ECE 410 ADVANCED DIGITAL LOGIC DESIGN

LAB 1: Finite State Machines (FSMs)and Parkade Design Using VHDL

LAB SECTION D51

PARTNER 1: MADEEHA Anjum, 1514645

PARTNER 2: ABUNI Gaiya, 1544553

Introduction

A Finite is a model of computation based on a hypothetical machine made of one or more states. It is a machine that must transition from one state to another in to perform different actions. A finite state machine (FSM) can have both Moore and Mealy outputs. Moore outputs depend only on the present state while mealy depends on both the present state and at least one of the present inputs. In this lab we would be implementing a finite state machine that has Moore outputs.

Abstract

In this lab, we designed a four Moore Finite State Machines using VHDL. There are several reasonable ways of implementing finite-state machines as a VHDL simulation model. However, it is important to pick an FSM style that is acceptable by the likely synthesis tool. In this lab we used Behavioural code to describe our finite state machines. We were required to implement FSM designs that could detect the sequences of "01101", "1010011" and "ATTCGC". In each of the three FSM we used case statements to detect what states where were in and then we implemented what the next state we would be going into was using if else statements.

The fourth finite state machine was a more realistic and complicated one because it mimicked an actual parkade system at the University of Alberta. In the part of the lab we broke the code down into five states:

i) Case where we need to add one more available space.
ii) Case where we need to add two more available spaces.
iii) Case where we need to subtract one space from the available spaces
iv) Case where we need to subtract two spaces from the available spaces
v) Case where we need do not need to change anything on the available spaces

For each case we must find all the conditions where this would apply and the exceptions for each case for example, we would not keep subtracting available spaces when the available space is already 0. We would talk more on our implementation in the design section.

Design / Testing/ Discussion:

<u>**PART #1: SEQUENCE DETECTOR for** "01101"</u>

Design: In part 1 of the lab, we were required to implement a finite state machine to detect the bit sequence "01101" and run the program on the Zybo Z7 board. We used the finite state Machine diagram that we designed in the prelab(*Figure 1: A state diagram for a Moore FSM for the sequence "01101)* to implement this program. We used case statements to detect state our FSM was in and then we implemented what the next state we would be going into using if else statements (*Figure 4: Design for sequence "01101*). After the program had been implemented, we programmed the Zybo Z7 board using Led lights to display the next state and using the switches to change the inputs into the FSM. We were required to replace the manual clock button with one that would automatically update the states on the given input by using the clock divider component from the tutorial lab. The way the program was designed it would continuously loop and check the case statements acting where needed depending on the data input. When the sequence hit the final state, the LED was used to indicate that we had hit out final sequence bit. In addition to handle any errors there was a default state. We also have a VHDL schematic (*Figure 5: schematic).* **Testing:** The following results were obtained from testing (*Figure 2: Sequence "01101" simulation, Figure 3: Testbench for "01101)*. To test the sequence detector the testbench provided was used to test the correct sequence as well as an extra bit. After verifying the simulation, we adjusted the constraint file and tested our simulation on the board using the generated bitstream. Both results were successful. **Discussion** Both results were successful as the board in put switches were used to provide a bit of inputs and the resulting LEDs would show the corresponding next states. The waveform would show the data going into and the data coming out and the also the next states.

PART#2: SEQUENCE DETECTOR for "1010011"

Design This part of the lab was basically a replica or addition of states from part 1 as we were designing a finite state machine with as a different sequence, namely "1010011" In order to do this, we had to construct a state diagram for the Moore FSM implementation of this sequence, as well as having 2 more states of 5 states. (*Figure 6 A state diagram for a Moore FSM for the sequence "1010011"*). We used a very similar design style, same case statements and same default states (*Figure 7: Design for sequence "1010011")*. Upon reaching the final state the LED was turned on. We also have a VHDL schematic *(Figure 9: schematic)*. **Testing*:*** The following results were obtained from testing (Figure *8: Testbench, Figure 10: Sequence "1010011" simulation)*. To test the sequence detector the testbench provided was used to test the correct sequence as well as an extra bit. After verifying the simulation, we adjusted the constraint file and tested our simulation on the board using the generated bitstream. Both results were successful. **Discussion** Both results were successful as the board in put switches were used to provide a bit of inputs and the resulting LEDs would show the corresponding next states. The waveform would show the data going into and the data coming out and the also the next states.

PART #3: DNA SEQUENCE DETECTOR

Design For part 3 of the lab we implemented a DNA sequence detector. This required their to four input options provided (A, T, C, G) instead of just 0 and 1. These inputs were represented with two bits "00", "01", "10", and "11", respectively and therefore used two switches to collect the input values.(*Figure 11 A state diagram for the sequence "ATTCGC"*). We followed the steps and skills gained in part 1 and 2 to implement this design (*Figure 12 Design for sequence "ATTCGC")*. In addition, the clock was used again and the case statements and the default state and final Led being lit. We also have a VHDL schematic *(Figure 13: schematic)*. **Testing:** The following results were obtained from testing (*Figure 14: Testbench, Figure 15: Sequence "ATTCGC"" simulation). To test the sequence detector the testbench provided was used to test the correct sequence as well as an extra bit. After verifying the simulation, we adjusted the constraint file and tested our simulation on the board using the generated bitstream. Both results were successful.* **Discussion** Both results were successful as the board in put switches were used to provide a flow of bits and the resulting LEDs would show the corresponding next states. The waveform would show the data going into and the data coming out and the also the next states.

PART #4: PARKADE DESIGN

Design For part 4 of the lab we went through the given code and implemented a parkade design using the knowledge gained in parts 1 through 3. The count incremented whenever a car left the parking lot and decreased whenever a car entered. The display had 3 LEDs and we used a 7 segment to display the available space modifying it based upon available space, most significant bit/least significant bit. (total99 spaces). Updating ever two seconds. The last 2 switches were used to indicate 2 cars leaving or on entering. For example, having two switches on for both incrementing and decrementing would result in no changes and all the different case where accounted for in total we had 16 different cases, in addition we used the same default sates using the statement "When other". (*Figure 16.1 Design, Figure 16.2 Design, Figure 16.3 Design, Figure 17 Schematic)* **Testing (9 spaces used for testing)** The following results were obtained from testing (*Figure 18: Testbench, Figure 19: Parkade simulation)*. *To test the sequence detector the testbench provided was used to test the correct sequence as well as an extra bit. After verifying the simulation, we adjusted the constraint file and tested our simulation on the board using the generated bitstream.* **Discussion** Both results were successful as the board in put switches were used to provide a flow of bits and the resulting LEDs and segment would chance according to the design would show the corresponding next states. The waveform would show the data going into and the data coming out and the also the next states.

Conclusion

In part one of the lab we used the code provided to generate a simulation and program the board. In the second part of this lab we designed our own Moore FSM to detect the sequence "1010011", In the third part of this lab we implemented the sequence detector for the sequence "ATTCGC", where A, T, C, and G represent the bit sequences "00", "01", "10", and "11". Finally, in the last part of this lab we went through the given code and implemented a parkade design. In these parts the generated bitstream tested on the board worked very well along with the simulation. There were no issues for each sequence detector, but some struggles on writing testbenches. In concluded in this lab we were able to achieve the objective of leaning how to use case statements and implementing and design various FSMs having part one as a base and building onto out code we were able to achieve the design of parkade.

Table of Contents for Figures:

PART #1: SEQUENCE DETECTOR for “01101”

Figure 2: A state diagram for a Moore FSM for the sequence “01101

Figure 2: Sequence “01101” simulation

Figure 3: Testbench for “01101

Figure 4: Design for sequence “01101

Figure 5: schematic

PART#2: **SEQUENCE DETECTOR for** “1010011”

Figure 6 A state diagram for a Moore FSM for the sequence “1010011”

Figure 7: Design for sequence “1010011”

Figure 8: Testbench

Figure 9: Schematic

Figure 10: Sequence “1010011” simulation

PART #3: DNA SEQUENCE DETECTOR

Figure 11 A state diagram for the sequence “ATTCGC”

Figure 12 Design for sequence “ATTCGC”

