Digital System Design Lab Report, Lab4

Author: Ying Yiwen
Number: 12210159

Abstract

Finite State Machine is an important part of FPGA design, which can be implemented by temporal logic circuit and combinational logic circuit. In this experiment, we made a sequence signal generator and a sequence signal detector. We also implement the testbench code to test the circuit, which shows the logic is correct.

Contents

1 Introduction

For this experiment, we made a sequence signal generator and a sequence signal detector. We implemented the FSM (Finite State Machine) in various ways (case statements and with-select statements), which is an important part of real circuit design.

We also dealt with both temporal logic circuits and combinational logic circuits. In the temporal logic circuit section, we usually write code using process, in which we tends to use clock and reset as sensitive lists, and write trigger logic within process, passing next_state to current_state. In the combinational logic circuit section, we usually define the signaling logic within the architecture to give the next_state that is needed for the next timing logic circuit trigger. In the testbench code, we can set reset multiple times to cycle through the timing logic circuit triggers and observe the output. We also use wait to wait for a certain amount of time and observe the output.

In vivado, we can get behavior simulation result, post-synthesis simulation result and post-implementation simulation result. They validate the behavioral simulation result of the circuit, the simulation result of adding devices and the simulation result of considering the wiring which can be used to validate our code. We also have access to schematic, which shows the gate level results and Register Transfer Level results corresponding to the code.

2 Sequential Signal Generator

A circuit that generates a specific set of serial sequences on demand is called a sequence signal generator. We can use state transfer and clock input to generate a sequence signal.

2.1 VHDL Code

Here shows the logic of the sequence signal generator. With renew part implemented by temporal logic circuit, and the state transfer part implemented by case sentense.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sequence_generator is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        dataout : out STD_LOGIC
    );
end sequence_generator;

architecture Behavioral of sequence_generator is
    signal current_state : std_logic_vector  (2 downto 0) := (others => '0');
    signal next_state :  std_logic_vector  (2 downto 0) := (others => '0');
begin
-- accident
process (clk, reset) is
begin
    if reset = '1' then
        current_state <= "000";
    elsif clk'event and clk = '1' then
        current_state <= next_state;
    end if;
end process;

-- state transfer
```

```
process (current_state) is
begin
    case current_state is
        when "000" =>
            dataout <= '1'; -- output
            next_state <= "001"; -- next state
        when "001" =>
            dataout <= '1';
            next_state <= "010";
        when "010" =>
            dataout <= '0';
            next_state <= "011";
        when "011" =>
            dataout <= '1';
            next_state <= "100";
        when "100" =>
            dataout <= '1';
            next_state <= "101";
        when "101" =>
            dataout <= '1';
            next_state <= "110";
        when "110" =>
            dataout <= '0';
            next_state <= "000";
        when others =>
            dataout <= '0';
            next_state <= "000";
    end case;
end process;
end Behavioral;
```

2.2 Testbench Code

Giving each signal a value, we can test the circuit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testbench is
end testbench;

architecture tb of testbench is
-- port declaration
component sequence_generator is
  Port (
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    dataout : out STD_LOGIC
  );
end component;

-- signal declaration
signal clk : STD_LOGIC := '0';
signal reset : STD_LOGIC := '0';
```

```
signal dataout : STD_LOGIC;
constant period : time := 10 ns;

begin
  -- port map
  uut: sequence_generator port map (
    clk => clk,
    reset => reset,
    dataout => dataout
  );

  clk <= not clk after period / 2;

  tb: process is
  begin
    reset <= '1';
    wait for period;
    reset <= '0';
    wait for period;
    wait;
  end process;

end tb;
```

2.3 Therotical Analysis

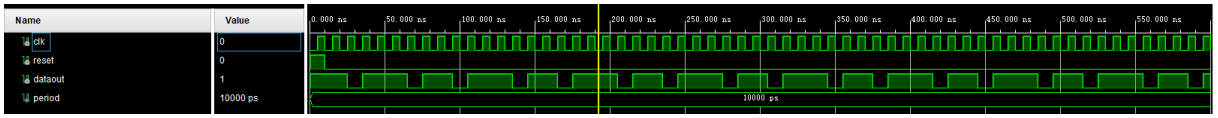
A sequence signal generator is a digital circuit based on a finite state machine (FSM) for generating a specific binary sequence (e.g., "1101110"). The core principle is the generation of sequences through a combination of state transfers and output signals. A finite state machine consists of a state register, state transfer logic and output logic. The state register stores the current state, the state transfer logic calculates the next state based on the current state and input signals, and the output logic generates output signals based on the current state.

The state transfer diagram defines the relationship between states, with each state corresponding to an output value. For example, in the diagram, the state starts from 000 and is driven by the clock signal to 001, 010, 011, etc. Each state corresponds to an output value (e.g., 1, 0, etc.), and these output values are combined to form the desired sequence "1101110". The state transfer is cyclic, when the state reaches the last state, it will revert to the initial state and continue to generate the sequence.

