EE2016: Microprocessers Lab - Assignment 2 Report

Aadarsh Ramachandiran (EE23B001), Aditya Kartik (EE23B003), Rohita Datta (EE23B065)

August 21, 2024

Abstract

This is a detailed report on the Verilog assignment on simulating a basic D flip-flop, D flip-flop with reset, writing code for clock divider and decoder to enable implementation of Johnson counter on a seven segment display on an FPGA board using Xilinx Vivado. It contains the details of the solution including code, observation and comments on the programming and debugging experience.

1 Introduction

This assignment involves (i) simulating a D flip-flop using Xilinx Vivado (ii) extending the design by adding a reset signal (iii) designing a 3-bit Johnson counter via instantiations of D flip-flop with reset (iv) writing clock divider and decoder modules in Verilog to enable implementation of the Johnson counter design on a seven segment display on the FPGA board.

A D flip-flop is the most simplest circuit that acts like a basic memory cell. It is used to store single bit input data. A reset signal is often added to the design of such circuits in order to clear previous outputs, if any. A Johnson counter is a type of digital counter that cycles through a set of states using a chain of flip-flops. A decoder, in this context, is a module used to convert binary output to a form compatible with displaying on a seven segment display. A clock divider is a module used to reduce the frequency of an input clock signal. In this case, an FPGA board has a default clock of 50 MHz, however, in order to observe the counting on the seven segment display on the board, we reduce the frequency using the clock divider (to, say, 2Hz)

2 Design

2.1 D flip-flop

2.1.1 Description

Figure 1 shows the symbol and truth table for a D (data) flip-flop. The D flip-flop is used to store data at a predetermined time and hold it until it is needed. The input to this is denoted by D. The other input of the circuit is the clock signal.The output is changed only at positive edge trigger on the clock input (denoted by >. Negative edge trigger is denoted by ∘ >). If there is a change of the state

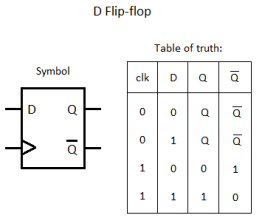

Figure 1: Symbol and Truth Table of D flip-flop (source)

of the input D, this change will not be reflected in the output Q until the next positive edge of the clock signal. Given below is the characteristic equation of a D flip-flop.

$$Q(t+1) = D$$

As it can be seen, the D flip-flop transfers its input as output at positive edge triggers. At other times, the output retains its old value (the previous input) despite new input changes. This brings out its storage function. The next section shows the code implemented for the D flip-flop in Verilog. The inputs are defined as d and clk and the outputs Q and Q' as q and qbar respectively. We write a "behavioural" model for this which contains procedural statements, which control the simulation and manipulate variables of the data types. Behavioral modeling provides a high-level abstraction that facilitates efficient design and simulation. It is a matter of style and does not affect the results of the experiment.

2.1.2 Code

D Flip-flop module:

```
// This code describes a behavioural model for D flip flop
module dflipflop(q, qbar, d, clk);
output q, qbar;
input d, clk;
reg q, qbar;
always@(posedge clk)
    begin
        q = d;
        qbar = ~d;
    end
endmodule
```

We also write a testbench to simulate the circuit in software. This allows us to test its veracity before implementing it in hardware. In this file, we define the inputs and outputs, instantiate the D flip-flop module and vary the input d sequentially while clk oscillates between 0 and 1 at a frequency of 50 MHz (in the initial block) and wemonitor corresponding outputs.

D Flip-flop testbench:

```
// This code describes a testbench for testing the d flip flop module
`timescale 1ns/1ps
module tb_dflipflop();
reg d, clk;
wire q, qbar;
dflipflop dff(q, qbar, d, clk);
initial
begin
    clk = 1'b0;
    forever # 10 clk = ~clk;
end
initial
begin
    d = 1'b0;
    #10
    d = 1'b1;
    #20
    d = 1'b0;
    #20
    d = 1'b1;
    #30 $finish;
end
```

Input			Output	
D	reset	clock	Q	Q'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Figure 2: Characteristic table of D flip-flop with Reset(source)

```
initial $monitor($time, "clk=%b, d=%b, q=%b, qbar=%b", clk, d, q, qbar);
endmodule
// The initial value of q and qbar will be denoted by 'x' because it is undefined
// until the first positive edge of clock
```

2.2 D flip-flop with Reset

2.2.1 Description

Figure 2 shows the characteristic table for a D flip-flop with reset. The outputs Q and Q' depend on the values of D, reset and clock. At a positive edge trigger, i.e., when the clock transitions from logic low (0) to logic high (1), if the reset is logic low (0) then q and qbar are automatically set to 0 and 1 respectively.