Figure 13: Schematic

Figure 14: Testbench

Figure 15: Sequence “ATTCGC”” simulation

PART #4: PARKADE DESIGN

Figure 16.1 Design

Figure 16.2 Design

Figure 16.3 Design

Figure 17 Schematic

Figure 18: Testbench

Figure 19: Parkade simulation

<u>**PART #1: SEQUENCE DETECTOR for** “01101”</u>

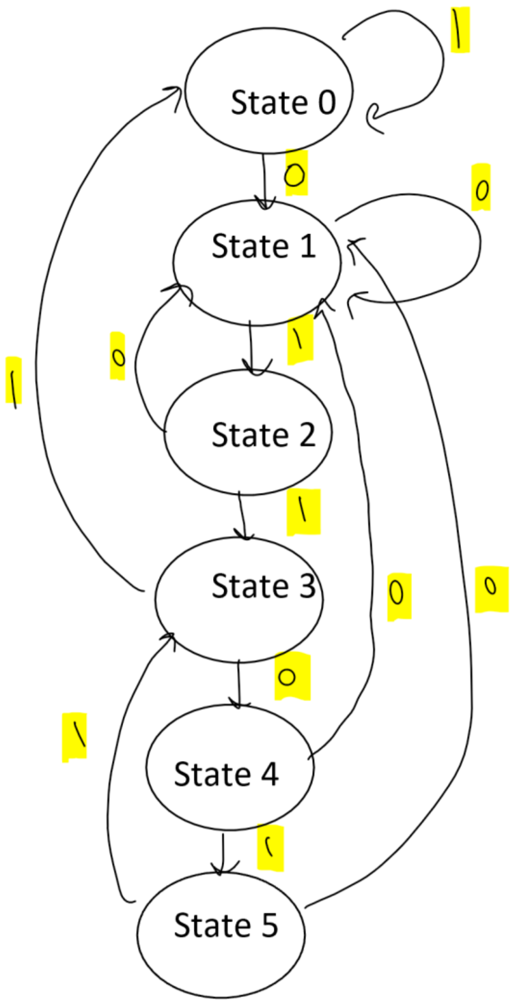
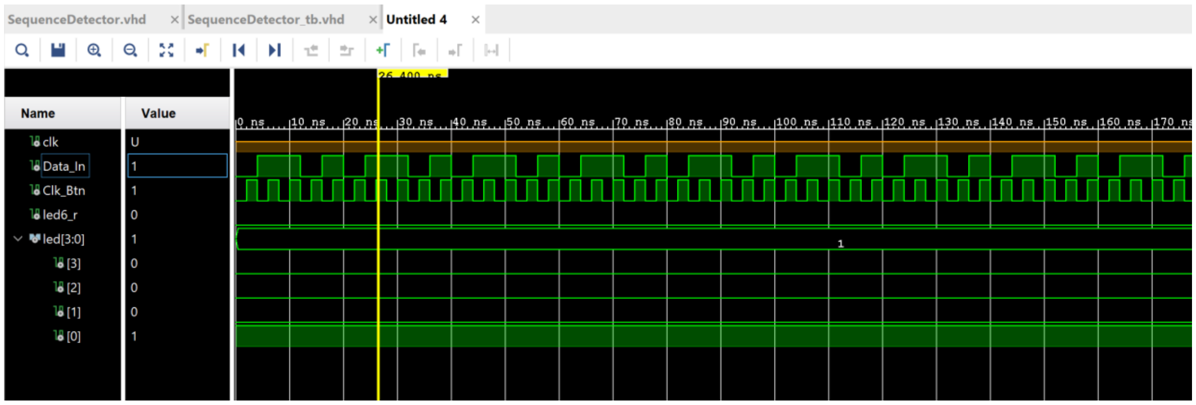
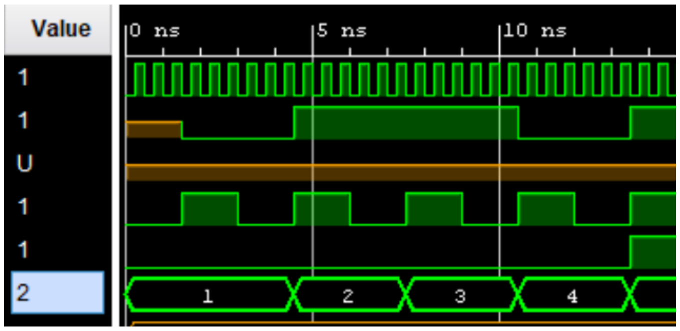


Figure 1: A state diagram for a Moore FSM for the sequence “01101”

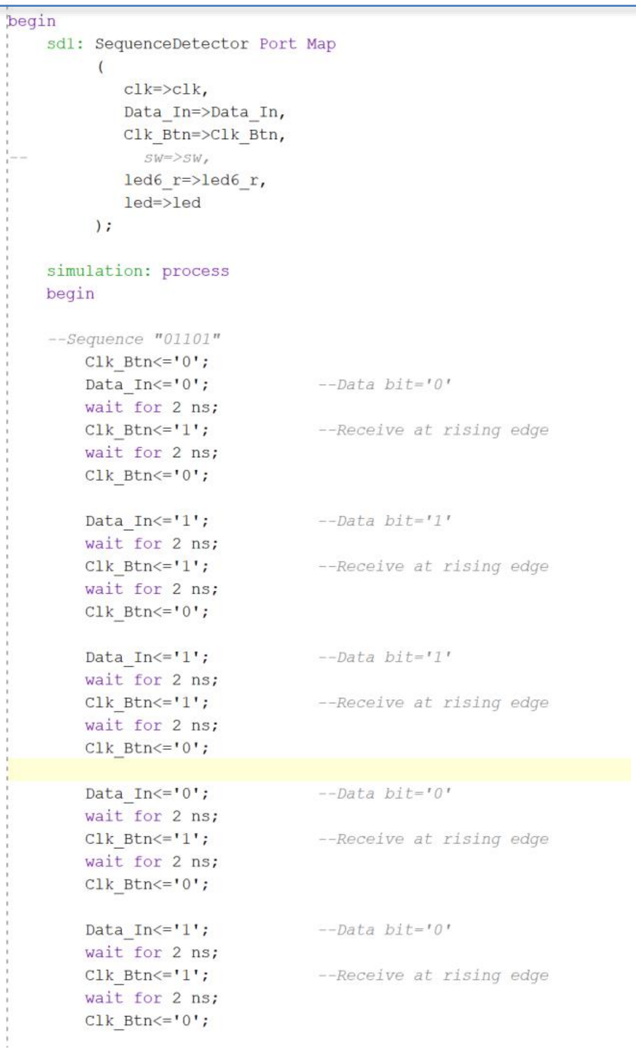


Name	Value
clk	U
Data_In	1
Clk_Btn	1
led6_r	0
led[3:0]	1
[3]	0
[2]	0
[1]	0
[0]	1



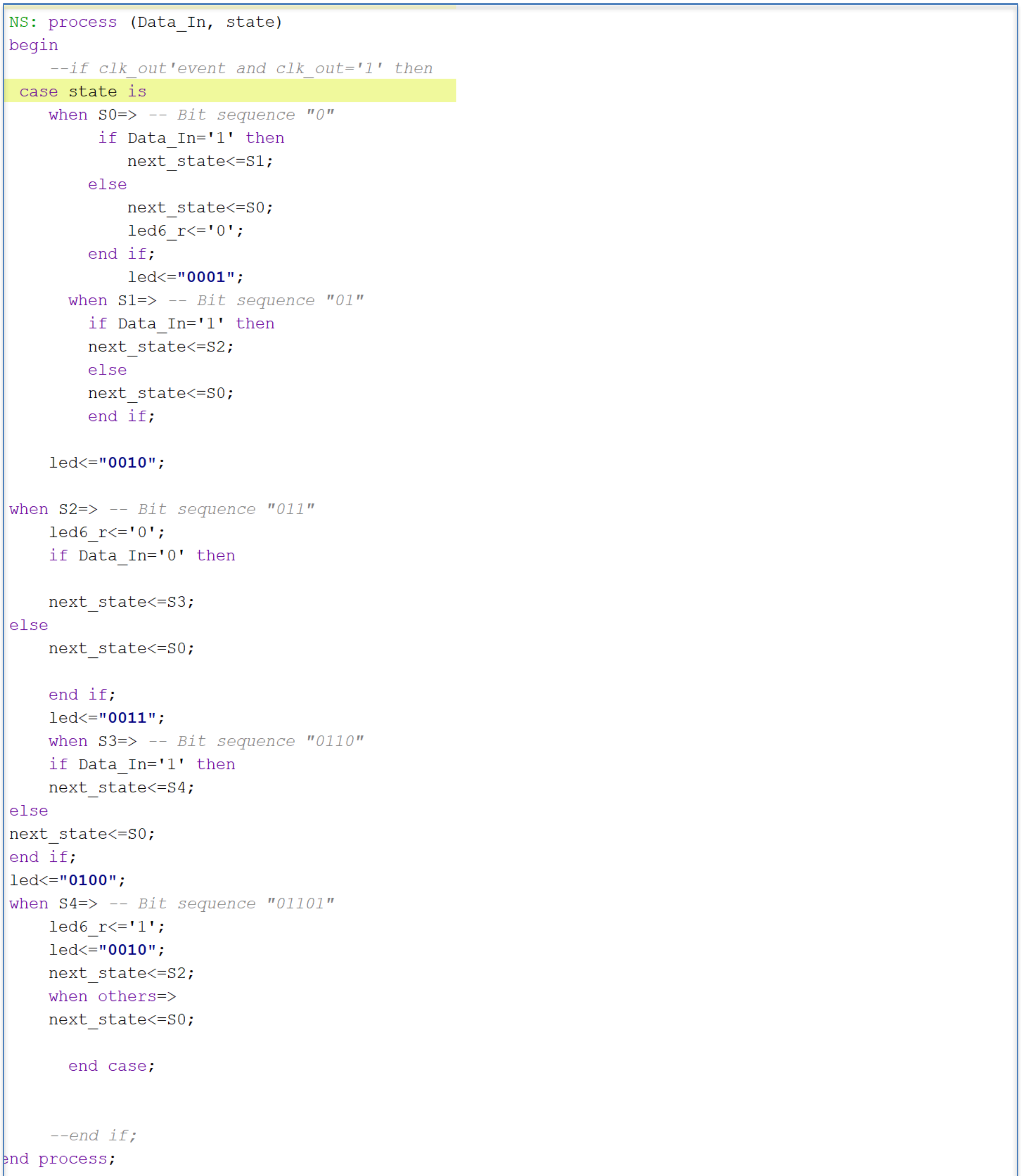
Value
1
1
U
1
1
2

Figure 2: Sequence “01101” simulation



```
begin
    sd1: SequenceDetector Port Map
         (
           clk=>clk,
           Data_In=>Data_In,
           Clk_Btn=>Clk_Btn,
--           sw=>sw,
           led6_r=>led6_r,
           led=>led
         );

    simulation: process
    begin

    --Sequence "01101"
        Clk_Btn<='0';
        Data_In<='0';           --Data bit='0'
        wait for 2 ns;
        Clk_Btn<='1';           --Receive at rising edge
        wait for 2 ns;
        Clk_Btn<='0';

        Data_In<='1';           --Data bit='1'
        wait for 2 ns;
        Clk_Btn<='1';           --Receive at rising edge
        wait for 2 ns;
        Clk_Btn<='0';

        Data_In<='1';           --Data bit='1'
        wait for 2 ns;
        Clk_Btn<='1';           --Receive at rising edge
        wait for 2 ns;
        Clk_Btn<='0';

        Data_In<='0';           --Data bit='0'
        wait for 2 ns;
        Clk_Btn<='1';           --Receive at rising edge
        wait for 2 ns;
        Clk_Btn<='0';

        Data_In<='1';           --Data bit='0'
        wait for 2 ns;
        Clk_Btn<='1';           --Receive at rising edge
        wait for 2 ns;
        Clk_Btn<='0';
```

