Digital Electronics - Project help
A simple way to design a Simon game

Academic year 2022-2023 / Teaching assistant : Arthur Fyon

Introduction

This file explains the steps you should follow if you wanted to design a Simon game, represented in Figure 1.

Figure 1 – Simon game

Let's explain the rules of the game. The game lights up one of the four colors. The player must then press the button of the color that has just lit up within a short time. The game repeats the same color, then randomly adds a new color. The player must repeat this new sequence. Each time the player correctly reproduces the sequence, the game adds a new color. The game ends when the player makes a mistake while reproducing the sequence of color (game is lost) or when the player succeeds in repeating the sequence of maximum length (game is won).

In the following, the main steps of designing this game on a CPLD will be described as well as the associated VHDL code. The first step is to define the different inputs and outputs that the game needs. Then, the state machine diagram will be constructed. Finally, the VHDL code running the game will be constructed step by step, starting with the code for a specific state, then adding the other states to finally arrive at the complete code.

1 Inputs/Outputs of the Simon game

Concerning the inputs, we will use 2 pins for the 2 clocks of the development board, 4 pins for the different buttons the player have to press when reproducing the sequence of color and

1 pin for a reset button in order to start a new game when the previous game is over.

Concerning the outputs, we will use 4 pins to drive the LEDs and 1 pin for a specific LED indicating if the user has to repeat the color sequence of not.

Thus, the following signals will be used in the game entity :

- `clock1` : input clock 1 signal coming from the 555 timer of the development board, this signal will be used as the main clock of the CPLD
  $\longrightarrow$ *std_logic* ;
- `clock0` : input clock 0 signal coming from the 555 timer of the development board, this signal will be used for the random number generator (more details about this later)
  $\longrightarrow$ *std_logic* ;
- `buttons` : input signal of the 4 buttons associated with the 4 different LEDs of the game in order to let the player reproduce the color sequence
  $\longrightarrow$ *std_logic_vector*(0 to 3) ;
- `reset` : input signal of the reset button to restart the game when the previous one is over
  $\longrightarrow$ *std_logic* ;
- `leds` : output signal of the 4 LEDs in order to display the color sequence
  $\longrightarrow$ *std_logic_vector*(0 to 3) ;
- `play` : output signal of the specific LED indicating in real time if it is the player's turn or not (not necessary but nice)
  $\longrightarrow$ *std_logic*.

The code of the whole entity is represented in Figure 2.

```
entity game_simon is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 3) := "0000";
   play : out std_logic := '0';
   buttons : in std_logic_vector(0 to 3);
   reset : in std_logic
);
end entity game_simon;
```

FIGURE 2 – Entity of the Simon game

2 State machine diagram

A possible state machine diagram is represented in Figure 3. The initial state of the game is **Add random color** that adds in the sequence a random number between 0 and 3 representing the LED color. The following state is **Display sequence** whose role is to display the whole current color sequence of the game. Then, when the color sequence display is over, the player

has to reproduce the color sequence that have been displayed. The state **User sequence verification** has to interpret the signal `buttons` and to verify if the user sequence matches the color sequence that had been displayed. On one hand, if the user sequence is wrong, the following state is **Game over**. On the other hand, if the user sequence is correct and that the size of the current sequence is maximum, the game is also over and the following state is **End game**. Otherwise, a random number is added to the sequence thanks to the state **Add random color**.

