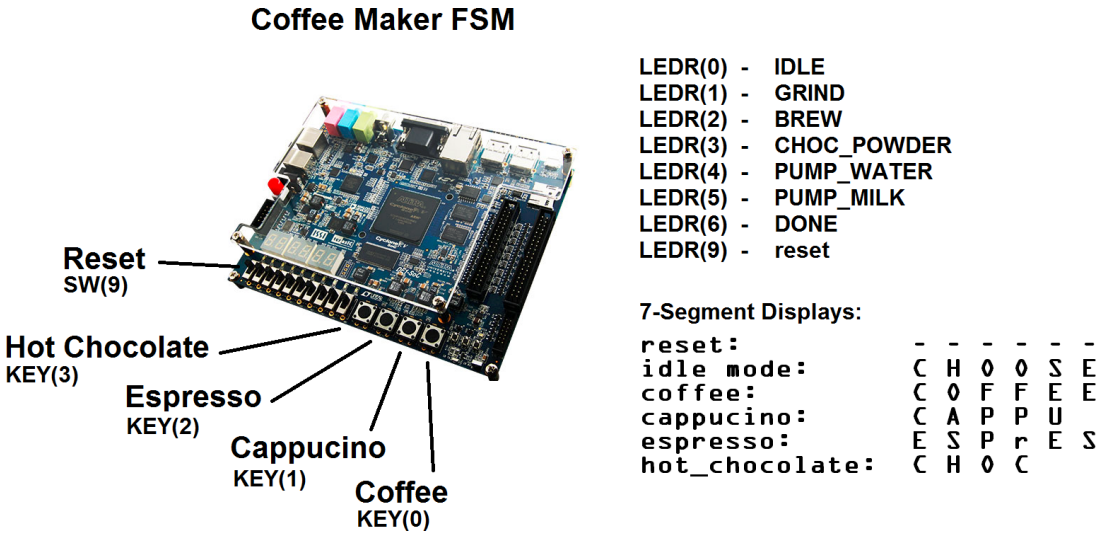


**Figure 5.1:** Coffee Maker Finite State Machine - overview

The circuit furthermore requires two counters, a 3-second counter, as well as a 6-second counter. Depending on the choice the state machine either stays 3 seconds or 6 seconds in the individual states. Following sequences are defined:

```
Coffee:        IDLE --> GRIND(6s) --> BREW(6s) --> PUMP_WATER(6s) --> DONE(3s) --> IDLE

Cappuccino:    IDLE --> GRIND(3s) --> BREW(6s) --> PUMP_WATER(3s) --> PUMP_MILK(3s) --> DONE(3s) --> IDLE

Espresso:      IDLE --> GRIND(3s) --> BREW(6s) --> PUMP_WATER(3s) --> DONE(3s) --> IDLE

Hot Chocolate: IDLE --> BREW(6s)  --> CHOC_POWDER(3s) --> PUMP_WATER(6s) --> DONE(3s) --> IDLE
```

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_coffee_maker_FSM**.

ii) Describe a VHDL entity of a generic counter **e_modulo_counter** with a asynchronous low-active reset and rollover output bit. Use this counter to derive a 1 second counter, as well as the required 3 second counter and 6 second counter.

iii) Develop the top-level entity **e_my_coffee_maker_FSM**.
Use a combinatorial process which handels the way the 7-segment displays must be driven.
Use a sequential process for the product choice.
Use a sequential process to synchronize the determined next states of the finite state machine.
Use a sequential process to handle the individual states by analysing the done signals and setting the appropriate reset signals of the counter circuits.

iv) Import the necessary pin assignments, compile the circuit and test its functionality in hardware.

6) *Morse-Code with finite state machine*

In this part a Morse-code encoder should be implemented using a FSM. This code uses patterns of short an long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). Here a dot represents one time unit, a dash represents three time units. The gap between two parts of the same character is one time unit, the gap between the characters of a word is three time units, the gap between two words is seven time units. Following figure illustrates the international Morse-Code.