Figure 3: Testbench for “01101



```
NS: process (Data_In, state)
begin
    --if clk_out'event and clk_out='1' then
 case state is
    when S0=> -- Bit sequence "0"
         if Data_In='1' then
            next_state<=S1;
        else
            next_state<=S0;
            led6_r<='0';
        end if;
            led<="0001";
      when S1=> -- Bit sequence "01"
        if Data_In='1' then
        next_state<=S2;
        else
        next_state<=S0;
        end if;

    led<="0010";

when S2=> -- Bit sequence "011"
    led6_r<='0';
    if Data_In='0' then

    next_state<=S3;
else
    next_state<=S0;

    end if;
    led<="0011";
    when S3=> -- Bit sequence "0110"
    if Data_In='1' then
    next_state<=S4;
else
next_state<=S0;
end if;
led<="0100";
when S4=> -- Bit sequence "01101"
    led6_r<='1';
    led<="0010";
    next_state<=S2;
    when others=>
    next_state<=S0;

      end case;


    --end if;
end process;
```

Figure 4: Design for sequence “01101

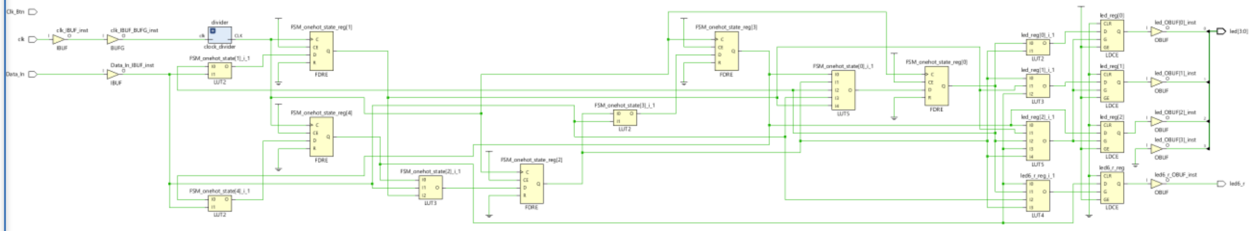


Figure 5: schematic

PART#2: SEQUENCE DETECTOR for “1010011”

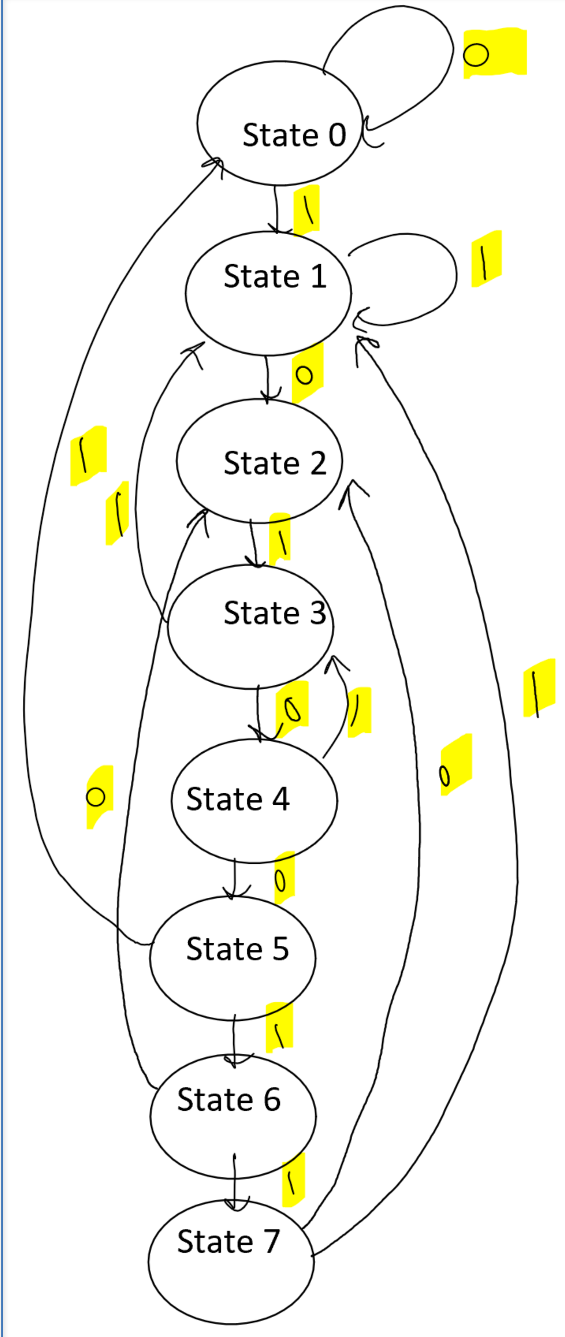
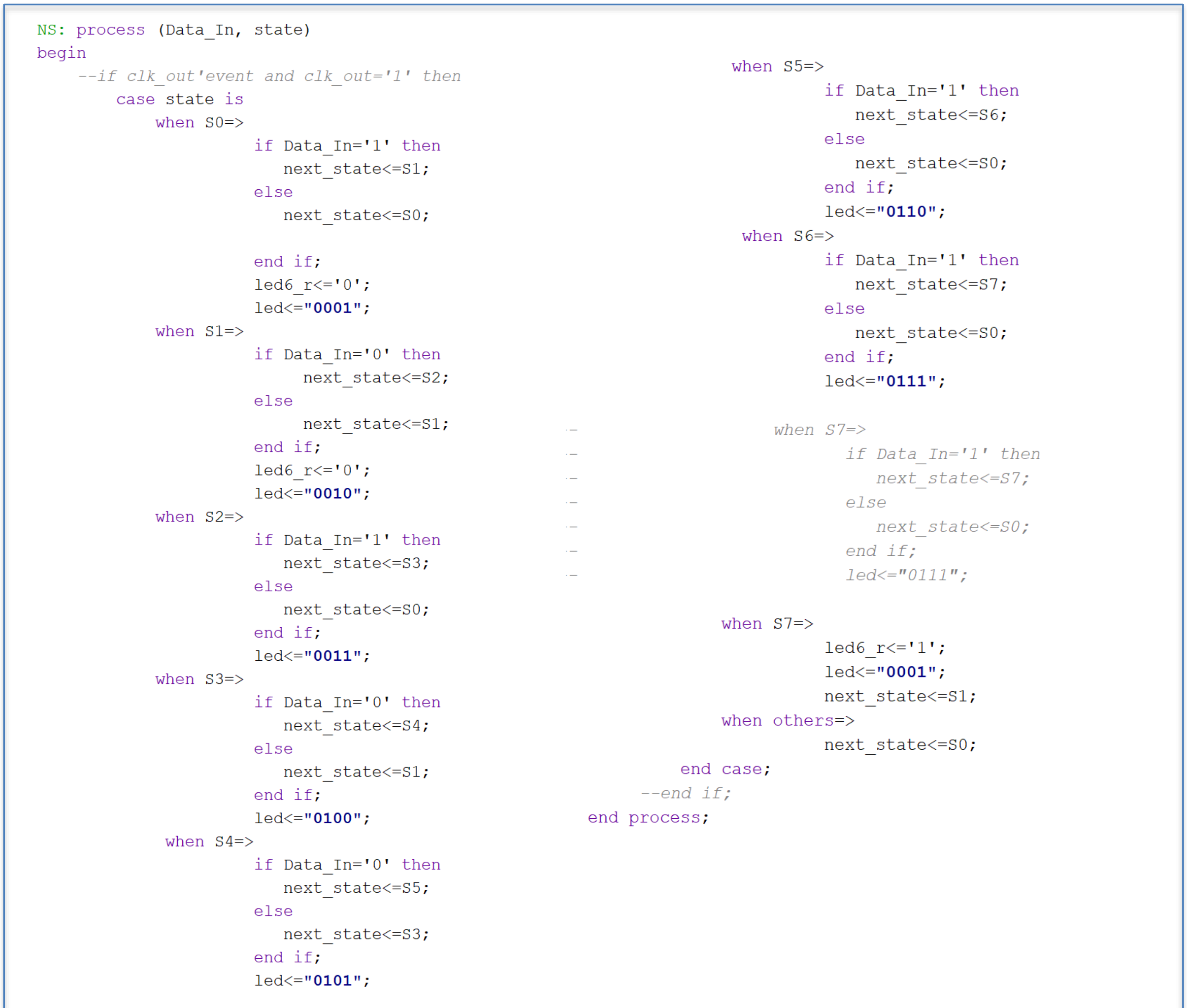


Figure 6 A state diagram for a Moore FSM for the sequence “1010011”



```
NS: process (Data_In, state)
begin
    --if clk_out'event and clk_out='1' then
        case state is
            when S0=>
                    if Data_In='1' then
                       next_state<=S1;
                    else
                       next_state<=S0;

                    end if;
                    led6_r<='0';
                    led<="0001";
            when S1=>
                    if Data_In='0' then
                         next_state<=S2;
                    else
                         next_state<=S1;
                    end if;
                    led6_r<='0';
                    led<="0010";
            when S2=>
                    if Data_In='1' then
                       next_state<=S3;
                    else
                       next_state<=S0;
                    end if;
                    led<="0011";
            when S3=>
                    if Data_In='0' then
                       next_state<=S4;
                    else
                       next_state<=S1;
                    end if;
                    led<="0100";
             when S4=>
                    if Data_In='0' then
                       next_state<=S5;
                    else
                       next_state<=S3;
                    end if;
                    led<="0101";
            when S5=>
                    if Data_In='1' then
                       next_state<=S6;
                    else
                       next_state<=S0;
                    end if;
                    led<="0110";
             when S6=>
                    if Data_In='1' then
                       next_state<=S7;
                    else
                       next_state<=S0;
                    end if;
                    led<="0111";

--           when S7=>
--                  if Data_In='1' then
--                     next_state<=S7;
--                  else
--                     next_state<=S0;
--                  end if;
--                  led<="0111";

            when S7=>
                    led6_r<='1';
                    led<="0001";
                    next_state<=S1;
            when others=>
                    next_state<=S0;
        end case;
    --end if;
end process;
```