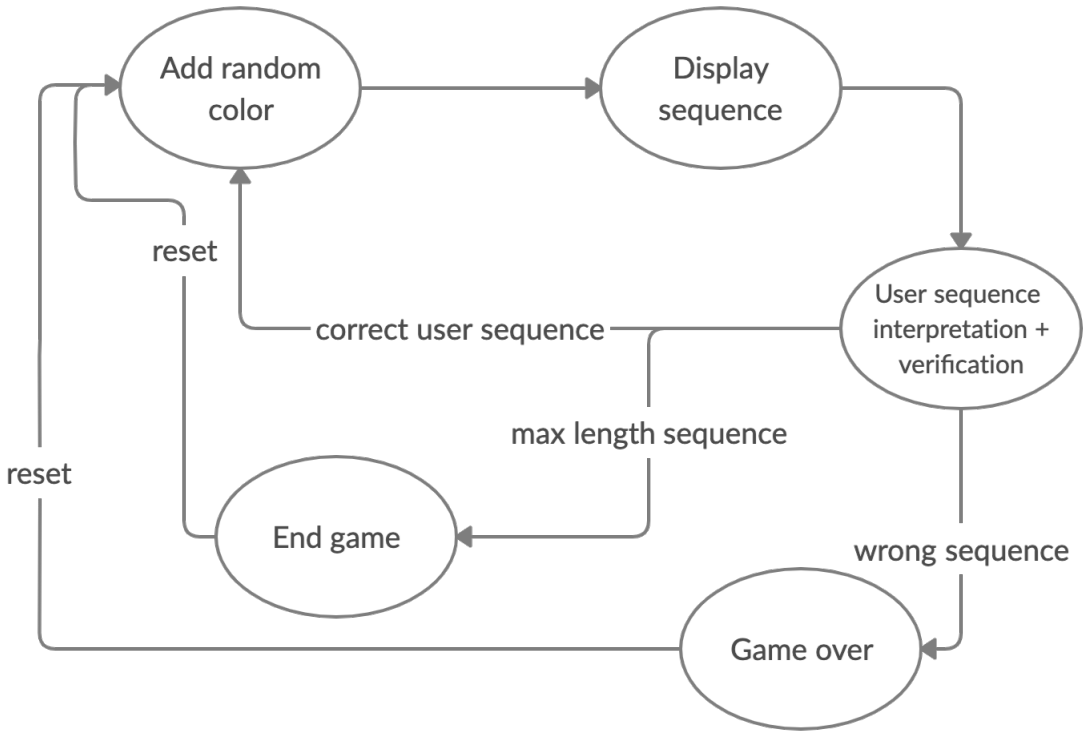

FIGURE 3 – State machine of the Simon game

From this, we can already think about essential signals and/or constant of the architecture of the game :

- `max_length_sequence` : a constant defining the maximum size of the color sequence
  $\longrightarrow$ *integer* ;
- `sequence` : a signal used to save the color sequence created during the execution of the game, this will be a vector containing number ranging from 0 to 3
  $\longrightarrow$ *array*(0 to `max_length_sequence - 1`) of *integer* range 0 to 3 [1] ;
- `length_sequence` : a signal indicating the current length of the color sequence during the game execution, its value is thus increased during the game
  $\longrightarrow$ *integer* range from 0 to `max_length_sequence` ;

1. "Range" is mainly useful to limit the number of logic elements used for the `sequence` signal. Indeed, by specifying the values that each element of `sequence` can take limit the allocated memory for it

- `counter_seq` : a signal used to save the index of the active element of the color sequence either during the **Display sequence** state or **User sequence verification** state
  $\longrightarrow$ *integer* range from 0 to `max_length_sequence` - 1 ;
- `state` : a signal used to indicate in which state the game is currently
  $\longrightarrow$ signal of *type* states is (add_random, display, sequence_verification, done, game_over).

3 Codes of the different states

The best way to implement such a consequent project is to implement each part (each state) progressively and to verify if everything works correctly progressively. Thus, a good way to do would be to first design the **Display sequence** state, then the **Add random color** state, to finally implement the **User sequence verification** state (being the most complicated one). The **End game** and **Game over** states are implemented at the very end.

Before starting coding, one has to think about the clocks of the system. In this project, `clock1` (2-89 Hz) will be used as the main clock of the game. One should always keep in mind that this clock is "fast" compared to human being, a single clock tick might be very fast to be detected by the naked eye. `clock0` (50-266 Hz) will be used to generate a random number. It is indeed convenient to chose the fastest clock to generate a random number (more details about this later).

3.1 Display sequence state

This state must display the color sequence of length `counter_seq` contained in the `sequence` signal using the output signal `leds`. To test this, we will set `sequence` as a constant as well as `length_sequence` since no random number generator is implemented yet. We will also use the `counter_seq` signal. The sequence display will be performed as the following :

1. All the LEDs are off at start ;
2. The LED of the `sequence` at index `counter_seq` is lit up for a certain amount of time ;
3. All the LEDs are again off during a certain amount of time to indicate the user that the game is about to lit up the next LED (this is used mainly to avoid any ambiguity if the same LED has to be lit up twice in a row to have a flashing signal rather than a continuous signal of a duration twice greater $\longrightarrow$ there is a clear separation between each LED of the `sequence`) ;
4. `counter_seq` is incremented ;
5. The code goes back to step 2 until `counter_seq` has reached `length_sequence`. If `counter_seq` = `length_sequence`, the state **Display sequence** is over.