2.2.2 Code

D Flip-flop with Reset Module:

```
// This code demonstrates a behavioural model for the D-flip flop circuit
// with the option of reset
module dflipflop_withreset(q, qbar, d, rst, clk);
output q, qbar;
input d, rst, clk;
reg q, qbar;
always@(posedge clk or negedge rst)
    begin
        if (~rst)
            begin
            q <= 1'b0;
            qbar <= 1'b1;
            end
        else
            begin
            q <= d;
            qbar <= ~d;
            end
    end
endmodule
```

We also write a testbench to simulate this in software. We vary the inputs d, rst and clk simultaneously and monitor how the outputs q and qbar change.

D Flip-flop with Reset Testbench:

```
// This code describes a testbench for testing the d flip flop with reset module
`timescale 1ns/1ps
module tb_dflipflop_withreset();
reg d, rst, clk;
```

```
wire q, qbar;
dflipflop_withreset dff(q, qbar, d, rst, clk);
initial
begin
    clk = 1'b0;
    forever #10 clk = ~clk;
end
initial
begin
    $dumpfile("test_dflipflop.vcd");
    $dumpvars(0, tb_dflipflop_withreset);
    rst = 1'b0;
    #15
    rst = 1'b1;
    d = 1'b0;
    #10
    d = 1'b1;
    #20
    rst = 1'b0;
    #10
    rst = 1'b1;
    #20
    d = 1'b0;
    #20
    d = 1'b1;
    #30 $finish;
end
initial $monitor($time, "clk=%b, rst=%b, d=%b, q=%b, qbar=%b", clk, rst, d,
endmodule
// The initial value of q and qbar will be denoted by 'x' because it is undefined
// until the first positive edge of clock
```

2.3 Clock divider

2.3.1 Description

A clock divider is a circuit that reduces the frequency of an input clock signal. In the context of this experiment, we use a clock divider since the internal clock of the FPGA has a frequency of 50 MHz and we want an output frequency of a few Hz so that we may view the changes on the Seven segment display.

Physically, this is implemented by a series of D flip-flops connected in a binary counter arrangement. The input clock triggers the first flip flop, which in turn triggers the subsequent flip flop when it toggles.

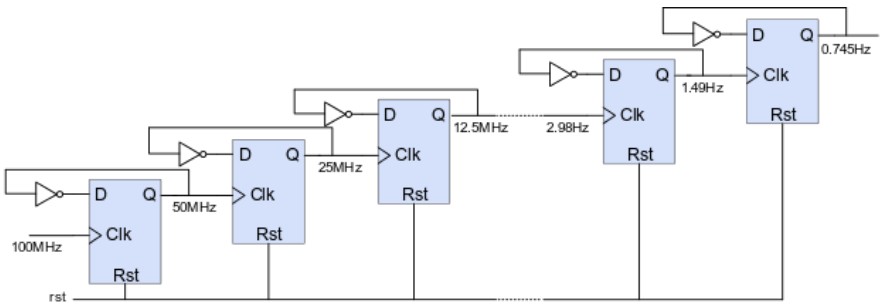

Figure 3: Clock divider with a cascade of D Flip Flops (source)

In the code implementation however, instead of using the cascade clock divider implementation

with instantiations of D flip flops, we use an internal variable clockCount, which increments with every cycle of the input clock, the output clock toggles after clockCount reaches a certain value, in this case 50,00,000.