Figure 7: Design for sequence “1010011”

```vhdl
simulation: process
begin

--Sequence "01101"
    Clk_Btn<='0';
    Data_In<='0';           --Data bit='0'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='1'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='1'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='0';           --Data bit='0'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='0'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

--Sequence "0101101"
    Clk_btn<='0';
    Data_In<='0';           --Data bit='0'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='1'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='0';           --Data bit='0'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='1'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='1'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='0';           --Data bit='0'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';

    Data_In<='1';           --Data bit='1'
    wait for 2 ns;
    Clk_Btn<='1';           --Receive at rising edge
    wait for 2 ns;
    Clk_Btn<='0';


end process;
```

Figure 8: Testbench

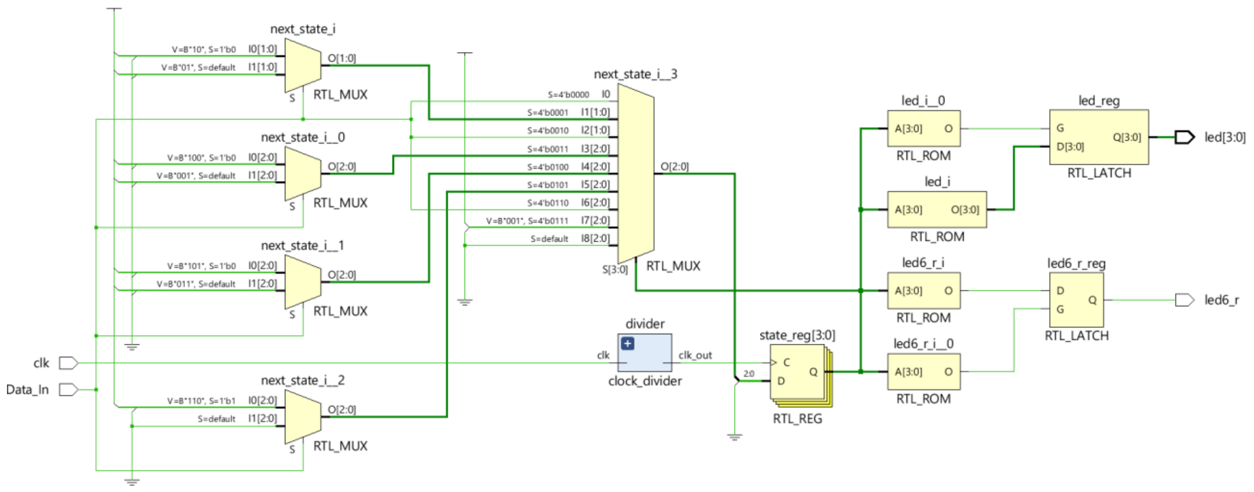
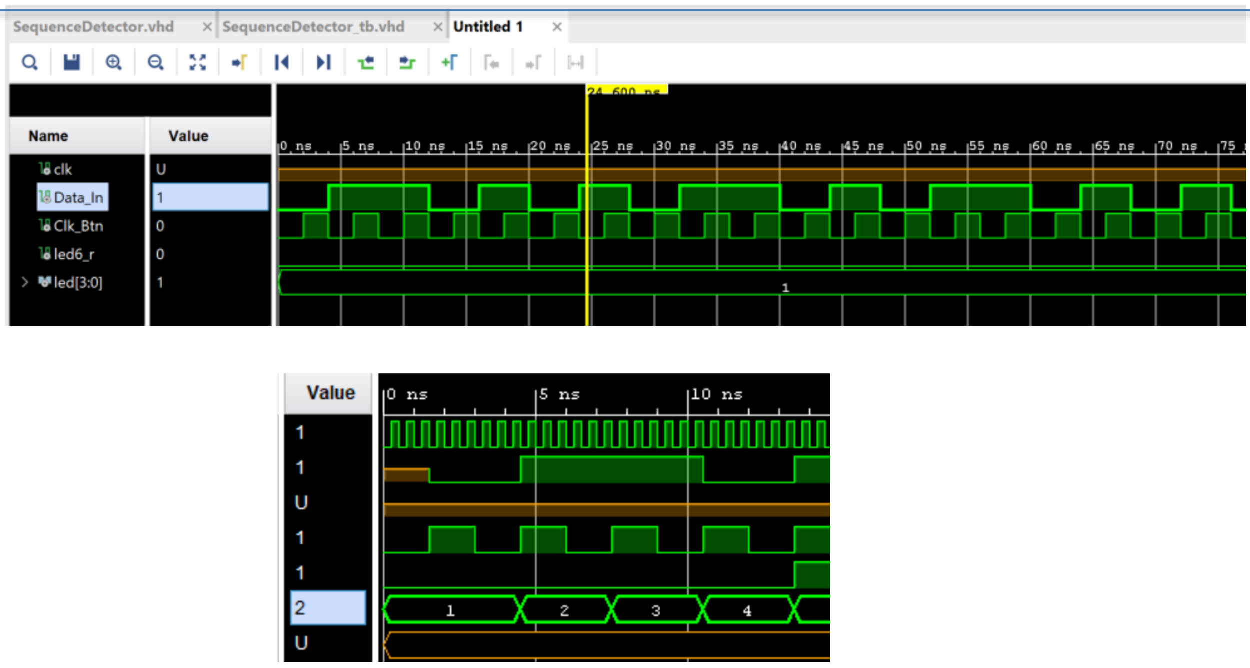


Figure 9: Schematic



Name	Value
clk	U
Data_In	1
Clk_Btn	0
led6_r	0
led[3:0]	1

Figure 10: Sequence "1010011" simulation

PART #3: DNA SEQUENCE DETECTOR

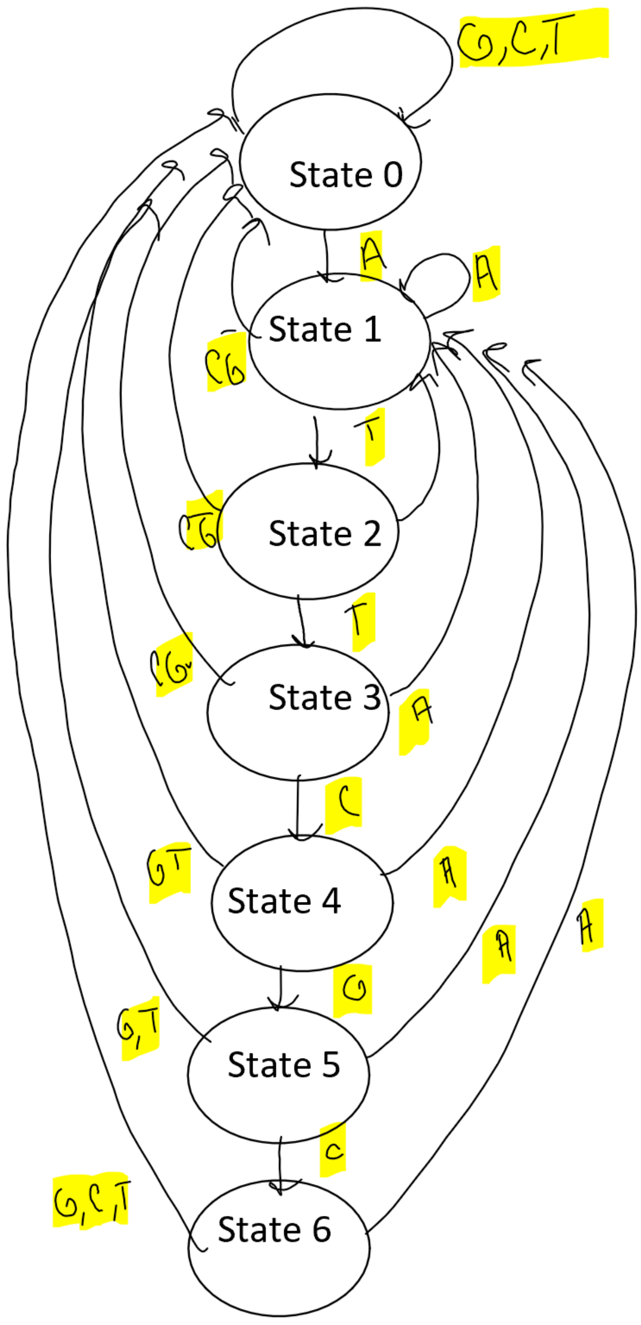
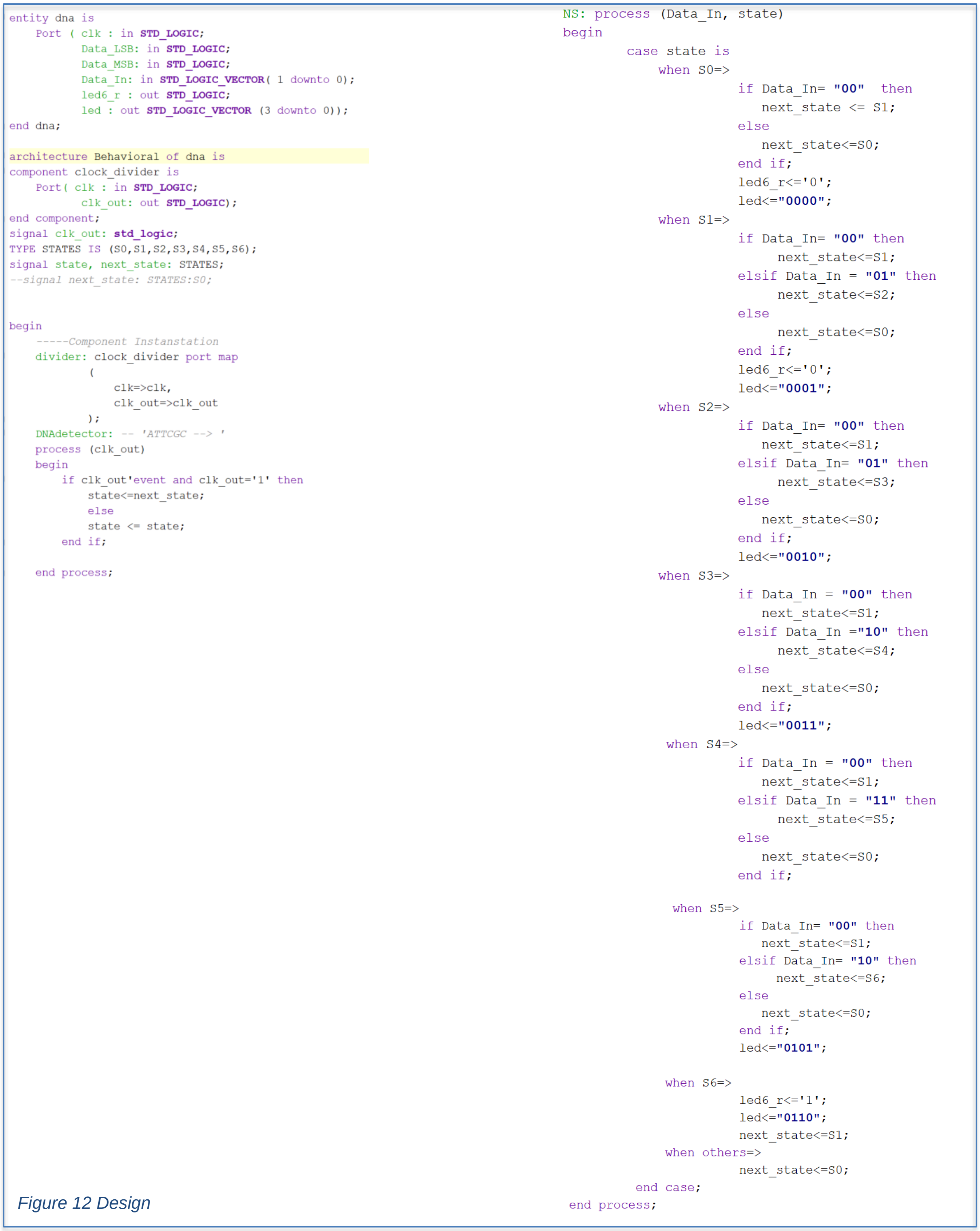


