

FPGA LAB
MINI PROJECT
IMPLEMENTATION OF MORSE CODE
DETECTION ON FPGA

Submitted in the partial fulfilment of the requirements of the
degree of
Bachelor of Technology

By:

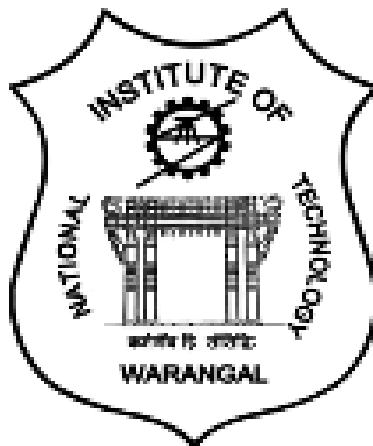
Abhishek Nalla-21ECB0B37

N.Sri Darshan-21ECB0B38

P.Mokshagna Reddy-21ECB0B39

Guided by:

Dr.P.Prithvi & Dr.V.Narendar



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION
ENGINEERING**

NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL

2022-23

ABSTRACT:

The objective of this project is to design and implement a Morse code detector using a field-programmable gate array (FPGA) and to display the transmitted characters on a 7-segment display. The input signals will be obtained through a push button interface. The FPGA will be programmed using Verilog hardware description language (HDL) to achieve the desired functionality.

Morse code is a communication system that represents alphanumeric characters using a sequence of dots and dashes. In this project, push buttons on the FPGA are used to input Morse code signals. According to the Morse code standard, a dot corresponds to one unit, and a dash corresponds to three units. The space between parts of the same letter is one unit, the space between letters is three units, and the space between different words is seven units. In binary code, these spaces are represented by low voltage (0), while dots and dashes are represented by high voltage (1).

The main concept is to establish a specific time period range to distinguish the signal as a dot, a dash, or a variety of spaces. When the push button is pressed (positive edge) or released (negative edge), a new clock cycle begins, and the duration of each clock pulse determines the sequence of dots and dashes. An asynchronous finite-state machine (FSM) is utilized to detect the input sequence, and the FPGA generates the output, which can be displayed on an LCD screen or a 7-segment display.

TABLE OF CONTENTS:

1.Introduction

2.Background

3.Applications

3.1.Clock

3.2.FSM in the Transmitter

3.3.FSM in the Receiver

4.Code

➤ **Elaborated Designs**

5.Applications

6.Conclusion

7.References

LIST OF FIGURES:

- Fig 1. International Morse Code
- Fig 2. State diagram of the FSM present in the transmitter
- Fig 3. State diagram of the FSM present in the receiver
- Fig 4. 7-Segment Display tables
- Fig 5. Elaborated Designs

CERTIFICATE

This is to certify that the dissertation work entitled IMPLEMENTATION OF MORSE CODE DETECTION ON FPGA is a Bonafide record of seminar work carried out by team members ABHISHEK NALLA(B37),N SRI DARSHAN(B38),P MOKSHAGNA REDDY(B39) submitted to the faculty of “Electronics and Communication Engineering Department”, in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in “Electronics and Communication Engineering” at National Institute of Technology, Warangal during the academic year (2022-23).

Dr.V. Narendar

Assistant Professor

**Department of Electronics and
Communication Engineering**

**National Institute of Technology,
Warangal**

Smt.P. Prithvi

Assistant Professor

**Department of Electronics and
Communication Engineering**

**National Institute of Technology,
Warangal**

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to our faculty in-charges Smt. P. Prithvi, D.V. Narendar, Assistant Professors, Department of Electronics and Communication Engineering, National Institute of Technology, Warangal for their constant supervision, guidance, suggestions and encouragement during this seminar & semester.

1.INTRODUCTION:

A continuous wave decoder, commonly known as a Morse code detector, is a tool or software designed to convert Morse code signals into text or audible sounds. Morse code is a form of communication that employs dots and dashes to represent letters and numerals.

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	• —	U	• • —
B	— • • •	V	• • • —
C	— • — •	W	• — —
D	— • •	X	— • • —
E	•	Y	— • — —
F	• • — •	Z	— — • •
G	— — •		
H	• • • •		
I	• •		
J	• — — —		
K	— • —		
L	• — • •		
M	— —		
N	— •		
O	— — —		
P	• — — •		
Q	— — • —		
R	• — •		
S	• • •		
T	—		
		1	• — — — —
		2	• • — — —
		3	• • • — —
		4	• • • • —
		5	• • • • •
		6	— • • • •
		7	— — • • •
		8	— — — • •
		9	— — — — •
		0	— — — — —

Fig.1-Interntional Morse Code

For radio operators, amateur radio enthusiasts, and emergency services personnel, Morse code detectors play a crucial role in decoding Morse code signals transmitted via radio waves. By analysing the audio or radio frequency signals, these tools can

recognize the patterns of dots and dashes that correspond to each letter or numeral in Morse code.

Morse code detectors come in various forms, ranging from hardware devices that connect to a radio or computer to software programs that run on a computer or mobile device. These tools have multiple applications, such as decoding emergency signals, monitoring Morse code transmissions, and enhancing Morse code communication proficiency.

A different variant of Morse code detector is a visual decoder that employs a flashing light or a set of LEDs to symbolize the dots and dashes of Morse code. This method enables the user to interpret the Morse code signal by observing the sequence of flashes.

Lastly, there exist computer algorithms capable of detecting Morse code signals from audio or video recordings. These algorithms utilize machine learning methods to identify the pattern of dots and dashes and convert them into written text.

In conclusion, Morse code detectors prove to be valuable instruments for amateur radio operators, emergency responders, and anyone who relies on Morse code communication. These devices are relatively straightforward to construct or acquire at a low cost, and they offer a dependable means to decipher Morse code signals.

2.BACKGROUND:

The origins of Morse code can be traced to the early 1800s when Samuel Morse, an American artist and inventor, created a telegraph system that employed a combination of dots and dashes to signify letters and numerals. This innovation enabled the swift transmission of messages across great distances, thereby transforming the field of communication during that era.

With the growing popularity of Morse code, several tools were invented to recognize and comprehend the signals. Initially, Morse code detectors were basic instruments that employed electromechanical parts to transform the dots and dashes into visible or audible cues.