In effect, the output clock will now have a frequency smaller than the input clock by a factor of $2 \times 50,000,000$ since the output clock will toggle twice, thereby, finishing one cycle, after 100,000,000 clock cycles of input clock. This ends up with an effective clock of frequency 0.5Hz. (The factor by which the clock is reduced is flexible as long as the result is a few Hz; we simply selected one for an effective 0.5 Hz clock).

2.3.2 Code

Clock divider module

```
// This code demonstrates a clock divider module to reduce clock frequency
// to observe results in an FPGA board
module clk_divider(outClk, inClk, reset);
input inClk;
input reset;
output reg outClk;
reg clockCount;//use this to test clk_divider module
//reg [25:0] clockCount;(use this for fpga)

always@(negedge reset or posedge inClk)
begin
    if (reset == 1'b0)
        begin
            outClk = 1'b0;
            clockCount = 1'd0;
            //clockCount = 26'd0;(for fpga)
        end
    else
        begin
        if (clockCount >= 1'd1)
        //if (clockCount >= 26'd50000000)
            begin
                clockCount = 1'd0;//for testing clk_divider module
                //clockCount = 26'd0;(for fpga clock division)
                outClk = ~ outClk;
            end
        clockCount <= clockCount + 1;

        end
end
endmodule
```

We note that the clock divider has an active low reset.(asynchronous)

We also write a testbench to simulate this. We vary the inputs and monitor how the outputs change.

Clock Divider test bench

```
// Code to test the clock divider module
`timescale 1ns/1ps
module tb_clk_divider();
reg inClk, reset;
wire outClk;
clk_divider cd(outClk, inClk, reset);
initial
    begin
    inClk = 1'b0;
```

```
    forever #20 inClk = ~inClk; //Simulating FPGA's 50MHz clock
    end
initial
begin
    reset = 1'b1;
    #5
    reset = 1'b0; //active low reset
    #5
    reset = 1'b1;
    #1000 $finish;
end
initial $monitor($time, " inClk =%b, reset =%b, outClk =%b", inClk, reset, outClk);
endmodule
```

2.4 Decoder

2.4.1 Description

A seven segment display has seven LEDs, labelled a to g, that can be turned on or off to form digits. The signals we must provide for this are obtained by mapping BCD digits to corresponding segments on the display.

For example, the digit 1 requires that segments b and c are turned on. We can write decoder logic for this purpose mapping 1'd1 as 7'b0000110.

The first bit corresponds to segment g and the last bit corresponds to segment a.

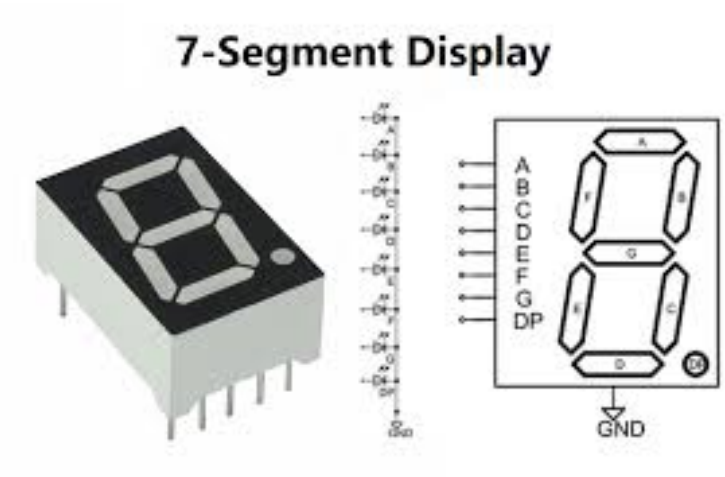

Figure 4: A Seven Segment display (source)

A point to note is that due to the implementation of the Seven Segment display,it can be considered "active low" per say, so we must provide 0 if we want a segment to be turned on. The code thus finds the complement of the mapping and provides it as output. There is also an eighth bit marked 1(turned off) corresponding to the decimal point.

The next section shows the Verilog code for this decoder logic.