The last thing to implement the so called "certain amount of time" during which a LED is lit. Indeed, we cannot only lit up the LED of interest and turn them all off for one clock cycle, this would be too fast. Thus, we need another counter signal, `counter_display`, to count a certain number of clock cycles for which the LED of interest must be lit on and another certain number of clock cycles for which all the LEDs are turned off. Note that this "certain number of clock cycles" depends on the `clock1` frequency you chose and on how long you want the LEDs to remain on and off.

An example of entity/architecture header is represented in Figure 4. An example of architecture code is represented in Figure 5.

```
library ieee;
use ieee.std_logic_1164.all;

entity display_sequence is port(
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 3)
);
end entity display_sequence;

architecture display_sequence_architecture of display_sequence is

   constant max_length_sequence : integer := 6;
   constant length_sequence : integer := 6;

   type sequence_type is array(0 to max_length_sequence - 1) of integer range 0 to 3;
   constant sequence : sequence_type := (1, 0, 3, 3, 2, 0);

   type states is (display, done);
   signal state : states := display;

   signal counter_seq : integer range 0 to max_length_sequence := 0;
   signal counter_display : integer range 0 to 31 := 0;

begin
   state_machine : process(clock1)
```

FIGURE 4 – Entity/architecture header of the state **Display sequence**

```
begin
   state_machine : process(clock1)
   begin
      if( rising_edge(clock1) ) then
         case state is
            when display =>
               leds <= "0000"; --All the LEDs are off if counter_display < 8
               if( counter_display > 7 ) then
                  leds(sequence(counter_seq)) <= '1'; --The LED of the sequence at index counter_seq is on

                  if( counter_display = 31 ) then --End of display of one LED of the sequence
                     counter_display <= 0;
                     if( counter_seq = length_sequence - 1) then --End of the sequence
                        counter_seq <= 0;
                        state <= done;
                     else --Need to display to following element of the sequence
                        counter_seq <= counter_seq + 1;
                     end if;
                  else
                     counter_display <= counter_display + 1;
                  end if;
               else
                  counter_display <= counter_display + 1;
               end if;

            when done =>
               leds <= "1111"; --All the LEDs are on to indicate end of display

         end case;
      end if;
   end process state_machine;
end architecture display_sequence_architecture;
```

FIGURE 5 – Architecture of the state **Display sequence**

3.2 Add random state

Once the display of the sequence is implemented, one can be interested in constructing the sequence dynamically and randomly until the maximum length is reached. To do so, one has to implement a random number generator.

In VHDL, the code is reconfigured to create a logic circuit whose results are deterministic (except for circuits with feedback which causes instability of the system). To create a probabilistic signal, it is necessary to find an unpredictable element to generate randomness. As our program works in real time, we can for example use the unknown time when the user will press a button. Here, it has been decided to use clock0, faster than clock1, as a random signal. Thus, by creating a signal varying from 0 to 3 on all the rising edges of clock0, we can then read the value of this indeterminate signal when it is needed, i.e. when an element must be added to our sequence. in our sequence. We speak here of pseudo-random because the system remains deterministic but the result is unknown to the user and therefore random.

In practice, one has to add a signal in the architecture header being the random number : an *integer* ranging from 0 to 3. This signal will be changed in another process triggered on rising edges of `clock0`. Example of updated entity/architecture header and random generator process is represented in Figure 6.

```
library ieee;
use ieee.std_logic_1164.all;

entity display_random is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 3) := "0000"
);
end entity display_random;

architecture display_random_arch of display_random is

   constant max_length_sequence : integer := 15;

   type sequence_type is array(0 to max_length_sequence-1) of integer range 0 to 3;
   signal sequence : sequence_type;

   type states is (add_random, display, done);
   signal state : states := add_random;

   signal length_sequence : integer range 0 to max_length_sequence := 0;
   signal counter_seq : integer range 0 to max_length_sequence-1 := 0;
   signal counter_display : integer range 0 to 31 := 0;

   signal random : integer range 0 to 3;

begin
   random_generator : process(clock0)
   begin
      if( rising_edge(clock0) ) then
         if( random = 3 ) then
            random <= 0;
         else
            random <= random + 1;
         end if;
      end if;
   end process random_generator;
```