Figure 11 A state diagram for the sequence "ATTCGC"



```vhdl
entity dna is
    Port ( clk : in STD_LOGIC;
           Data_LSB: in STD_LOGIC;
           Data_MSB: in STD_LOGIC;
           Data_In: in STD_LOGIC_VECTOR( 1 downto 0);
           led6_r : out STD_LOGIC;
           led : out STD_LOGIC_VECTOR (3 downto 0));
end dna;

architecture Behavioral of dna is
component clock_divider is
    Port( clk : in STD_LOGIC;
           clk_out: out STD_LOGIC);
end component;
signal clk_out: std_logic;
TYPE STATES IS (S0,S1,S2,S3,S4,S5,S6);
signal state, next_state: STATES;
--signal next_state: STATES:S0;


begin
    -----Component Instanstation
    divider: clock_divider port map
            (
                clk=>clk,
                clk_out=>clk_out
            );
    DNAdetector: -- 'ATTCGC --> '
    process (clk_out)
    begin
        if clk_out'event and clk_out='1' then
            state<=next_state;
            else
            state <= state;
        end if;

    end process;
```

```vhdl
NS: process (Data_In, state)
begin
        case state is
            when S0=>
                        if Data_In= "00"  then
                           next_state <= S1;
                        else
                           next_state<=S0;
                        end if;
                        led6_r<='0';
                        led<="0000";
            when S1=>
                        if Data_In= "00" then
                             next_state<=S1;
                        elsif Data_In = "01" then
                             next_state<=S2;
                        else
                             next_state<=S0;
                        end if;
                        led6_r<='0';
                        led<="0001";
            when S2=>
                        if Data_In= "00" then
                           next_state<=S1;
                        elsif Data_In= "01" then
                             next_state<=S3;
                        else
                           next_state<=S0;
                        end if;
                        led<="0010";
            when S3=>
                        if Data_In = "00" then
                           next_state<=S1;
                        elsif Data_In ="10" then
                             next_state<=S4;
                        else
                           next_state<=S0;
                        end if;
                        led<="0011";
             when S4=>
                        if Data_In = "00" then
                           next_state<=S1;
                        elsif Data_In = "11" then
                             next_state<=S5;
                        else
                           next_state<=S0;
                        end if;

              when S5=>
                        if Data_In= "00" then
                           next_state<=S1;
                        elsif Data_In= "10" then
                             next_state<=S6;
                        else
                           next_state<=S0;
                        end if;
                        led<="0101";

             when S6=>
                        led6_r<='1';
                        led<="0110";
                        next_state<=S1;
             when others=>
                        next_state<=S0;
         end case;
end process;
```

Figure 12 Design

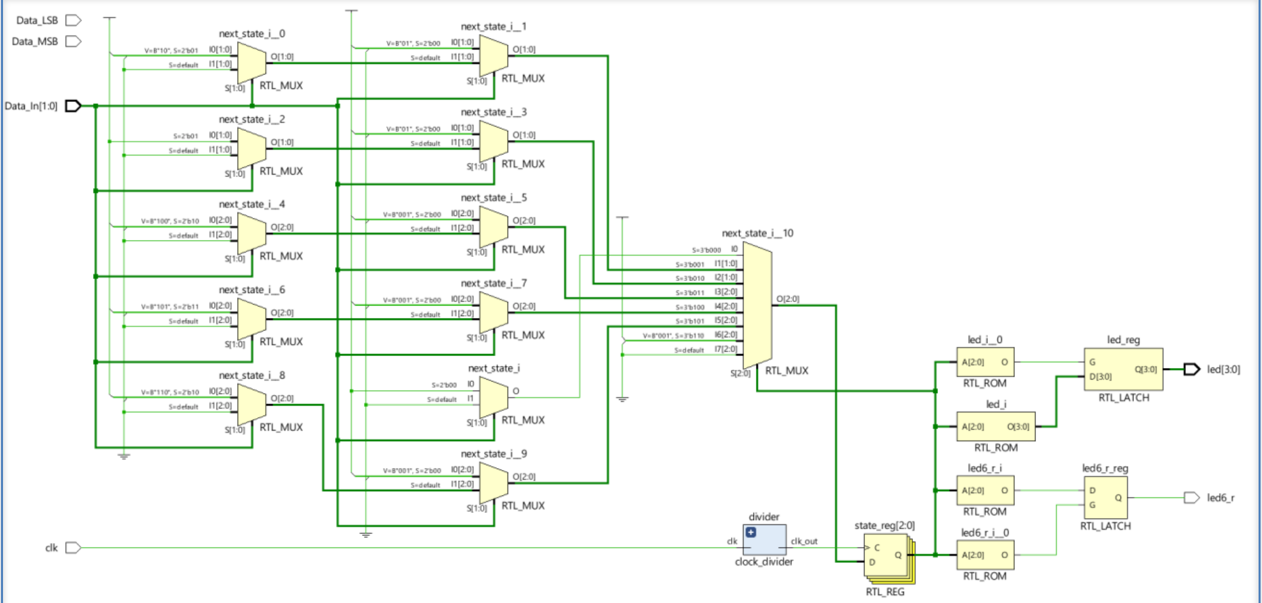
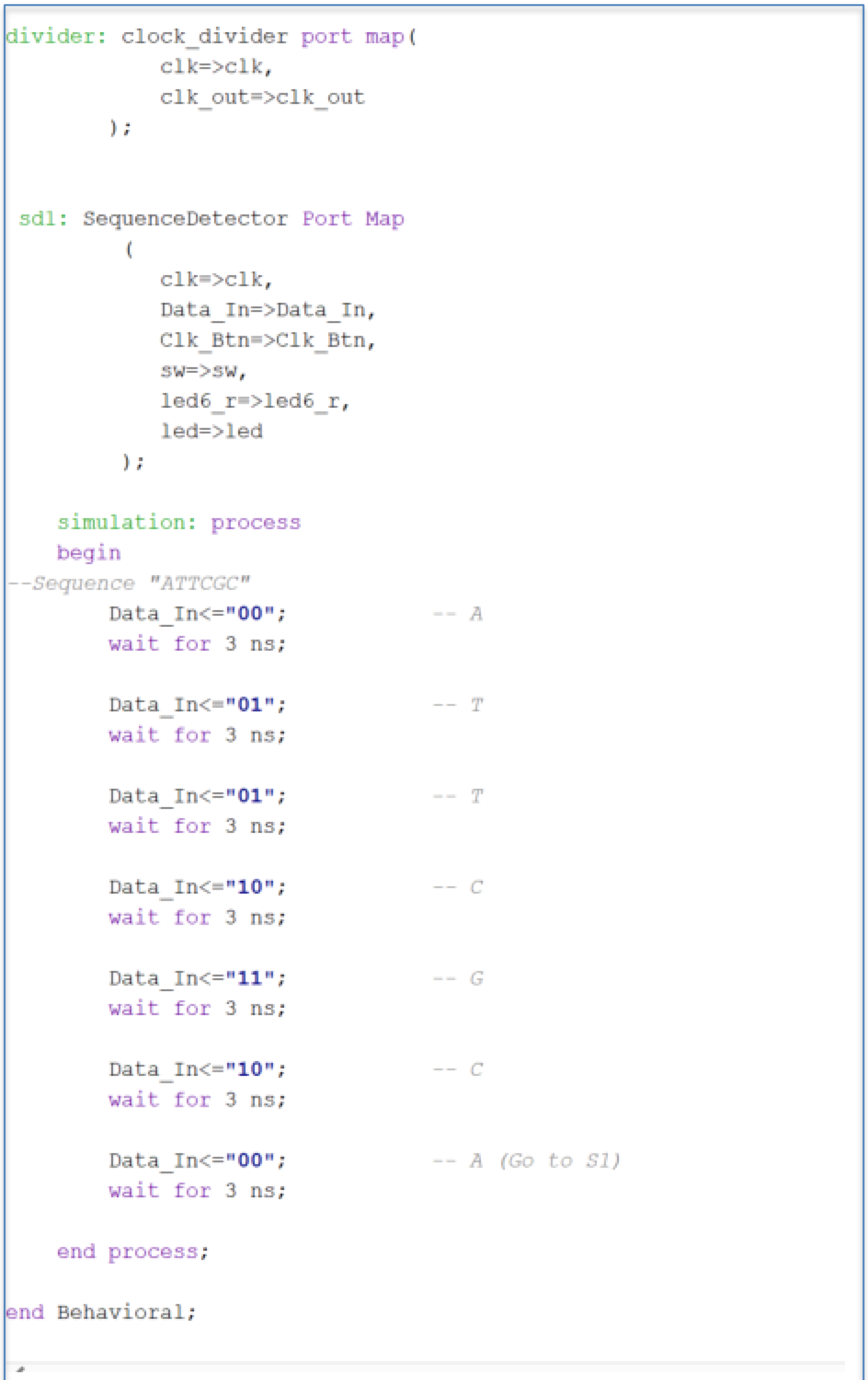