2.4.2 Code

Seven-Segment Display Decoder Module

```
// This code demonstrates the decoder module to display 3-bit Johnson counter results
// on the Seven-Segment display
module decoder(Seven_Seg, cntr);
input [2:0] cntr;
```

```
output [7:0] Seven_Seg;
reg [6:0] val;
assign Seven_Seg = {1'b1, ~val};
always@(cntr) // Whenever counter value changes -> get value for display
begin
    case(cntr)
        3'd0: val = 7'b0111111;
                3'd1: val = 7'b0000110;
                3'd2: val = 7'b1011011;
                3'd3: val = 7'b1001111;
                3'd4: val = 7'b1100110;
                3'd5: val = 7'b1101101;
                3'd6: val = 7'b1111101;
                3'd7: val = 7'b0000111;
        //      3'd8: val = 7'b1111111;
        //      3'd9: val = 7'b1101111;
 //counter never reaches 8 or 9.

    endcase
end
 // Concatenate bits for decimal and counter value
 // (value negated because Seven Segment display is active low)

endmodule
```

2.5 3-bit Johnson Counter

2.5.1 Description

Figure 4 shows the circuit diagram for a general n-bit Johnson counter using a series of D flip-flops. The circuit is to function as a counter and hence there are no variable inputs. The outputs are the outputs of each flip-flop. These together represent the count. There is also a clock input which is used to enable counting in a synchronized manner (with same frequency as clock). Additionally, there also optional inputs such as reset (or clear) and preset. The reset option comes handy when you desire to clear any previous outputs that prevail initially. In our case, it is useful in order to start counting from zero when required. The preset achieves the opposite of reset function. (This is not necessary in our case and is ignored this point onward) Notice that the negation of the last flip-flop output (Qbar) is returned back as input (D) to the first flip-flop. For this reason, it is sometimes called as a "twisted ring-counter". If the output were connected back instead of its negation, you get a ring counter. For a 3-bit Johnson counter the counting states transition as follows:

$$000 \rightarrow 100 \rightarrow 110 \rightarrow 111 \rightarrow 011 \rightarrow 001 \rightarrow \text{repeat}$$

with the bits representing the outputs of the three flip-flops.

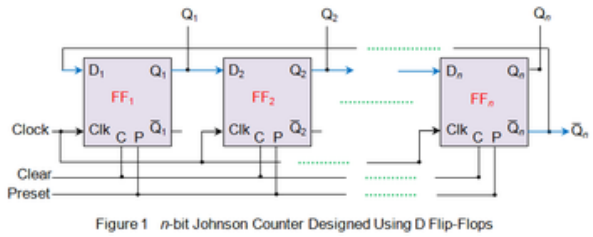

Figure 5: Circuit of a general n-bit Johnson counter (source)

The next section shows the code implemented for the 3-bit Johnson counter circuit in Verilog. We use the clock divider, D flip-flop and decoder module introduced previously. The decoder is to modify the bit outputs to a form compatible with a seven segment display and the clock divider is to reduce the frequency of counting. An FPGA board's input clock usually has a frequency of 50 MHz which is very high to allow easy reading of the counts. Thus, we reduce clock frequency to 1 or 2 Hz. Note the introduction of a new variable called "digit" in the code. This represents an enable pin for the seven segment display.

2.5.2 Code

3-bit Johnson counter module

```
// This code demonstrates a trhee-bit Johnson Counter circuit model
module three_bit_johnson_counter(Seven_Seg, digit, in_clk, rst);
input in_clk, rst;
wire out_clk;
output [7:0] Seven_Seg;
output [3:0] digit;
wire [2:0] cntr;

wire q0, q1, q2;
wire q2bar, q1bar, q0bar;

assign digit = 4'b0001; //this enables seven seg display.
assign cntr = {q0, q1, q2};
clk_divider cd0(out_clk, in_clk, rst);
dflipflop_withreset d2(q2, q2bar, q1, rst, out_clk);
dflipflop_withreset d1(q1, q1bar, q0, rst, out_clk);
dflipflop_withreset d0(q0, q0bar, q2bar, rst, out_clk);
decoder dc0(Seven_Seg, cntr);

endmodule
```

We also implement a testbench where we input a 50 MHz clock, enable reset initially and simulate the output of a seven segment display.