FIGURE 6 – Updated entity/architecture header with the random generator process

Indeed, it is impossible to assign a signal to two different processes. This would produce a voltage conflict between 2 output voltages. Here, the random signal is assigned in the `random_generator` process and read in the `state_machine process`, which is allowed by *Quartus* and can be synthesized into a logic circuit.

An example of updated state machine process is represented in Figure 7.

```
state_machine : process(clock1)
begin
   if( rising_edge(clock1) ) then
      case state is
         when add_random =>
            state <= display;
            sequence(length_sequence) <= random; --The random number is added to the end of the sequence
            length_sequence <= length_sequence + 1; --The size of the sequence is incremented

         when display =>
            leds <= "0000"; --All the LEDs are off if counter_display < 8
            if( counter_display > 7 ) then
               leds(sequence(counter_seq)) <= '1'; --The LED of the sequence at index counter_seq is on

               if( counter_display = 31 ) then --End of display of one LED of the sequence
                  counter_display <= 0;
                  if( counter_seq = length_sequence - 1) then --End of the sequence
                     counter_seq <= 0;
                     if( length_sequence = max_length_sequence ) then --We reached the maximum sequence length
                        state <= done;
                     else --We add an element to the sequence
                        state <= add_random;
                     end if;
                  else --Need to display to following element of the sequence
                     counter_seq <= counter_seq + 1;
                  end if;
               else
                  counter_display <= counter_display + 1;
               end if;
            else
               counter_display <= counter_display + 1;
            end if;

         when done => --All the LEDs are on to indicate end of display
            leds <= "1111";

      end case;
   end if;
end process state_machine;
end architecture display_random_arch;
```

FIGURE 7 – Updated state machine process

3.3 User sequence verification

Now that we can create and display a random color sequence, it is time to interpret the sequence produced by a player. To do so, we first need to read the button the user had pressed. This information is contained in the input signal `buttons`. However, when the user presses a button, the input signal is at high voltage for several clock cycles of `clock1`. However, this input signal should not be interpreted as multiple selection of the same button. To avoid this, a new Boolean signal `new_user_command` is true if there was no button pressed on the previous `clock1` tick. Thus, only the first rising edge of the signal from the button is interpreted as a new user command. Moreover, the bounces of the buttons are also filtered because they take place over a very short period of time (much less than one period of the `clock1`). If we had a clock with a higher frequency (in the MHz range), an additional filter would have to be implemented.

In addition to that, it is required to add a variable `user_selection` containing the index of the button pressed by the user. It must be defined as a variable since it must be updated directly in order to be reused during the current execution of the process. An example of updated entity/architecture header is represented in Figure 8.

```
library ieee;
use ieee.std_logic_1164.all;

entity game_simon is port(
   clock0 : in std_logic;
   clock1 : in std_logic;
   leds : out std_logic_vector(0 to 3) := "0000";
   play : out std_logic := '0';
   buttons : in std_logic_vector(0 to 3);
   reset : in std_logic
);
end entity game_simon;

architecture game_simon_architecture of game_simon is

   constant max_length_sequence : integer := 15;

   type sequence_type is array(0 to max_length_sequence - 1) of integer range 0 to 3;
   signal sequence : sequence_type;

   type states is (add_random, display, sequence_verification, done, game_over);
   signal state : states := add_random;

   signal length_sequence : integer range 0 to max_length_sequence := 0;
   signal counter_seq : integer range 0 to max_length_sequence-1 := 0;
   signal counter_display : integer range 0 to 31 := 0;

   signal random : integer range 0 to 3;

   signal new_user_command : boolean := true;

begin
   random_generator : process(clock0)
   begin
      if( rising_edge(clock0) ) then
         if( random = 3 ) then
            random <= 0;
         else
            random <= random + 1;
         end if;
      end if;
   end process random_generator;

   state_machine : process(clock1)
   variable user_selection : integer range 0 to 3;
   begin
      if( rising_edge(clock1) ) then
```

FIGURE 8 – Updated entity/architecture header

An example of updated state machine process is represented in Figure 9.