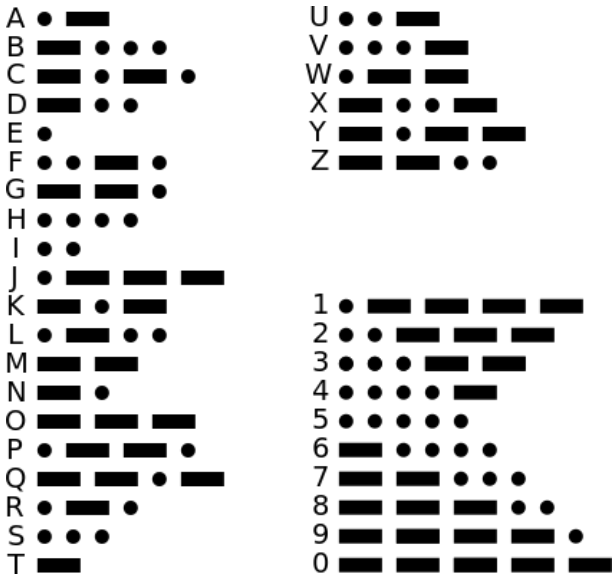


**Figure 6.1:** Morse Code

Goal of this part is to design a circuit, that takes as input one of the *first eight* characters of the Morse-Alphabet and displays the Morse-Code of this character on the red LEDs. Therefore the circuit should use switches $SW[2-0]$, as well as buttons $KEY[1-0]$ as inputs. If $KEY[1]$ is pressed, the circuit should display the Morse-Code of the character which has been set by the switches $SW[2-0]$ ($000\hat{=}A$, $001\hat{=}B, \ldots$). $KEY[0]$ acts as asynchronous reset input. Following figure shows the high-level diagram of the circuit.

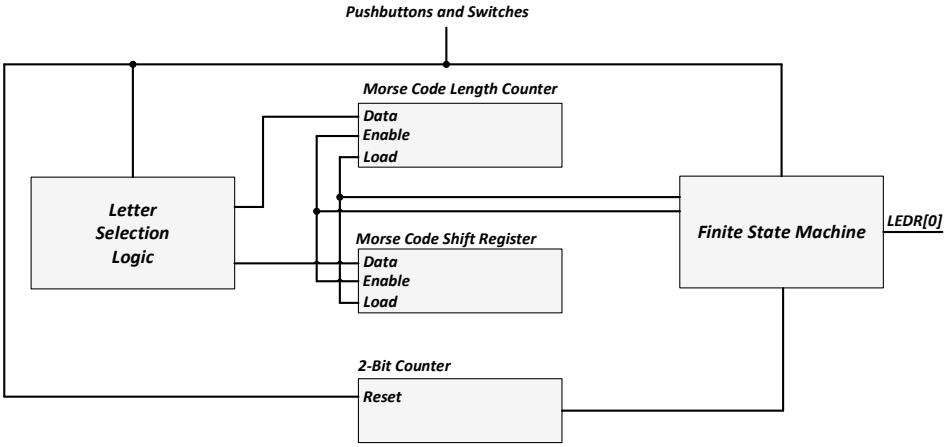


**Figure 6.2:** High-level illustration of the Morse-Code circuit using an FSM

Perform following steps:

i) Create a new Quartus project for the circuit, named **e_my_morseFSM**.

ii) For the description of the generic modulo-$k$ counter (**e_modulo_counter_ser**) use the same as in exercise V) part 4).

iii) Again use a *combinatorial process* for the determination of the chosen Morse-Code character as well as its length.

iv) Again use a *sequential process* with asynchronous reset which deals with the buffering and processing of the chosen Morse-Code character.

v) Use the VHDL type **constant** for the specification of the characters, whose length and the switch encoding.

vi) Instead of using the logic blocks of exercise V) part 4) here a finite state machine should be established. The FSM should exhibit following states:
**S_WAIT_SEND**, **S_WAIT_BLANK**,
**S_SEND_DOT**, **S_SEND_DASH_1**, **S_SEND_DASH_2**, **S_SEND_DASH_3**,
**S_RELEASE_SEND**.

vii) Import the necessary pin assignments, compile the circuit and test its functionality in hardware.