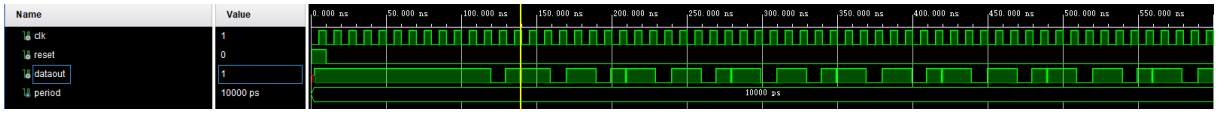
The reset signal is one of the important inputs to the circuit and is used to initialize the state to the initial state (e.g., 000), ensuring that the sequence is generated from the beginning. The reset signal is usually activated when the circuit is started or when the sequence needs to be restarted. The clock signal (clk) is another key input used to drive state transfers. Each clock cycle, the state register is updated to the next state and the corresponding signal is output. This synchronization mechanism ensures that sequence generation is orderly and stable.

The advantage of the sequence generator is its simple and efficient design, which realizes the complex sequence generation by means of a finite state machine. This design not only saves hardware resources, but also makes it easy to modify the generated sequences by adjusting the state transfer diagram.

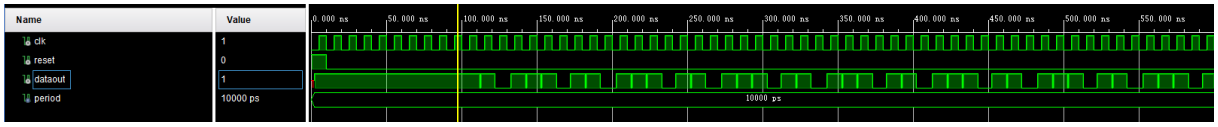
2.4 Simulation Result



(a) Behavioral Simulation Result



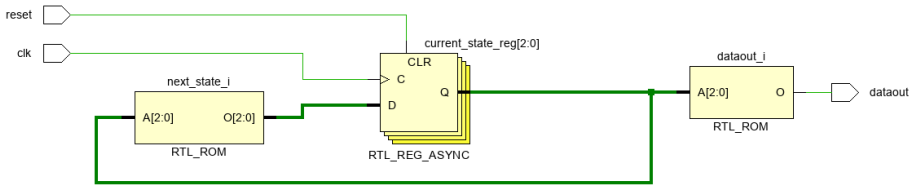
(b) Post-Synthesis Simulation Result



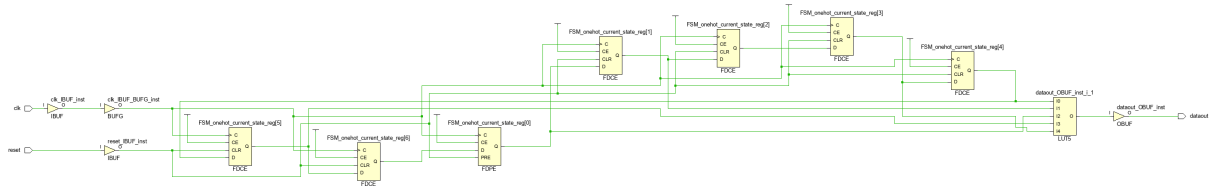
(c) Post-Implementation Simulation Result

Figure 1: Simulation Result

The behavior simulation matches the code more, while the post-synthesis timing simulation considers which device to use, thus imports more delay. And the post-implementation simulation considers the wiring, thus is more similar with the true result on the board. We can also see race-hazard in the post-synthesis simulation result, which is caused by the delay of the device. As we use combinational logic circuit to transfer the state, the delay of the device will cause the race-hazard.



(a) Behavior Schematic



(b) Post-Synthesis Schematic

Figure 2: Schematic Result

The schematic shows the gate level results and Register Transfer Level results corresponding to the code.

3 Sequential Signal Detector

A circuit that finds a specific sequence in a serial sequence and outputs the result is called a sequence signal detector.