Figure 13: Schematic



```
divider: clock_divider port map(
            clk=>clk,
            clk_out=>clk_out
        );


 sdl: SequenceDetector Port Map
         (
            clk=>clk,
            Data_In=>Data_In,
            Clk_Btn=>Clk_Btn,
            sw=>sw,
            led6_r=>led6_r,
            led=>led
         );

    simulation: process
    begin
--Sequence "ATTCGC"
        Data_In<="00";           -- A
        wait for 3 ns;

        Data_In<="01";           -- T
        wait for 3 ns;

        Data_In<="01";           -- T
        wait for 3 ns;

        Data_In<="10";           -- C
        wait for 3 ns;

        Data_In<="11";           -- G
        wait for 3 ns;

        Data_In<="10";           -- C
        wait for 3 ns;

        Data_In<="00";           -- A (Go to S1)
        wait for 3 ns;

    end process;

end Behavioral;
```

Figure 14: Testbench

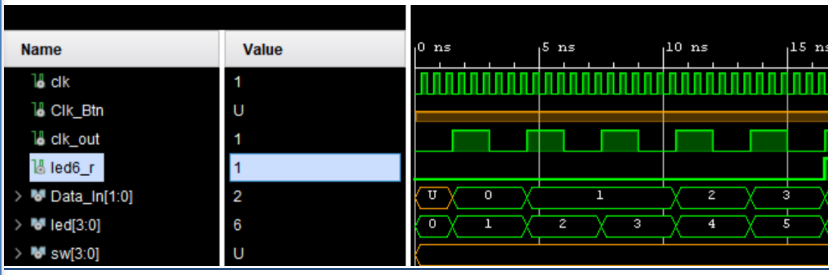
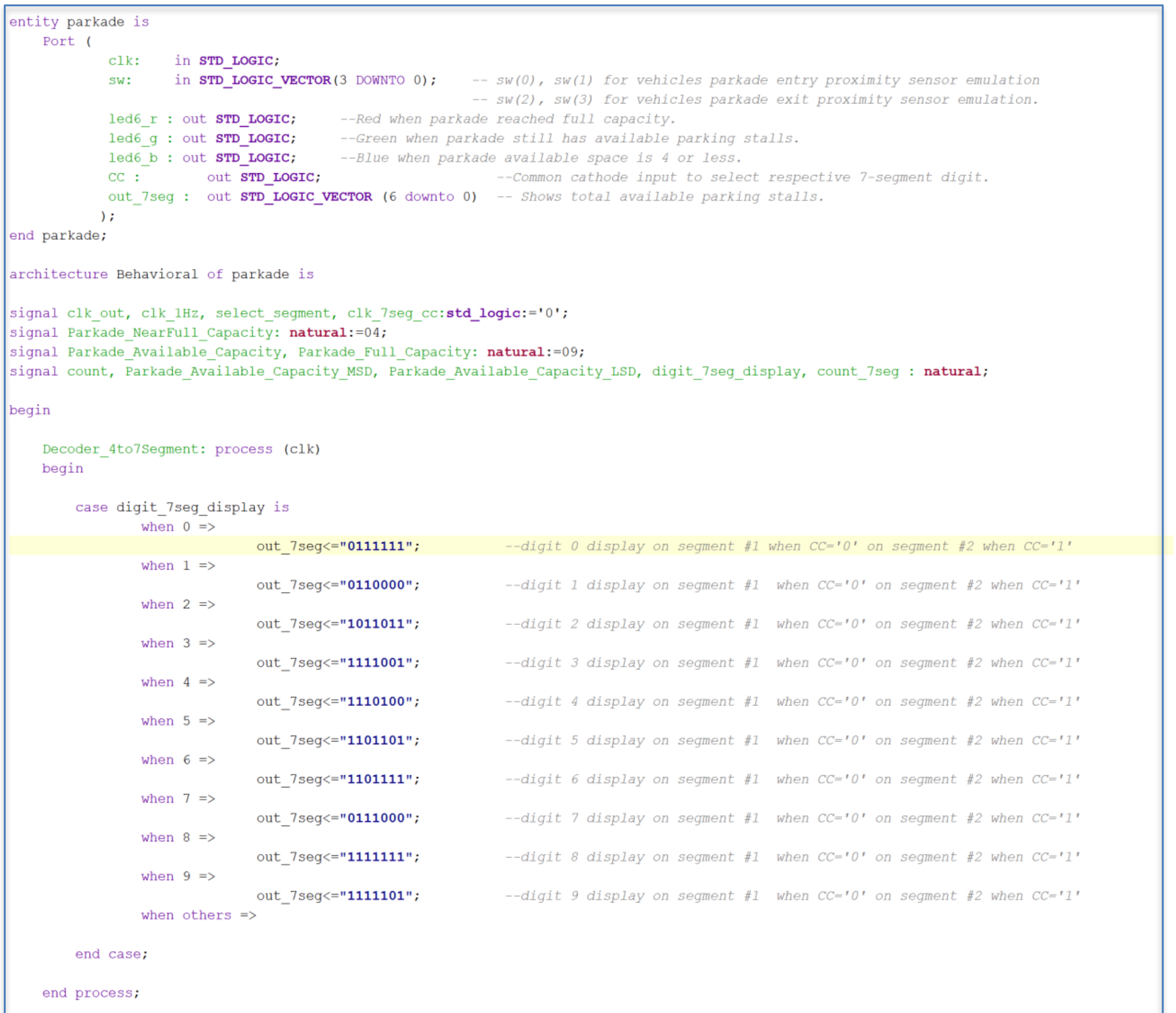


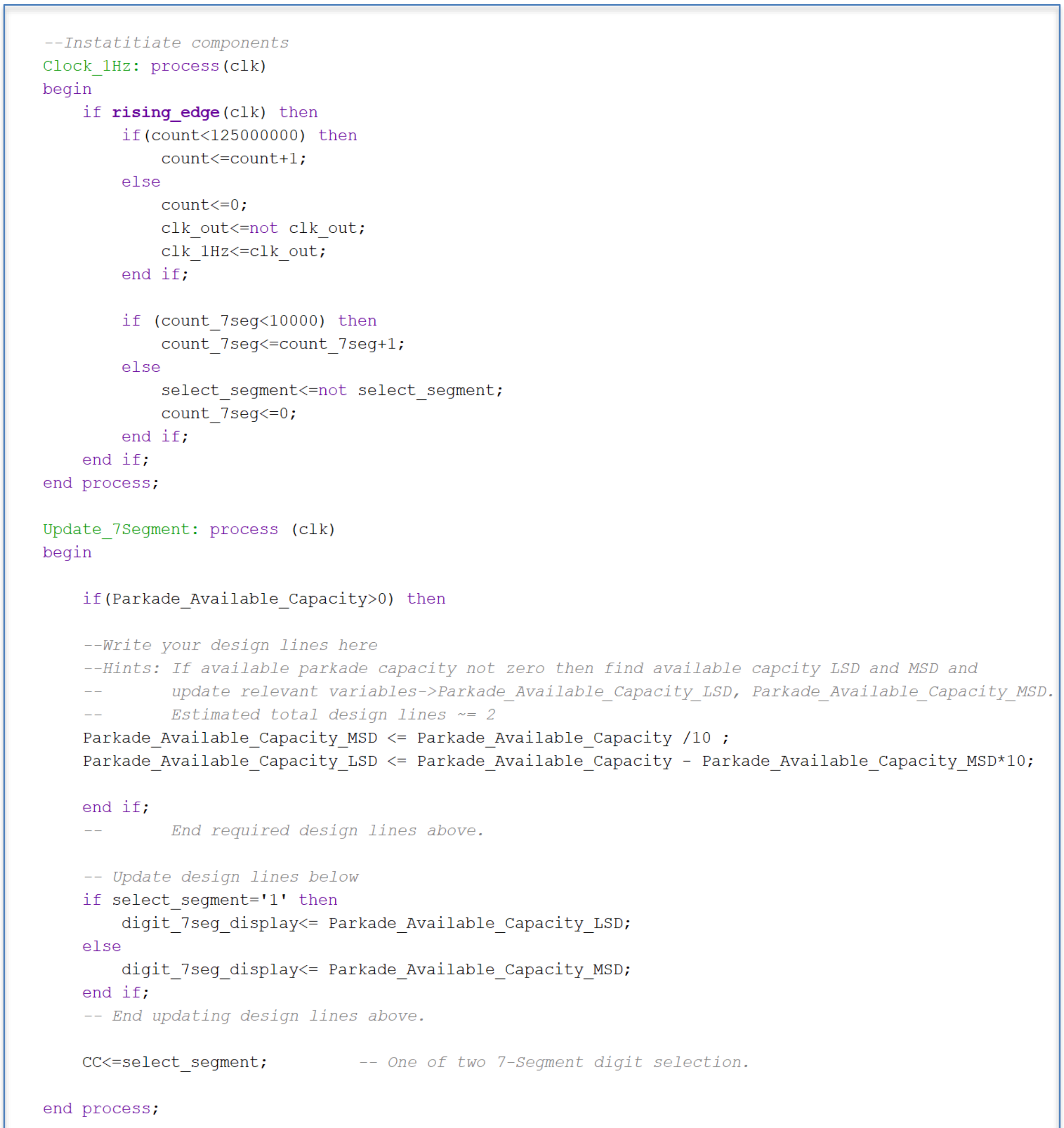
Figure 15: Sequence "ATTCGC"" simulation