Towards the end of the 1800s, advancements in electronic components, such as vacuum tubes, led to the production of advanced Morse code detectors. These devices were capable of boosting and filtering the signals, thereby facilitating the differentiation of the dots and dashes from the surrounding noise.

Morse code played a critical role in military communication during World War II, and specialized Morse code detectors were devised specifically for use by military personnel. These tools were frequently compact and mobile, enabling soldiers to transmit messages via Morse code across extended distances.

In the subsequent years, Morse code persisted as a communication method used by amateur radio enthusiasts, and a range of Morse code detectors were designed for this

application. With the emergence of digital technology, computer algorithms were created to identify and interpret Morse code signals in audio or video recordings.

In the present day, Morse code detectors are still employed by amateur radio operators, emergency personnel, and individuals requiring Morse code communication. Despite technological advancements, the fundamental concepts of detecting and deciphering Morse code signals remain unaltered.

3.DESIGN AND WORKING

3.1 CLOCK:

In order to create a 1Hz clock, we utilized the 50MHz clock available in the FPGA and implemented a clock divider. By employing a counter and a comparator, we were able to count up to 25,000,000 before resetting to zero. The count was then compared to 24,999,999, and the comparator generated a '1' output when both values matched. This output was directed to a flip flop, which toggled its output every half second, effectively generating a 1Hz clock signal.

3.2 FSM IN THE TRANSMITTER:

This particular FSM operates as a Moore machine with inputs that consist of 0s or 1s, and is designed to capture input data at each positive edge-triggered clock pulse. The input data is interpreted as either a dot, dash, or character space, depending on the timing of button presses or releases. Specifically, if a

single '1' is detected, flanked by zeros (010), the machine records a dot, whereas if three '1's are detected, flanked by zeros (01110), the machine records a dash.

The output of the FSM consists of two bits, which are as follows:

00: denotes the space between dashes and dots in a single letter (when the input is 0)

01: denotes a dot (when the input is 010)

10: denotes a dash (when the input is 01110)

11: denotes a character space (when the input is 000)

Finally, this 2-bit output is transmitted to a receiver for further processing.

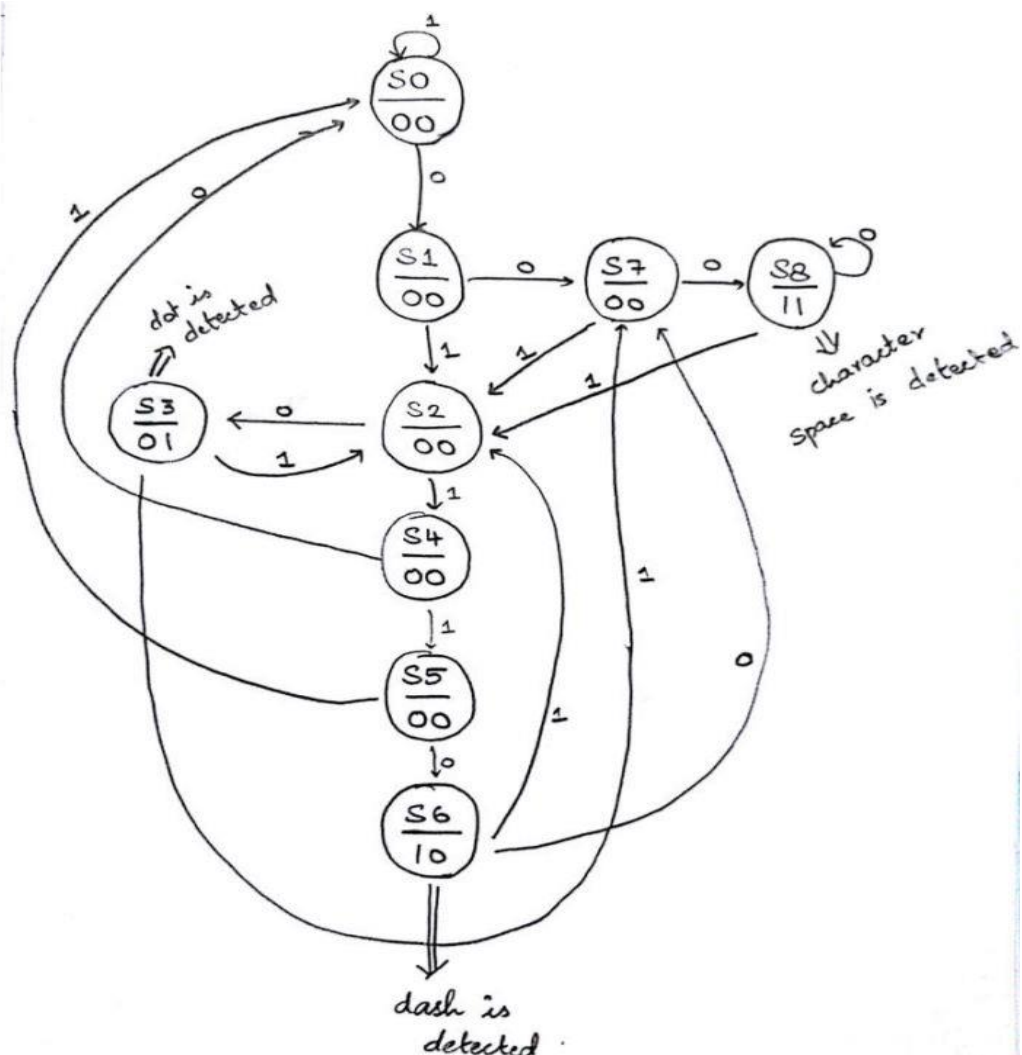


Fig.2- State diagram of the FSM present in the transmitter

3.3 FSM in the Receiver:

The receiver employs a Mealy machine as its FSM, which receives 2-bit inputs that are the outputs of the previous FSM. Similar to the previous FSM, the input is captured at every positive edge-triggered clock pulse. The primary function of this FSM is to detect a sequence of dots and dashes and generate the corresponding output. Upon receiving a character space input, the output is changed according to the corresponding state, and the state is reset.