3-bit Johnson counter testbench

```
// This code serves as a testbench to simulate the Johnson counter
`timescale 1ns/1ps
module tb_three_bit_johnson_counter();
reg in_clk, rst;
wire out_clk;
wire [2:0] cntr;
wire [7:0] Seven_Seg;
wire [3:0] digit;
three_bit_johnson_counter jc(Seven_Seg, digit, in_clk, rst);
initial
begin
    in_clk = 1'b0;
    forever #10 in_clk = ~in_clk;
end
initial
begin
    $dumpfile("test_johnson_counter.vcd");
    $dumpvars(0, tb_three_bit_johnson_counter);
    rst = 1'b0;
    #5
```

```
    rst = 1'b1;
    #200 $finish;
end
initial $monitor($time, "in_clk=%b, rst=%b, digit=%b, Seven_Seg=
    ↪ %b", in_clk, rst, digit, Seven_Seg);
endmodule
```

3 Implementation

Once the Verilog code for all the circuits are done, we proceed to simulate the Johnson Counter in Xilinx and realize the results on an FPGA board. We create a new project in Vivado and add the required Verilog files (module and testbench) through the "Add sources" option.
Note: It must be ensured that the module is declared as "top-level" in the hierarchy of files. Vivado displays errors, otherwise. Also, the Verilog files of all relevant modules in the experiment must be included. (For example, the decoder, clock divider and D flip-flop modules should be included alongside Johnson counter module when attempting to simulate the latter.)

3.1 Simulating in Xilinx

After adding the verilog files, we simulate them. Vivado shows the outputs and pops open the waveform veiwer to display the waveforms of various signals. The waveforms for the counter can be found in 6. We also simulate the clock divider and the D flip-flop to test its correctness.

3.2 FPGA Realization

Once the outputs are analyzed and the circuit is verified to be correct, we proceed to implement it on the hardware. We write a Xilinx Design Constraints (.xdc) file to configure the required ports to be used in the FPGA board. (Clock, reset, seven segment display enable and the segments of the seven segment display)
Code for the.xdc file to implement the Johnson counter

```
# Design constraint file for the three-bit Johnson Counter
# Clock signal
set_property -dict { PACKAGE_PIN N11 IOSTANDARD LVCMOS33 } [get_ports { in_clk }];

# Sliding switch -- to start count
set_property -dict { PACKAGE_PIN L5 IOSTANDARD LVCMOS33 } [get_ports { rst }];

#Enable seven segment display
set_property -dict { PACKAGE_PIN F2 IOSTANDARD LVCMOS33 } [get_ports {digit[0]}];
set_property -dict { PACKAGE_PIN E1 IOSTANDARD LVCMOS33 } [get_ports {digit[1]}];
set_property -dict { PACKAGE_PIN G5 IOSTANDARD LVCMOS33 } [get_ports {digit[2]}];
set_property -dict { PACKAGE_PIN G4 IOSTANDARD LVCMOS33 } [get_ports {digit[3]}];

#locations of the segments on the display
set_property -dict { PACKAGE_PIN G2 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[0]}];
set_property -dict { PACKAGE_PIN G1 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[1]}];
set_property -dict { PACKAGE_PIN H5 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[2]}];
set_property -dict { PACKAGE_PIN H4 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[3]}];
set_property -dict { PACKAGE_PIN J5 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[4]}];
set_property -dict { PACKAGE_PIN J4 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[5]}];
set_property -dict { PACKAGE_PIN H2 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[6]}];
set_property -dict { PACKAGE_PIN H1 IOSTANDARD LVCMOS33 } [get_ports {Seven_Seg[7]}];
```

Then we connect the FPGA board to the computer and generate the bitstream by clicking "Generate Bitstream" in the Xilinx menu. Then, we program the device by clicking "Program device". The circuit will then be successfully implemented in the FPGA board.

4 Observation

We use the sliding switch in the FPGA board for the reset input and the output count will be displayed in the seven segment display. When reset is 0, the count resets and when it is 1, the Johnson counting is displayed.

5 Conclusion

The experiment successfully demonstrated the design and simulation of a D flip-flop, extended it with reset option, designed a decoder and clock divider and combined them for the simulation and implementation of the 3-bit Johnson counter.