PART #4: PARKADE DESIGN



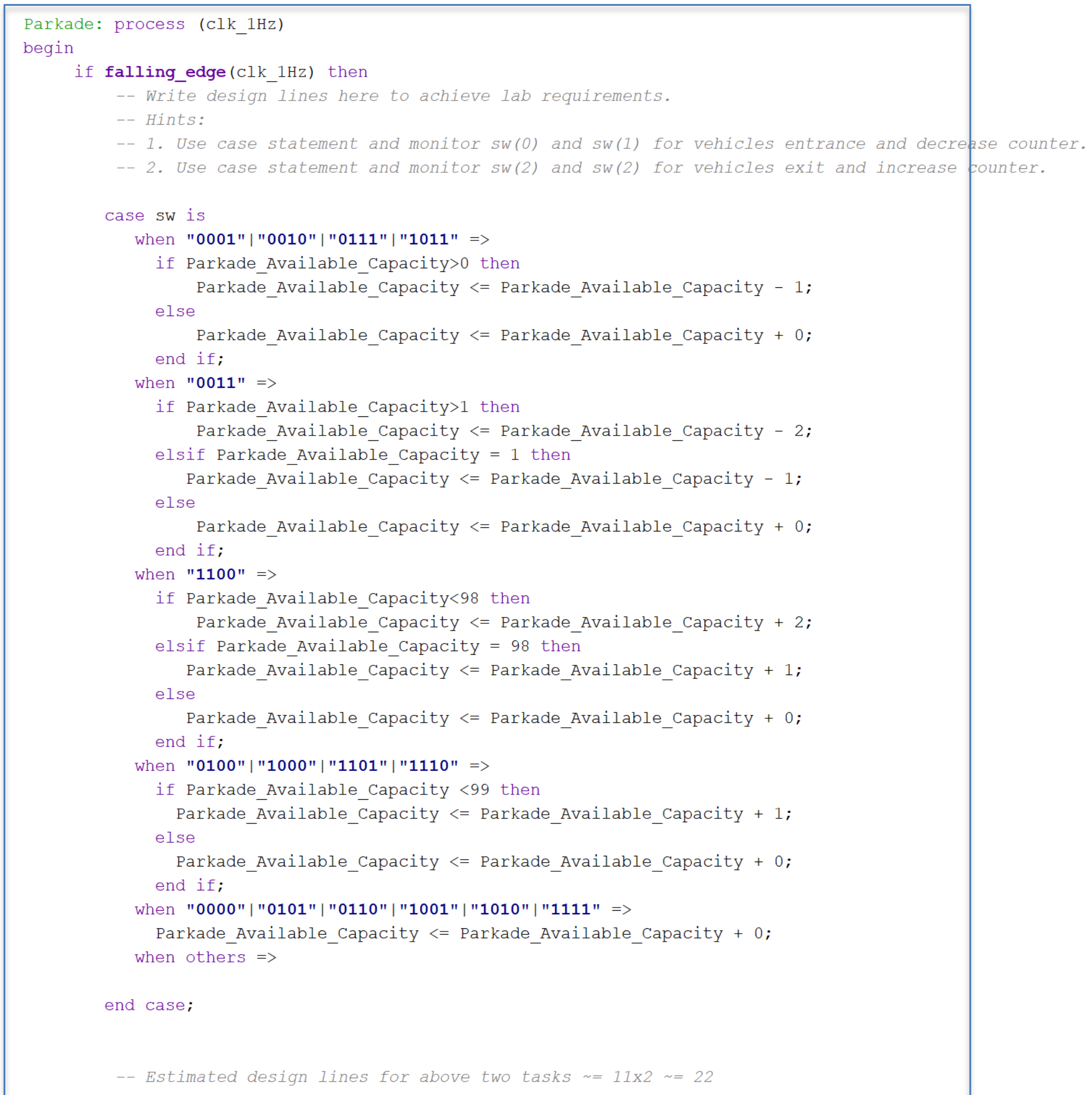
```
entity parkade is
    Port (
            clk:    in STD_LOGIC;
            sw:     in STD_LOGIC_VECTOR(3 DOWNTO 0);    -- sw(0), sw(1) for vehicles parkade entry proximity sensor emulation
                                                        -- sw(2), sw(3) for vehicles parkade exit proximity sensor emulation.
            led6_r : out STD_LOGIC;     --Red when parkade reached full capacity.
            led6_g : out STD_LOGIC;     --Green when parkade still has available parking stalls.
            led6_b : out STD_LOGIC;     --Blue when parkade available space is 4 or less.
            CC :        out STD_LOGIC;                  --Common cathode input to select respective 7-segment digit.
            out_7seg :  out STD_LOGIC_VECTOR (6 downto 0)  -- Shows total available parking stalls.
           );
end parkade;

architecture Behavioral of parkade is

signal clk_out, clk_1Hz, select_segment, clk_7seg_cc:std_logic:='0';
signal Parkade_NearFull_Capacity: natural:=04;
signal Parkade_Available_Capacity, Parkade_Full_Capacity: natural:=09;
signal count, Parkade_Available_Capacity_MSD, Parkade_Available_Capacity_LSD, digit_7seg_display, count_7seg : natural;

begin

    Decoder_4to7Segment: process (clk)
    begin

        case digit_7seg_display is
                when 0 =>
                              out_7seg<="0111111";          --digit 0 display on segment #1 when CC='0' on segment #2 when CC='1'
                when 1 =>
                              out_7seg<="0110000";          --digit 1 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 2 =>
                              out_7seg<="1011011";          --digit 2 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 3 =>
                              out_7seg<="1111001";          --digit 3 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 4 =>
                              out_7seg<="1110100";          --digit 4 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 5 =>
                              out_7seg<="1101101";          --digit 5 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 6 =>
                              out_7seg<="1101111";          --digit 6 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 7 =>
                              out_7seg<="0111000";          --digit 7 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 8 =>
                              out_7seg<="1111111";          --digit 8 display on segment #1  when CC='0' on segment #2 when CC='1'
                when 9 =>
                              out_7seg<="1111101";          --digit 9 display on segment #1  when CC='0' on segment #2 when CC='1'
                when others =>

        end case;

    end process;
```

Figure 16.1 Design



```
--Instatitiate components
Clock_1Hz: process(clk)
begin
    if rising_edge(clk) then
        if(count<125000000) then
            count<=count+1;
        else
            count<=0;
            clk_out<=not clk_out;
            clk_1Hz<=clk_out;
        end if;

        if (count_7seg<10000) then
            count_7seg<=count_7seg+1;
        else
            select_segment<=not select_segment;
            count_7seg<=0;
        end if;
    end if;
end process;

Update_7Segment: process (clk)
begin

    if(Parkade_Available_Capacity>0) then

    --Write your design lines here
    --Hints: If available parkade capacity not zero then find available capcity LSD and MSD and
    --        update relevant variables->Parkade_Available_Capacity_LSD, Parkade_Available_Capacity_MSD.
    --        Estimated total design lines ~= 2
    Parkade_Available_Capacity_MSD <= Parkade_Available_Capacity /10 ;
    Parkade_Available_Capacity_LSD <= Parkade_Available_Capacity - Parkade_Available_Capacity_MSD*10;

    end if;
    --        End required design lines above.

    -- Update design lines below
    if select_segment='1' then
        digit_7seg_display<= Parkade_Available_Capacity_LSD;
    else
        digit_7seg_display<= Parkade_Available_Capacity_MSD;
    end if;
    -- End updating design lines above.

    CC<=select_segment;          -- One of two 7-Segment digit selection.

end process;
```

Figure 16.2 Design



```
Parkade: process (clk_1Hz)
begin
     if falling_edge(clk_1Hz) then
         -- Write design lines here to achieve lab requirements.
         -- Hints:
         -- 1. Use case statement and monitor sw(0) and sw(1) for vehicles entrance and decrease counter.
         -- 2. Use case statement and monitor sw(2) and sw(2) for vehicles exit and increase counter.

        case sw is
           when "0001"|"0010"|"0111"|"1011" =>
             if Parkade_Available_Capacity>0 then
                 Parkade_Available_Capacity <= Parkade_Available_Capacity - 1;
             else
                 Parkade_Available_Capacity <= Parkade_Available_Capacity + 0;
             end if;
           when "0011" =>
             if Parkade_Available_Capacity>1 then
                 Parkade_Available_Capacity <= Parkade_Available_Capacity - 2;
             elsif Parkade_Available_Capacity = 1 then
                Parkade_Available_Capacity <= Parkade_Available_Capacity - 1;
             else
                 Parkade_Available_Capacity <= Parkade_Available_Capacity + 0;
             end if;
           when "1100" =>
             if Parkade_Available_Capacity<98 then
                 Parkade_Available_Capacity <= Parkade_Available_Capacity + 2;
             elsif Parkade_Available_Capacity = 98 then
                Parkade_Available_Capacity <= Parkade_Available_Capacity + 1;
             else
                Parkade_Available_Capacity <= Parkade_Available_Capacity + 0;
             end if;
           when "0100"|"1000"|"1101"|"1110" =>
             if Parkade_Available_Capacity <99 then
               Parkade_Available_Capacity <= Parkade_Available_Capacity + 1;
             else
               Parkade_Available_Capacity <= Parkade_Available_Capacity + 0;
             end if;
           when "0000"|"0101"|"0110"|"1001"|"1010"|"1111" =>
             Parkade_Available_Capacity <= Parkade_Available_Capacity + 0;
           when others =>

        end case;


         -- Estimated design lines for above two tasks ~= 11x2 ~= 22
```