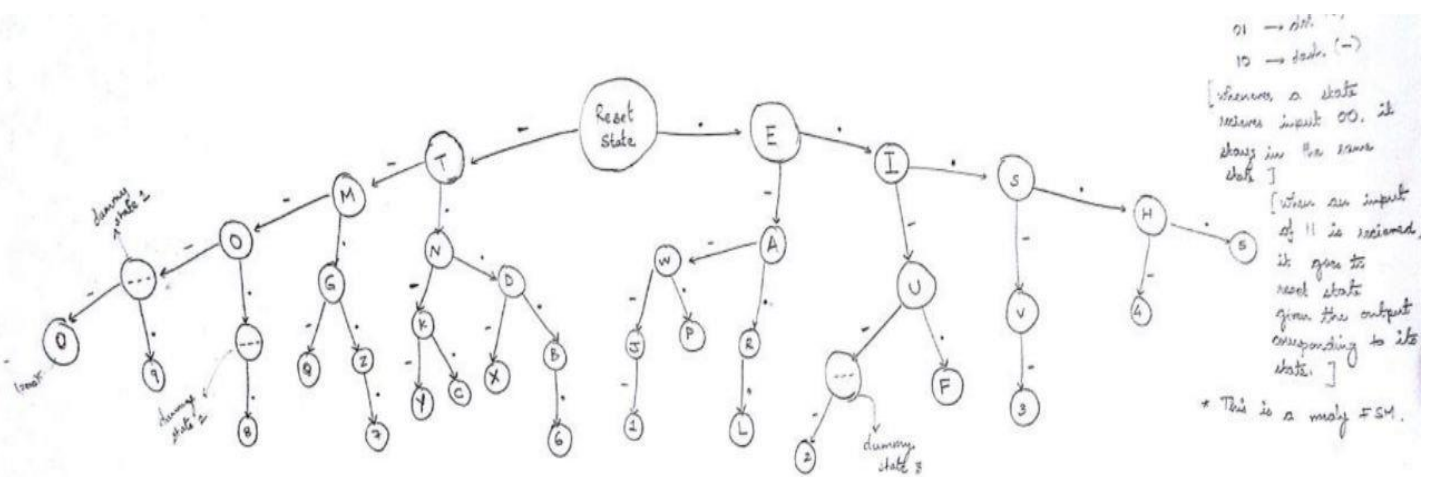













Fig.3- State diagram of the FSM present in the receiver

For the LED Display , we make use of the following table:

Character	In LED display	Value for 7 segment							In hex code	Morse Code
		a	b	c	d	e	f	g		
A		0	0	0	0	1	0	0	#03	. -
B		0	1	1	0	0	0	0	#60	- . . .
C		0	0	1	1	0	0	0	#31	- . . .
D		0	1	0	0	0	0	1	#42	- . .
E		0	0	1	1	0	0	0	#30	.
F		0	0	1	1	1	0	0	#38	. . . -
G		0	0	1	0	0	0	0	#21	- - .
H		0	1	1	0	1	0	0	#68
I		0	1	1	1	1	0	0	#79	. .
J		0	1	0	0	0	0	1	#43	. - - -
K		0	1	0	0	1	0	0	#48	- . -

character	In LED Display	Value for 7 segment							In hex code	Mouse code
		(dot)	a	b	c	d	e	f		
L		0	1	1	1	0	0	0	#71	...-
M		0	0	1	0	1	0	1	#2A	--
N		0	1	1	0	1	0	1	#6A	-.
O		0	1	1	0	0	0	1	#62	----
P		0	0	0	1	1	0	0	#18	.---.
Q		0	0	0	0	1	1	0	#0C	---.-
R		0	0	1	1	1	0	0	#39	.-.
S		0	0	1	0	0	1	1	#24	...
T		0	1	1	1	0	0	0	#70	-
U		0	1	0	0	0	0	1	#41	...-
V		0	1	0	1	0	1	0	#54	...-
W		0	0	1	0	0	0	1	#23	.--
X		0	1	0	0	1	0	0	#48	...-
Y		0	1	0	0	0	1	0	#44	---












character	In I/O Display	Value for 7 segments (a-g)	In hex code	Morse code
2		0 0 0 1 0 0 1 0	#12	--..
0		0 0 0 0 0 0 0 1	#01	-----
1		0 1 0 0 1 1 1 1	#0F	.-----
2		0 0 0 1 0 0 1 0	#12	..----
3		0 0 0 0 0 1 1 0	#06	...---
4		0 1 0 0 1 1 0 0	#4C-
5		0 0 1 0 0 1 0 0	#24
6		0 0 1 0 0 0 0 0	#20	-.....
7		0 0 0 0 1 1 1 1	#0F	--...
8		0 0 0 0 0 0 0 0	#00	-----
9		0 0 0 0 0 1 0 0	#04	-----.