6 Appendices

The verilog codes used in this assignment are available here
The waveforms for various circuits can be seen below:

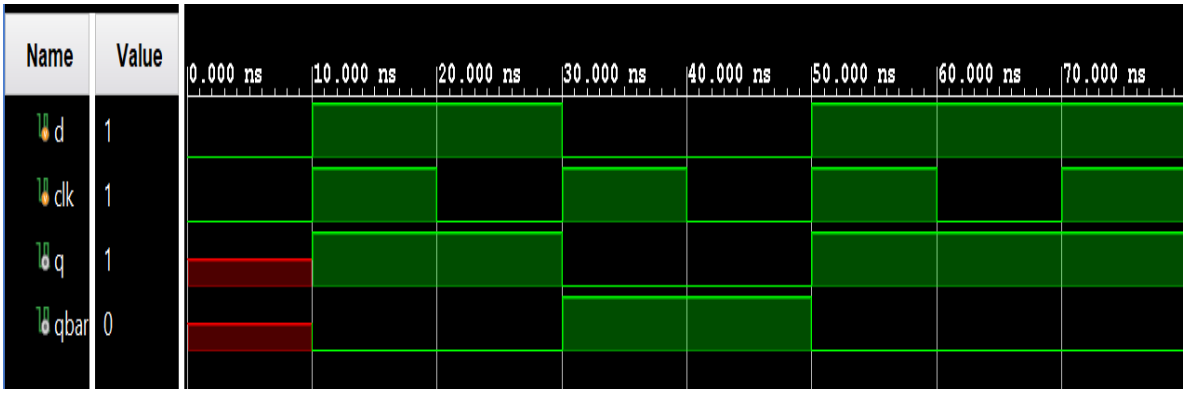

Figure 6: D flip flop waveform

We notice that q is in an indeterminate state until the first positive edge of clk is encountered. This illustrates the importance of initialising clock and connecting it to the flip flop properly.

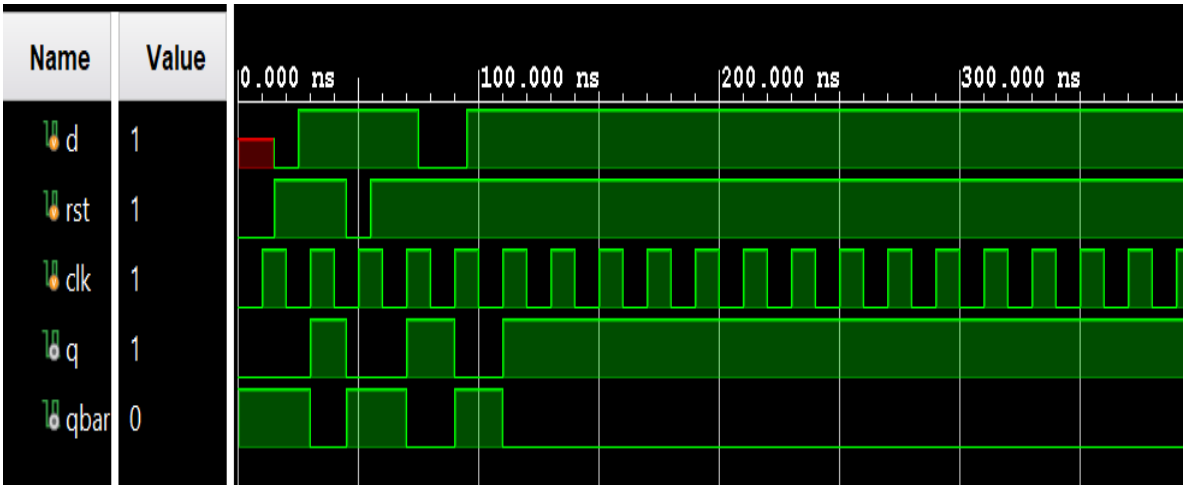

Figure 7: D flip flop with reset waveform

The signal q is initialised to 0 since reset is initialsed to 0(active low). We also notice that the flip flop reset is asynchronous,as it immediately gets reset without needing a positive edge.