Figure 16.3 Design

```
            -- End required design lines above.

            if(Parkade_Available_Capacity=0) then
                led6_r<='1';
                led6_g<='0';
                led6_b<='0';

            elsif(Parkade_Available_Capacity>4) then
                led6_r<='0';
                led6_g<='1';
                led6_b<='0';
            elsif(Parkade_Available_Capacity<4 or Parkade_Available_Capacity=4 ) then
                led6_r<='0';
                led6_g<='0';
                led6_b<='1';
            else
                led6_r<='0';
                led6_g<='0';
                led6_b<='0';

            -- Write design lines here to update parkade three leds for remaining scenarios.
            -- Estimate design lines ~= 8
            end if;

         end if;

    end process;

end Behavioral;
```

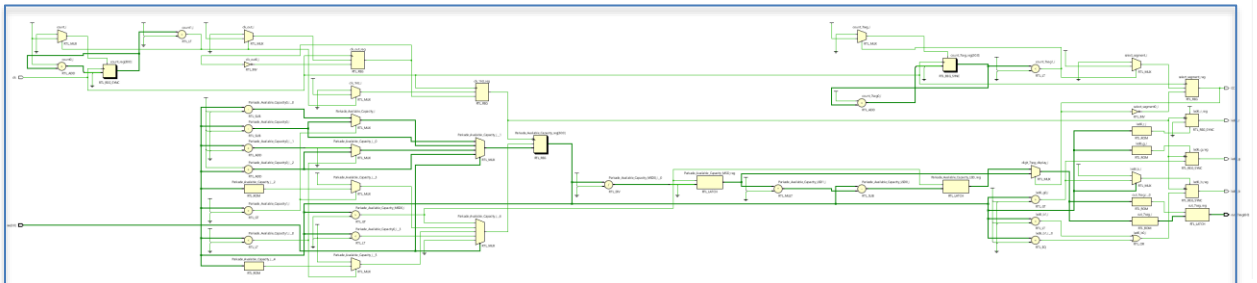
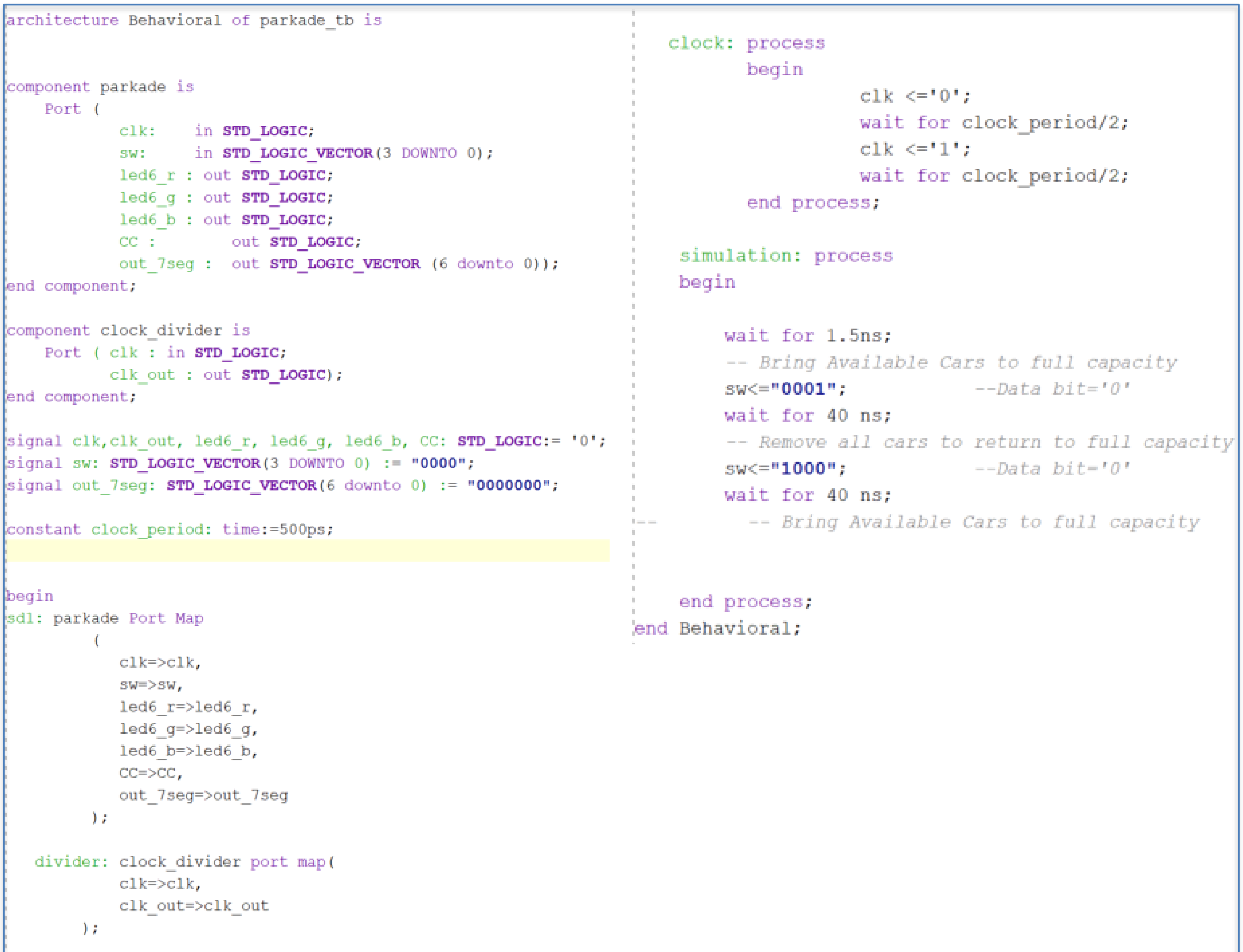


Figure 17 Schematic



```
architecture Behavioral of parkade_tb is


component parkade is
    Port (
            clk:    in STD_LOGIC;
            sw:     in STD_LOGIC_VECTOR(3 DOWNTO 0);
            led6_r : out STD_LOGIC;
            led6_g : out STD_LOGIC;
            led6_b : out STD_LOGIC;
            CC :        out STD_LOGIC;
            out_7seg :  out STD_LOGIC_VECTOR (6 downto 0));
end component;

component clock_divider is
    Port ( clk : in STD_LOGIC;
           clk_out : out STD_LOGIC);
end component;

signal clk,clk_out, led6_r, led6_g, led6_b, CC: STD_LOGIC:= '0';
signal sw: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
signal out_7seg: STD_LOGIC_VECTOR(6 downto 0) := "0000000";

constant clock_period: time:=500ps;


begin
sdl: parkade Port Map
         (
            clk=>clk,
            sw=>sw,
            led6_r=>led6_r,
            led6_g=>led6_g,
            led6_b=>led6_b,
            CC=>CC,
            out_7seg=>out_7seg
         );

   divider: clock_divider port map(
            clk=>clk,
            clk_out=>clk_out
        );
```

```
   clock: process
          begin
                   clk <='0';
                   wait for clock_period/2;
                   clk <='1';
                   wait for clock_period/2;
          end process;

    simulation: process
    begin

        wait for 1.5ns;
        -- Bring Available Cars to full capacity
        sw<="0001";             --Data bit='0'
        wait for 40 ns;
        -- Remove all cars to return to full capacity
        sw<="1000";             --Data bit='0'
        wait for 40 ns;
--        -- Bring Available Cars to full capacity


    end process;
end Behavioral;
```

Figure 18: Testbench

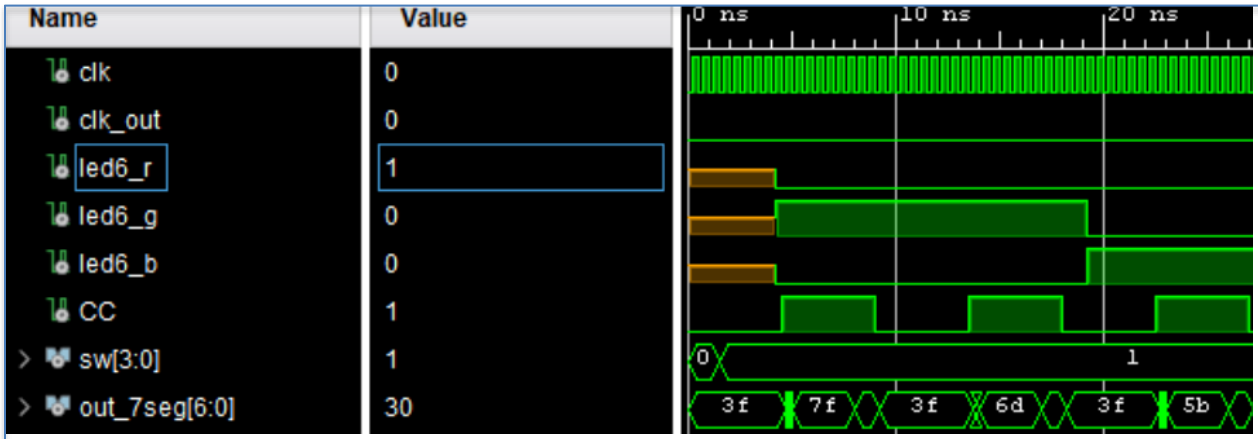


Figure 19: Parkade simulation