Fig .4- 7-Segment Display tables

4.CODE:

```
module clock_divider(in_clk, rst,
out_clk);
input in_clk;
input rst;
output reg out_clk;
localparam cnst = 25000000;
reg [31:0] count;
```

```
always@(posedge in_clk, posedge rst)
```

```
begin
```

```
if(rst == 1'b1)
```

```
count <= 32'b0;
```

```
else if (count == cnst - 1)
```

```
count <= 32'b0;
```

```
else
```

```
count <= count + 1;
```

```
end
```

```
always@(posedge in_clk, posedge rst)
```

```
begin
```

```
if(rst == 1'b1)
```

```
out_clk <= 1'b0;
```

```
else if(count == cnst - 1)
```

```
out_clk <= ~out_clk;
```

```
else
```

```
out_clk <= out_clk;
```

```
end
```

```
endmodule
```

```
module rec_fsm(mid_clk, s_out, clk, rst,
serial_inp, en, dot);
```

```
input serial_inp;
```

```
input rst;
```

```
input clk;
```

```
reg [7:0]state = 8'hff;
```

```
wire [1:0]p_in;
```

```
output reg [7:0]s_out;
```

```
output mid_clk;
```

```
output [3:0]en;
```

```
output dot;
```

```
parameter [7:0]reset_state=8'hff;
```

```
parameter [7:0]a = 8'h00;
```

```
parameter [7:0]b = 8'h01;
```

```
parameter [7:0]c = 8'h02;
```

```
parameter [7:0]d = 8'h03;
```

```
parameter [7:0]e = 8'h04;
```

```
parameter [7:0]f = 8'h05;
```

```
parameter [7:0]g = 8'h06;
```

```
parameter [7:0]h = 8'h07;
```

```
parameter [7:0]i = 8'h08;
```

```
parameter [7:0]j = 8'h09;
```

```
parameter [7:0]k = 8'h0a;
```

```
parameter [7:0]l = 8'h0b;
```

```
parameter [7:0]m = 8'h0c;
```

```
parameter [7:0]n = 8'h0d;
```

```
parameter [7:0]o = 8'h0e;
```

```
parameter [7:0]p = 8'h0f;
```



```

parameter [7:0]q = 8'h10;
parameter [7:0]r = 8'h11;
parameter [7:0]s = 8'h12;
parameter [7:0]t = 8'h13;
parameter [7:0]u = 8'h14;
parameter [7:0]v = 8'h15;
parameter [7:0]w = 8'h16;
parameter [7:0]x = 8'h17;
parameter [7:0]y = 8'h18;
parameter [7:0]z = 8'h19;
parameter [7:0]zero = 8'h20;
parameter [7:0]one = 8'h21;
parameter [7:0]two = 8'h22;
parameter [7:0]three = 8'h23;
parameter [7:0]four = 8'h24;
parameter [7:0]five = 8'h25;
parameter [7:0]six = 8'h26;
parameter [7:0]seven = 8'h27;
parameter [7:0]eight = 8'h28;
parameter [7:0]nine = 8'h29;

parameter [7:0]ds1 = 8'h2a;
parameter [7:0]ds2 = 8'h2b;
parameter [7:0]ds3 = 8'h2c;

clock_divider clk_rec(
.in_clk(clk), //may be prob
.rst(rst),
.out_clk(mid_clk)
);

trans_fsm trans_rec(
.serial_inp(serial_inp),

.rst(rst),
.parallel_out(p_in),
.clk(clk) //may be prob
);

assign en=4'b1110;
assign dot = 1'b1;

always@(posedge mid_clk)
begin

case(state)

reset_state: //for reset state
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end

2'b01:begin
state<=e;
s_out<=8'hff;end

2'b10:begin
state<=t;
s_out<=8'hff;end

2'b11:begin
state<=reset_state;
s_out<=8'hff;end
endcase

a: //for a
case(p_in)

```

```
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=r;
s_out<=8'hff;end
```

```
2'b10:begin
state<=w;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h08;end
endcase
```

```
b: //for b
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=six;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h60;end
```

```
endcase
```

```
c: //for c
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h31;end
endcase
```

```
d: //for d
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=b;
s_out<=8'hff;end
```

```
2'b10:begin
state<=x;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h42;end
endcase
```

```
e: //for e
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=i;
s_out<=8'hff;end
```

```
2'b10:begin
state<=a;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h30;end
endcase
```

```
f: //for f
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h38;end
endcase
```

```
g: //for g
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=z;
s_out<=8'hff;end
```

```
2'b10:begin
state<=q;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h21;end
endcase
```

```
h: //for h
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

2'b01:begin	case(p_in)
state<=five;	2'b00:begin
s_out<=8'hff;end	state<=state;
	s_out<=8'hff;end
2'b10:begin	2'b01:begin
state<=four;	state<=reset_state;
s_out<=8'hff;end	s_out<=8'hff;end
2'b11:begin	2'b10:begin
state<=reset_state;	state<=one;
s_out<=8'h68;end	s_out<=8'hff;end
endcase	
i: //for reset state	2'b11:begin
case(p_in)	state<=reset_state;
2'b00:begin	s_out<=8'h43;end
state<=state;	endcase
s_out<=8'hff;end	k: //for k
2'b01:begin	case(p_in)
state<=s;	2'b00:begin
s_out<=8'hff;end	state<=state;
	s_out<=8'hff;end
2'b10:begin	2'b01:begin
state<=u;	state<=c;
s_out<=8'hff;end	s_out<=8'hff;end
2'b11:begin	2'b10:begin
state<=reset_state;	state<=y;
s_out<=8'h79;end	s_out<=8'hff;end
endcase	
j: //for j	2'b11:begin
	state<=reset_state;

```
s_out<=8'h48;end  
endcase
```

```
l: //for l  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end  
  
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h71;end  
endcase
```

```
m: //for m  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=g;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=o;
```

```
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h2a;end  
endcase
```

```
n: //for reset state  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=d;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=k;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h6a;end  
endcase
```

```
o: //for o  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=ds1;
```

```
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h63;end  
endcase
```

```
p: //for p  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h18;end  
endcase
```

```
q: //for q  
case(p_in)  
2'b00:begin  
state<=state;
```

```
s_out<=8'hff;end
```

```
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h0c;end  
endcase
```

```
r: //for r  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=1;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h39;end  
endcase
```

```
s: //for o
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=h;
s_out<=8'hff;end
```

```
2'b10:begin
state<=v;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h24;end
endcase
```

```
t: //for t
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=n;
s_out<=8'hff;end
```

```
2'b10:begin
state<=m;
s_out<=8'hff;end
```

```
2'b11:begin
```

```
state<=reset_state;
s_out<=8'h70;end
endcase
```

```
u: //for u
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=f;
s_out<=8'hff;end
```

```
2'b10:begin
state<=ds3;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h41;end
endcase
```

```
v: //for v
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
```

```
state<=three;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h54;end  
endcase
```

```
w: //for o  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=p;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=j;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h23;end  
endcase
```

```
x: //for x  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin
```

```
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h48;end  
endcase
```

```
y: //for y  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h44;end  
endcase
```

```
z: //for z  
case(p_in)  
2'b00:begin
```



```
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=seven;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h12;end  
endcase
```

```
one: //for 1  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h4f;end  
endcase
```

```
two: //for 1  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin  
state<=reset_state;  
s_out<=8'h12;end  
endcase
```

```
three: //for 3  
case(p_in)  
2'b00:begin  
state<=state;  
s_out<=8'hff;end
```

```
2'b01:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b10:begin  
state<=reset_state;  
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h06;end
endcase
```

```
four: //for 4
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h4c;end
endcase
```

```
five: //for 5
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h24;end
endcase
```

```
six: //for 6
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h20;end
endcase
```

```
seven: //for 7
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h0f;end
endcase
```

```
eight: //for 8
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h00;end
endcase
```

```
nine: //for 1
case(p_in)
```

```
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h08;end
endcase
```

```
zero: //for 1
case(p_in)
2'b00:begin
state<=state;
s_out<=8'hff;end
```

```
2'b01:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b10:begin
state<=reset_state;
s_out<=8'hff;end
```