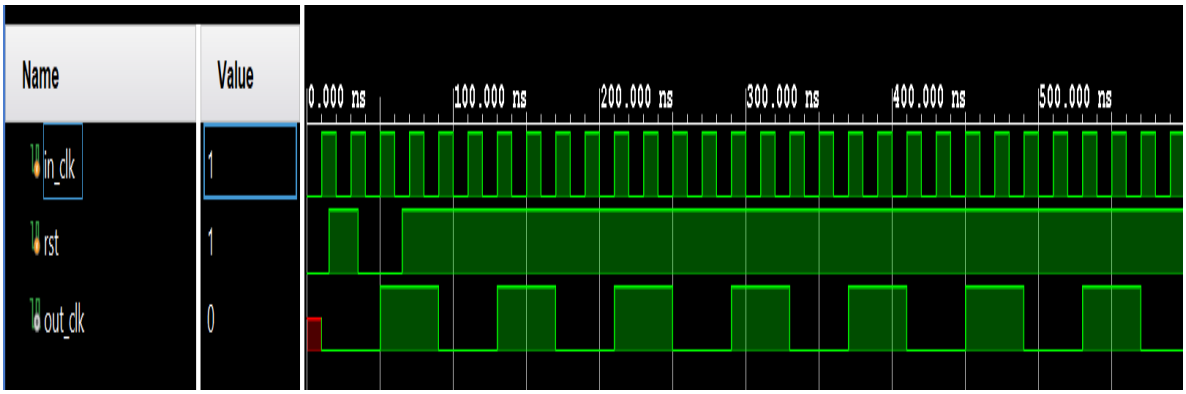

Figure 8: Clock division by 4

The signal for out_clk is reset synchronously.