Note that, in the current design, if the user presses 2 buttons, only the first one pushed will be interpreted. This is an arbitrary choice of program configuration. However, you should not allow a sequence of inputs to cause an error in the execution of your game. Your implementation must be robust to all possible input signals.

```
state_machine : process(clock1)
variable user_selection : integer range 0 to 3;
begin
   if( rising_edge(clock1) ) then
      case state is
         when add_random =>
            state <= display;
            sequence(length_sequence) <= random; --The random number is added to the end of the sequence
            length_sequence <= length_sequence + 1; --The size of the sequence is incremented
            play <= '0';

         when display =>
            leds <= "0000"; --All the LEDs are off if counter_display < 8
            if( counter_display > 7 ) then
               leds(sequence(counter_seq)) <= '1'; --The LED of the sequence at index counter_seq is on

               if( counter_display = 31 ) then --End of display of one LED of the sequence
                  counter_display <= 0;
                  if( counter_seq = length_sequence - 1) then --End of the sequence
                     counter_seq <= 0;
                     state <= sequence_verification;
                  else --Need to display to following element of the sequence
                     counter_seq <= counter_seq + 1;
                  end if;
               else
                  counter_display <= counter_display + 1;
               end if;
            else
               counter_display <= counter_display + 1;
            end if;

         when sequence_verification =>
            play <= '1';
            leds <= buttons; --Display of the LEDs when the player presses a button
            if( buttons = "0000" ) then --No button pressed by the player
               new_user_command <= true;
            elsif( new_user_command ) then --New signal send by the player
               new_user_command <= false;
               if( buttons(0) = '1' ) then user_selection := 0;
               elsif( buttons(1) = '1' ) then user_selection := 1;
               elsif( buttons(2) = '1' ) then user_selection := 2;
               elsif( buttons(3) = '1' ) then user_selection := 3;
               end if;  --Button signals interpretation

               if( user_selection = sequence(counter_seq) ) then --Verification of the chosen color
                  if( counter_seq = length_sequence - 1 ) then --All the sequence is verified
                     counter_seq <= 0;
                     if( length_sequence = max_length_sequence ) then --We reached the maximum sequence length: game won
                        state <= done;
                     else --We add an element to the sequence
                        state <= add_random;
                     end if;
                  else --The whole sequence isn't verified, we check the next color
                     counter_seq <= counter_seq + 1;
                  end if;
               else --The player is wrong
                  counter_seq <= 0;
                  state <= game_over;
               end if;
            end if;
```

FIGURE 9 – Updated state machine process

3.4 End game and Game over states

Finally, one needs to code the **End game** and **Game over** states. In these states, it is necessary to wait that the user presses the reset button to go back to the **Add random color** state. In this design, it has been designed to implement a very easy animation to inform the user if he lost or won. Note that the choice of animations for these states are arbitrary. A more complex animation taking place on several clock cycles could have been implemented. In addition, a 7-segment display could have been used to show the size of the current sequence or the level of the game. An example of architecture of the **End game** and **Game over** states is represented in Figure 10.

```
                when game_over =>
                    leds <= "1010"; --Arbitrary animation of the game over
                    if( reset = '1' ) then
                        state <= add_random;
                        length_sequence <= 0;
                        counter_seq <= 0;
                    end if;

                when done =>
                    leds <= "1111"; --Arbitrary animation of the game is won
                    if( reset = '1' ) then
                        state <= add_random;
                        length_sequence <= 0;
                        counter_seq <= 0;
                    end if;

            end case;
        end if;
    end process state_machine;
end architecture game_simon_architecture;
```

FIGURE 10 – Architecture of the **End game** and **Game over** states

Conclusion

The final code of this Simon game example is contained in the file *Simon_game.vhd* in myUliege. This file gives you a nice idea of how to start and tackle such kind of project. I invite you to follow the same kind of steps when designing your own project. Good luck and enjoy, this is probably the coolest project of bachelor you might have !