```
2'b11:begin
state<=reset_state;
s_out<=8'h01;end
```

endcase

ds1: //for ds1

case(p_in)

2'b00:begin

state<=state;

s_out<=8'hff;end

2'b01:begin

state<=eight;

s_out<=8'hff;end

2'b10:begin

state<=reset_state;

s_out<=8'hff;end

2'b11:begin

state<=reset_state;

s_out<=8'hff;end

endcase

ds2: //for ds2

case(p_in)

2'b00:begin

state<=state;

s_out<=8'hff;end

2'b01:begin

state<=nine;

s_out<=8'hff;end

2'b10:begin

state<=zero;

s_out<=8'hff;end

2'b11:begin

state<=reset_state;

s_out<=8'hff;end

endcase

ds3: //for 1

case(p_in)

2'b00:begin

state<=state;

s_out<=8'hff;end

2'b01:begin

state<=reset_state;

s_out<=8'hff;end

2'b10:begin

state<=two;

s_out<=8'hff;end

2'b11:begin

state<=reset_state;

s_out<=8'hff;end

endcase

default: //for default

case(p_in)

2'b00:begin

state<=state;

s_out<=8'hff;end

2'b01:begin

state<=reset_state;

s_out<=8'hff;end

```

2'b10:begin
state<=reset_state;
s_out<=8'hff;end

```

```

2'b11:begin
state<=reset_state;
s_out<=8'hff;end
endcase
endcase
end
endmodule

```

```

module      trans_fsm(serial_inp,
parallel_out, clk,rst);

```

```

input serial_inp;
input clk;
input rst;

```

```

wire mid_clk; //we changed this
reg [3:0]state=4'b0000;
//reg [3:0]next_state=4'b0000;

```

```

output reg [1:0]parallel_out;

```

```

parameter [3:0] s0 = 4'b0000;
parameter [3:0] s1 = 4'b0001;
parameter [3:0] s2 = 4'b0010;
parameter [3:0] s3 = 4'b0011;
parameter [3:0] s4 = 4'b0100;
parameter [3:0] s5 = 4'b0101;
parameter [3:0] s6 = 4'b0110;
parameter [3:0] s7 = 4'b0111;

```

```

parameter [3:0] s8 = 4'b1000;

```

```

clock_divider clk_trans(
.in_clk(clk),
.rst(rst),
.out_clk(mid_clk)
);

```

```

always@(posedge mid_clk)

```

```

begin
/*if(rst) //check
parallel_out<=2'b00;*/

```

```

case(state)

```

```

s0:
begin
parallel_out<=2'b00;
if(~serial_inp)
state<=s1;
else
state<=s0;
end

```

```

s1:
begin
parallel_out<=2'b00;
if(~serial_inp)
state<=s7;
else
state<=s2;
end

```

```

s2:                                     end
begin
parallel_out<=2'b00;
if(~serial_inp)
state<=s3;
else
state<=s4;
end

s3:
begin
parallel_out<=2'b01;
if(~serial_inp)
state<=s7;
else
state<=s2;
end

s4:
begin
parallel_out<=2'b00;
if(~serial_inp)
state<=s0;
else
state<=s5;
end

s5:
begin
parallel_out<=2'b00;
if(~serial_inp)
state<=s6;
else
state<=s0;

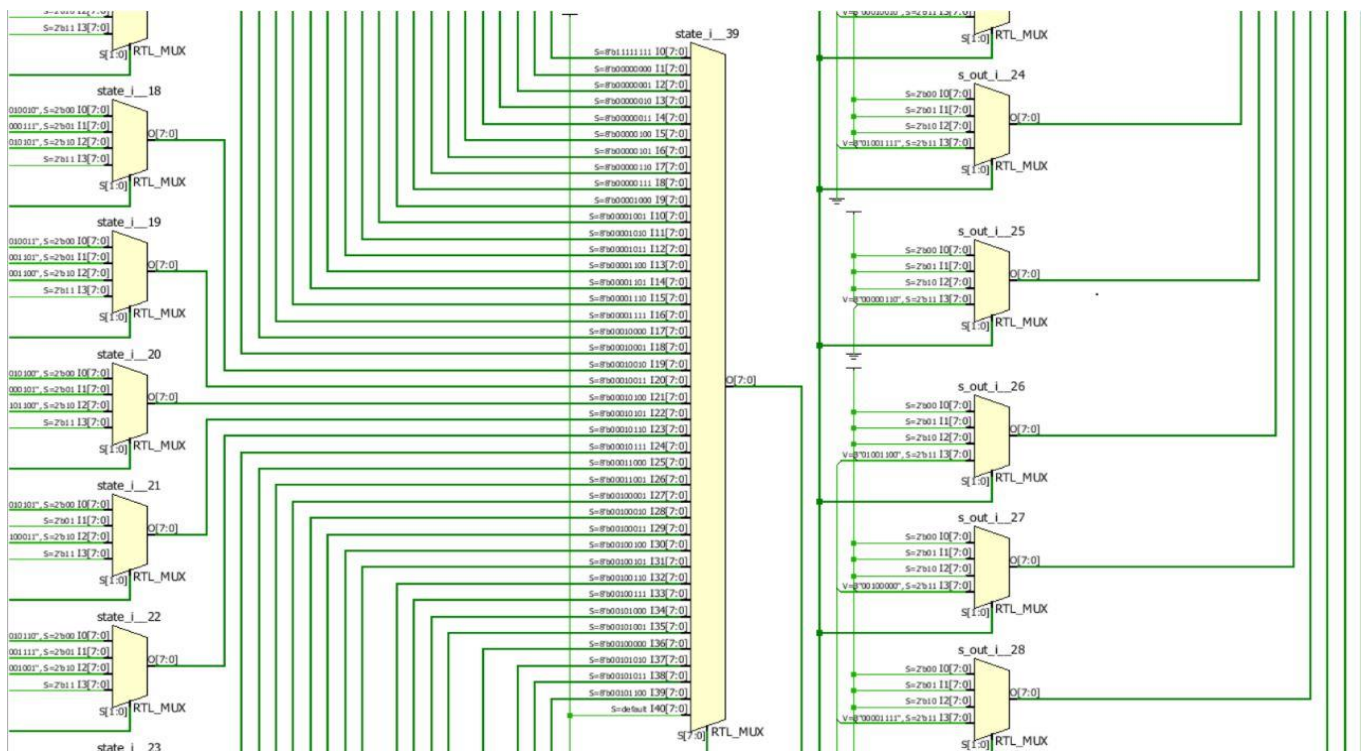
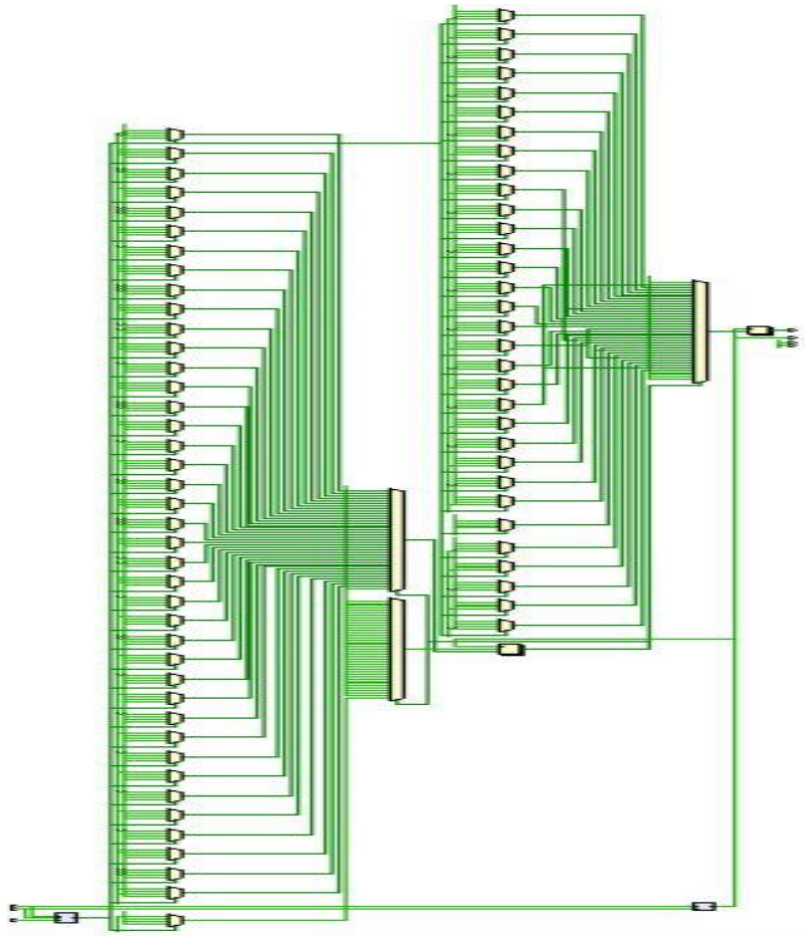
s6:
begin
parallel_out<=2'b10;
if(~serial_inp)
state<=s7;
else
state<=s2;
end