3.1 VHDL Code

Here shows the logic of the sequence signal detector. With renew part implemented by temporal logic circuit, and the state transfer part implemented by with-select statements.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sequence_generator is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        input : in STD_LOGIC;
        dataout : out STD_LOGIC
    );
end sequence_generator;

architecture Behavioral of sequence_generator is
    signal current_state : std_logic_vector(1 downto 0) := "00";
    signal next_state : std_logic_vector(1 downto 0);
begin
    -- state register
    process(clk, reset)
    begin
        if reset = '1' then
            current_state <= "00";
        elsif CLK'event and CLK = '1' then
            current_state <= next_state;
        end if;
    end process;

    -- next state logic
    with current_state & input select
        next_state <=
            "00" when "000",
            "01" when "001",
            "10" when "010",
            "01" when "011",
            "00" when "100",
            "11" when "101",
            "11" when "110",
            "11" when "111",
            "00" when others;
    with current_state & input select
        dataout <=
            '0' when "000" | "001" | "010" | "011" | "100",
            '1' when "101" | "110" | "111",
            '0' when others;

end Behavioral;
```

3.2 Testbench Code

Giving each signal a value, we can test the circuit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testbench is
end testbench;

architecture tb of testbench is
-- port declaration
component sequence_generator is
  Port (
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    input : in STD_LOGIC;
    dataout : out STD_LOGIC
  );
end component;

-- signal declaration
signal clk : STD_LOGIC := '0';
signal reset : STD_LOGIC := '0';
signal input : STD_LOGIC;
signal dataout : STD_LOGIC;
constant period : time := 10 ns;

begin
  -- port map
  uut: sequence_generator port map (
    clk => clk,
    reset => reset,
    input => input,
    dataout => dataout
  );

  -- clock
  clk <= not clk after period / 2;

  tb: process is
  begin
    for i in 1 to 10 loop
      reset <= '1';
      wait for period;
      reset <= '0';
      wait for period;

      -- test input sequence
      for j in 1 to 2 loop
        input <= '0';
        wait for period;
        input <= '1';
        wait for period;
        input <= '1';
```

```
        wait for period;
        input <= '1';
        wait for period;
        input <= '0';
        wait for period;
        input <= '1';
        wait for period;
        input <= '1';
        wait for period;
      end loop;
    end loop;
    wait;
  end process;

end tb;
```

3.3 Therotical Analysis

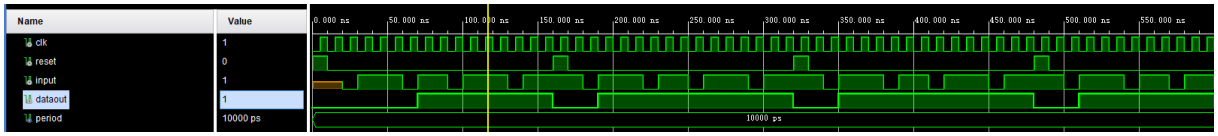
The 101 Sequence Detector is a digital circuit based on a finite state machine (FSM) for detecting "101" patterns in an input sequence. It works by combining state transfer and output signals to achieve sequence detection. The detector has three inputs: a reset signal, a serial input and a clock signal. The reset signal is used to initialize the state to the initial state (e.g., 00) to ensure that the detection starts from the beginning; the serial input is used to input the binary sequence to be detected; the clock signal drives the state transfer to ensure that the detection process is carried out synchronously. The output signal, which is set to 1 when a "101" sequence is detected, is held at 1 until the reset signal is activated.

The detector realizes the state transfer by means of a finite state machine. The initial state is 00, which is forced back to this state when the reset signal is activated. Each state determines the next state based on the current input and outputs the corresponding signal. For example, when the input is 1, the state is transferred from 00 to 01; when the input is 0, the state is transferred from 01 to 10; when the input is 1, the state is transferred from 10 to 11, and at this time, the output is changed to 1, which indicates that the "101" sequence is detected. The output signal becomes 1 when the state is transferred to 11, and remains 1 until the reset signal resets it to 0.

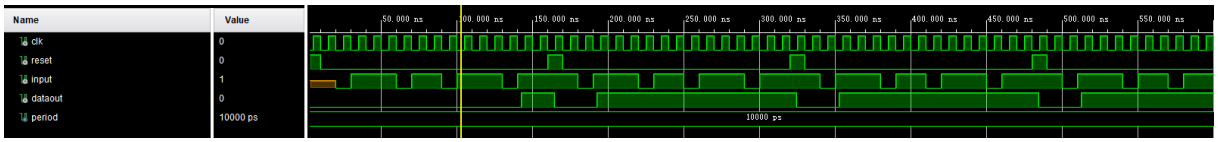
The detector operates as follows: when the reset signal is activated, the state is initialized to 00. The clock signal drives the state transfer, and the state is updated every clock cycle according to the input. When the state shifts to 11, the output signal is 1, indicating that the "101" sequence is detected. The output remains at 1 until a reset signal reinitializes the state.

The 101 Sequence Detector is used to recognize a specific signal pattern or data sequence. Advantages include efficiency, flexibility and synchronization. Efficient sequence detection is achieved through a finite state machine, the sequence detected can be easily modified by adjusting the state transfer diagram, and the clock signal ensures the stability and reliability of the detection process. This design is simple and efficient and is widely used in various digital systems.