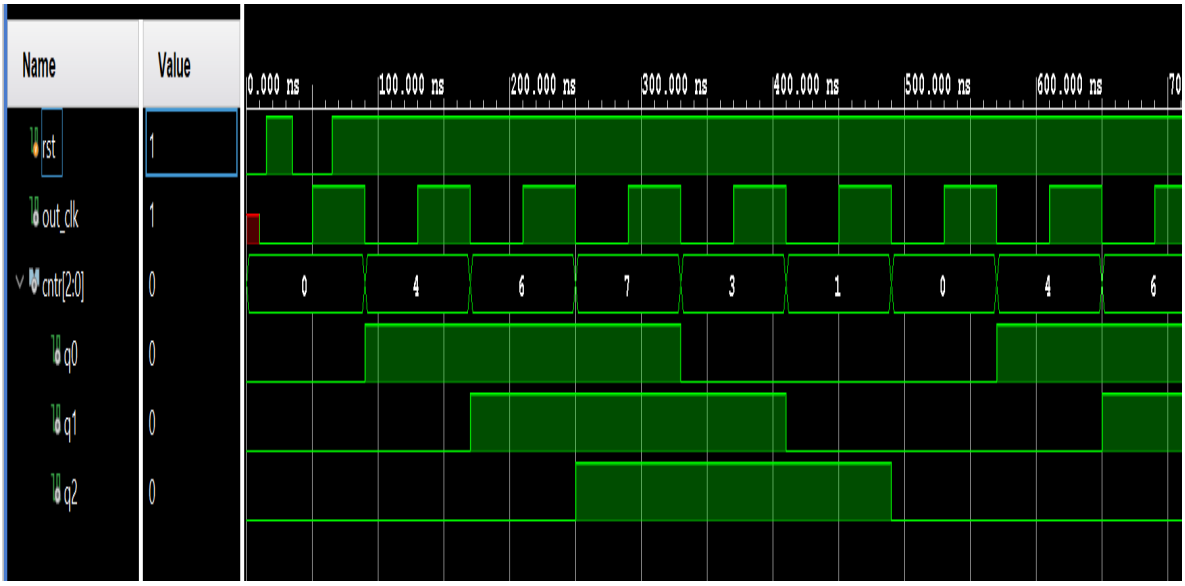

Figure 9: Counter waveform

Counter shows a sequence 0-4-6-7-3-1-

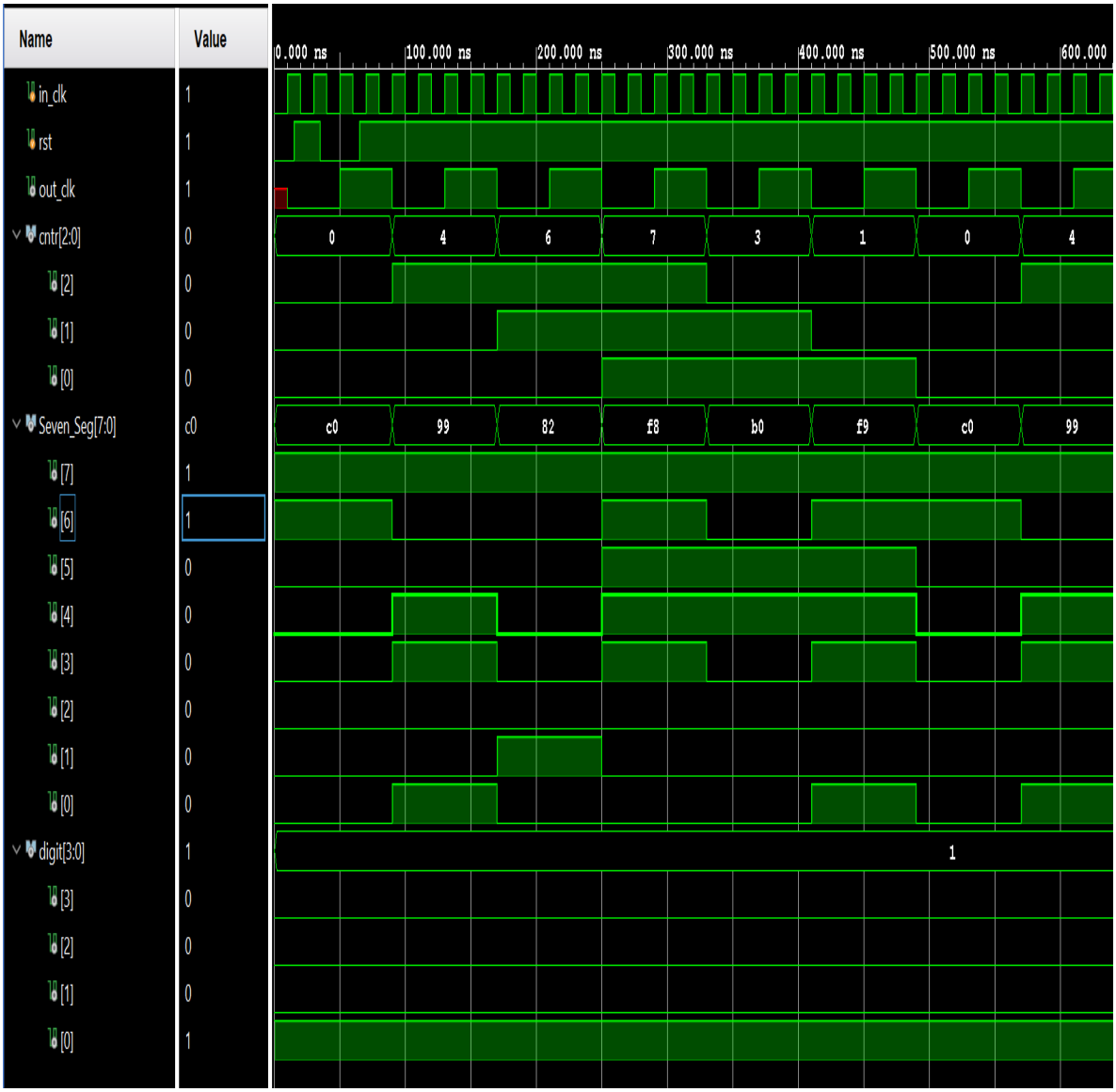

Figure 10: Johnson Counter waveform with decoder outputs

The signal, digit[3:0] enables a single seven segment display unit. Seven_Seg is the output of the decoder, serves as input to the segments numbered g to a.