s7:
begin
parallel_out<=2'b00;
if(~serial_inp)
state<=s8;
else
state<=s2;
end

s8:
begin
parallel_out<=2'b11;
if(~serial_inp)
state<=s8;
else
state<=s2;
end
endcase
end
endmodule

```

ELABORATED DESIGN:



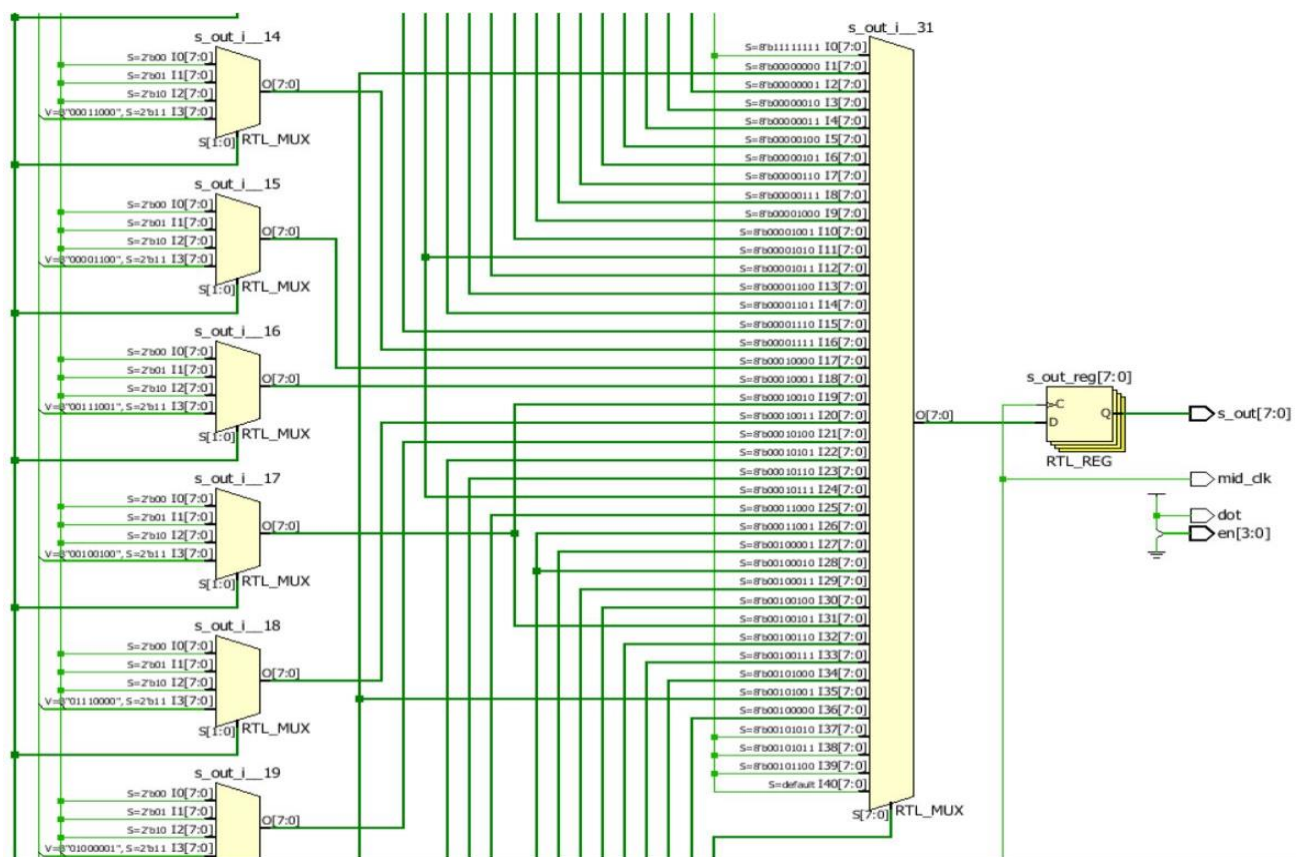
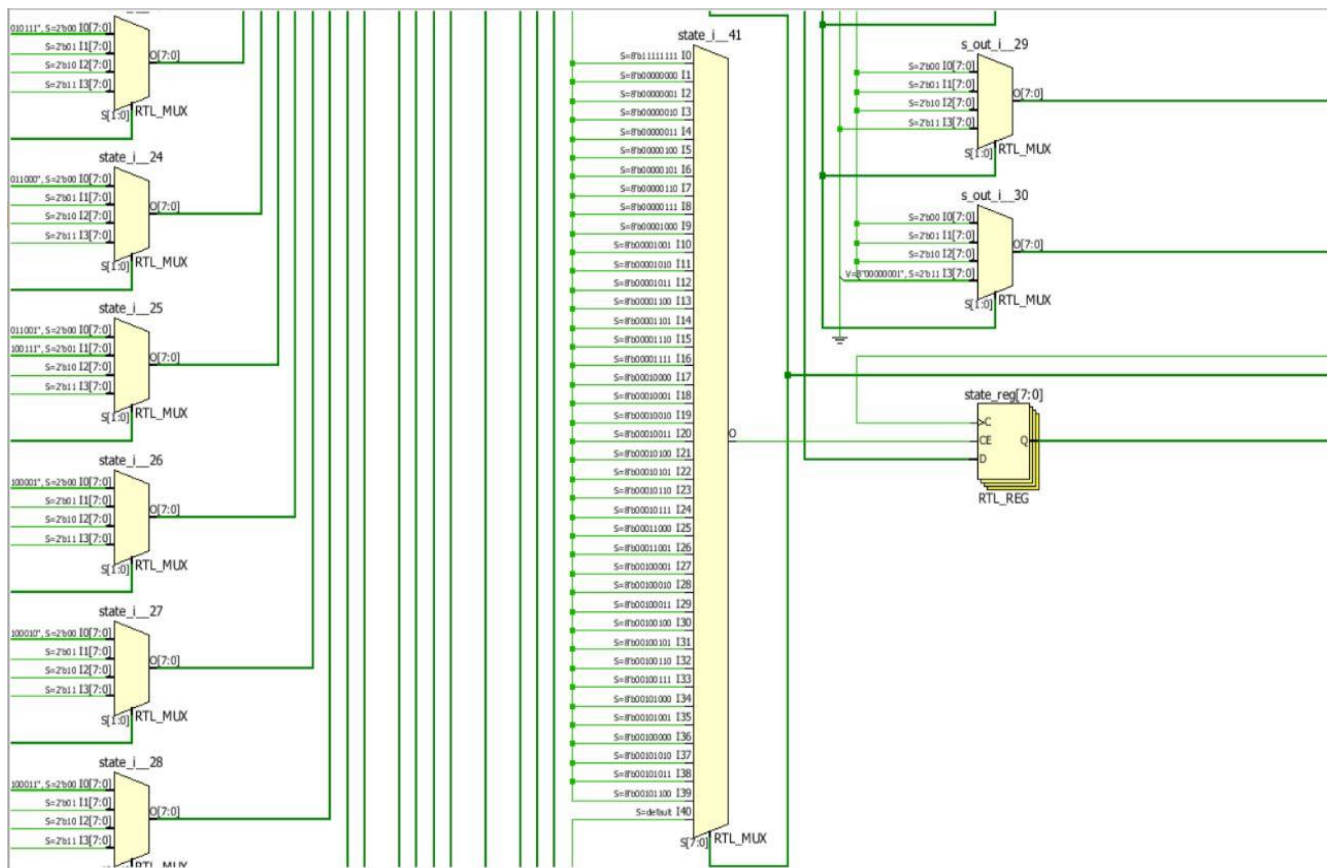










Fig.5



SIMULATIONS :



#	Name	Value	0.000 ns	20.000 ns	40.000 ns
1	>  in[15:0]	1111000000	11110000000110001	1111000000010110	1111000000011110
2	>  out[15:0]	1101010100	11010000000000000	1011011011011000	1010110110110000

#	Name	Value	80.000 ns	100.000 ns	1
1	>  in[15:0]	1111000000	11110000000100110	11110000000100101	11110000001000110
2	>  out[15:0]	1101010100	10101011011000000	10101010110000000	1101101101101000

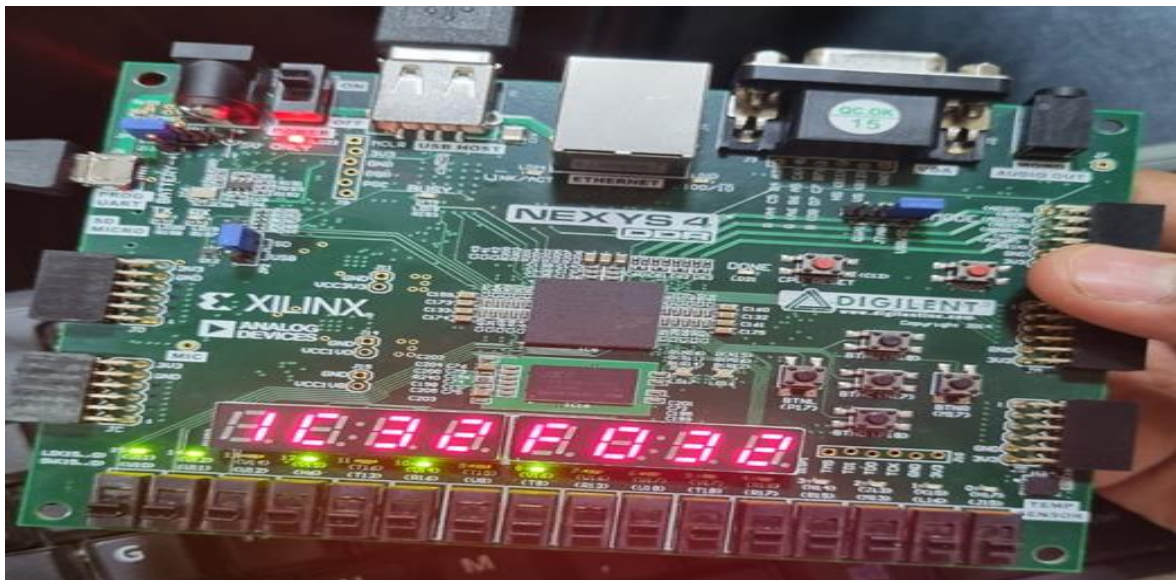
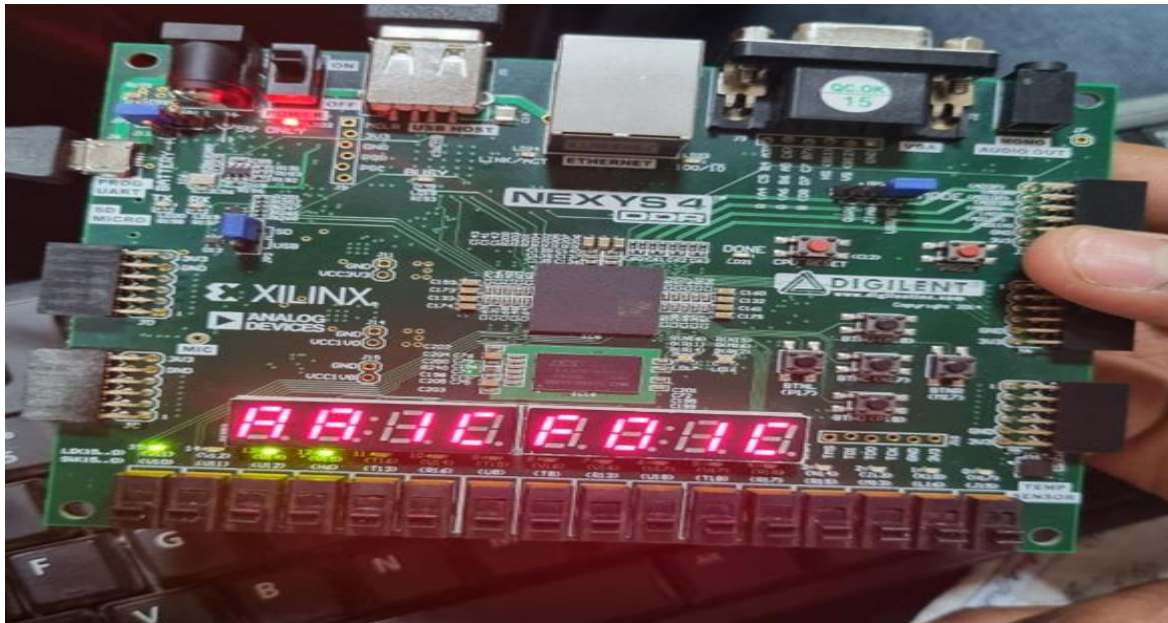
#	Name	Value	120.000 ns	130.000 ns	140.000 ns	150.000 ns
1	>  in[15:0]	1111000000	1111000000010101		1111000000011101	
2	>  out[15:0]	1101010100	11011010110000000		10110110000000000	

#	Name	Value	180.000 ns	200.000 ns	220.000 ns
1	>  in[15:0]	1111000000	11110000000101101	11110000000111100	11110000001000011
2	>  out[15:0]	1101010100	10110100000000000	10101100000000000	10100000000000000

#	Name	Value	360.000 ns	380.000 ns	400.000 ns
1	>  in[15:0]	1111000000	11110000000...	111100000001...	111100000001...
2	>  out[15:0]	1101010100	101010000000...	110101000000...	101011010000...

#	Name	Value	450.000 ns	500.000 ns
1	>  in[15:0]	1111000000	11110... 11110... 11110... 11110...	