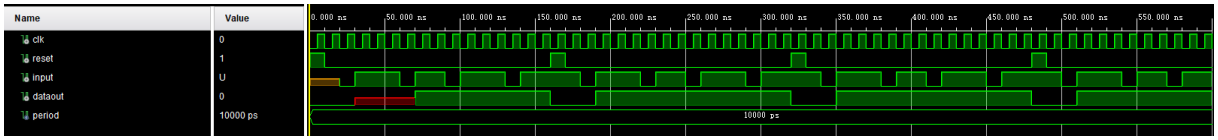
3.4 Simulation Result



(a) Behavioral Simulation Result



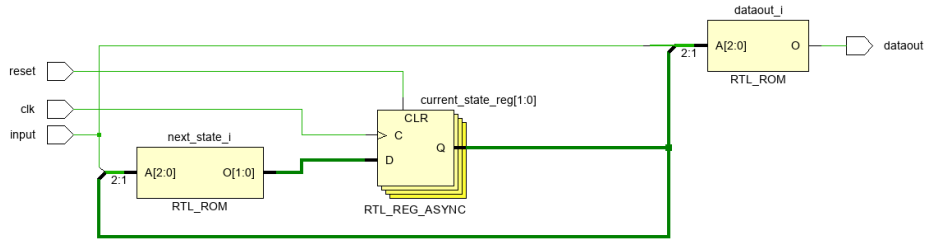
(b) Post-Synthesis Simulation Result



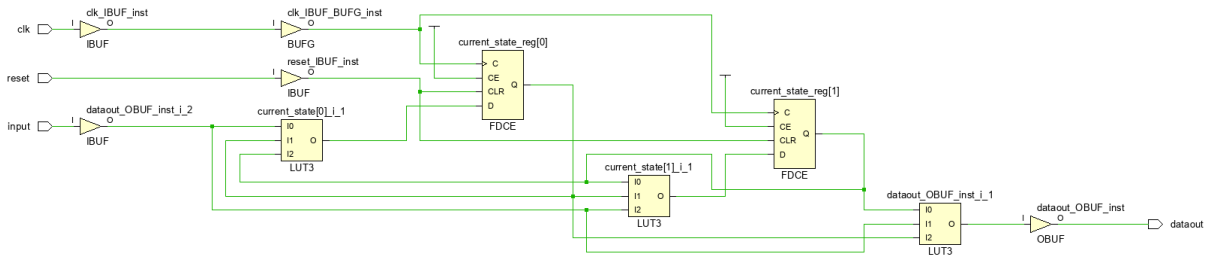
(c) Post-Implementation Simulation Result

Figure 3: Simulation Result

The behavior simulation also shows the logic of the code. In post-synthesis simulation and post-implementation simulation, there exists delays, which are caused by the device and wiring implementation. We can also see undefined signal in post-implementation simulation result, which is because the signal from the last state hasn't be transported to the next state yet.



(a) Behavior Schematic



(b) Post-Synthesis Schematic

Figure 4: Schematic Result

The schematic shows the gate level results and Register Transfer Level results corresponding to the code.

4 Conclusion

In this experiment, we designed and implemented a finite state machine (FSM) based sequence signal generator and sequence signal detector. By using the VHDL programming language, we successfully implemented these two circuits on an FPGA and verified their logical correctness through testbed code.

The Sequence Signal Generator is capable of generating specific binary sequences such as “1101110” on demand. It generates sequences through a combination of state transfers and output signals. A finite state machine consists of state registers, state transfer logic and output logic. A state transfer map defines the relationship between states, and each state corresponds to an output value. Reset signals are used to initialize the states and ensure that the sequence is generated from scratch, while clock signals drive the state transfers to ensure orderly and stable sequence generation.

The sequence signal detector is used to detect specific patterns in the input sequence, such as “101”. It is also based on a finite state machine and implements sequence detection through a combination of state transfers and output signals. The detector has three inputs: a reset signal, a serial input and a clock signal. The reset signal is used to initialize the state, the serial input is used to input the binary sequence to be detected, and the clock signal drives the state transfer. The output signal is set to 1 when the target sequence is detected and remains at 1 until the reset signal is activated.

In Vivado software, we performed behavioral simulation, post-synthesis simulation and post-implementation simulation to verify the behavior of the circuit, the simulation results after the devices are added, and the simulation results after the wiring is considered. These simulation results helped us to confirm the correctness of the code and observe the phenomenon of competing adventures due to device delays and wiring.

Through this experiment, we gained an in-depth understanding of the application of finite state machines in FPGA design, mastered the implementation of timing logic circuits and combinational logic circuits, and learned to use testbed code to verify circuit logic. Experimental results show that our design is correct and can efficiently generate and detect specific binary sequences. This design not only saves hardware resources, but also provides a high degree of flexibility by allowing the generated or detected sequences to be easily modified by adjusting the state transfer diagram. In conclusion, this experiment successfully demonstrates the importance and usefulness of finite state machines in digital system design.