2	>  out[15:0]	1101010100	10101... 11011... 11011... 11010...	

FPGA IMPLEMENTATION:



5.APPLICATIONS:

Here are some applications of Morse code detectors:

1. Amateur radio communication: Morse code is still widely used by amateur radio operators around the world. Morse code detectors can be used to decode signals sent by these operators.

2. Military and emergency communication: Morse code is a standard means of communication in military and emergency situations. Morse code detectors can be used by soldiers and emergency responders to decode distress signals and other important messages.

3. Morse code training: Morse code is still taught in some schools and used by some organizations for communication. Morse code detectors can be used by students and trainees to practice and improve their skills.

4. Historical research: Morse code has a rich history, and some researchers may use Morse code detectors to decode historical messages sent through telegraph or other means.

5. Electronic experimentation: Morse code detectors can be used by electronics enthusiasts to experiment with radio communication and Morse code encoding and decoding.

Overall, Morse code detectors have a wide range of applications in various fields, including communication, emergency response, education, research, and experimentation.

6.CONCLUSION:

In conclusion, Morse code detectors play an important role in various fields such as amateur radio communication, military and emergency communication, Morse code training, historical research, and electronic experimentation. They allow users to decode Morse code signals sent over the radio or through other means of communication, helping to ensure that important messages are properly received and understood. Morse code may be an old form of communication, but it still has practical uses today, and Morse code detectors continue to be valuable tools for those who need to communicate using this system.

7.REFERENCES:

Here are some references related to Morse code detectors:

1. "Morse code decoders and related technologies" by Dr. J.N. Pearce, Radio Science Bulletin, No. 322, June 2007.
2. "A CW Decoder Using the Goertzel Algorithm" by John L. Melton, QST Magazine, January 2001.
3. "A Compact and Simple Morse Code Decoder" by Owen Duffy, Electronics Australia Magazine, January 2006.
4. "Arduino-Based Morse Code Decoder" by Javier Galán, Ham Radio Magazine, January 2016.
5. "Design and Implementation of a Morse Code Decoder Using FPGA" by Karim Mohamed and Ahmed Zahran, Journal of Signal and Information Processing, Vol. 4, No. 3, August 2013.

These references provide information about different types of Morse code detectors, their design and implementation, and their applications. They can be helpful for researchers, engineers, and enthusiasts interested in understanding and building Morse code detectors.