Sri Lanka Institute of Information Technology

Models of Computing Assignment 2

Peiris S.Y.

EN15521576

Table of Contents

List of Figures

1. Introduction

State diagrams can be used to model the data flow of complex algorithms. The state diagram for a single elevator controller was designed and tested during this assignment. The states and inputs were modelled after studying the operation of the elevators and making assumptions where necessary. After testing on JFLAP, a VHDL code was developed and tested on a FPGA board. The state transitions were also observed using a simulation software.

2. Operation of the Elevator

Elevators are useful mechanisms that help people ascend or descend into different floors without much physical effort. The controller of an elevator is designed to be efficient, thus it responds to different user requests by considering its current state and the requested states depending on their order. In reality these controllers utilize higher level programming structures and memory elements (to remember each request and travel in order) and there are numerous variables that effect the decisions made by the controller, such as priority requests and timers. For the sake of this experiment, the functions carried out by the elevator controller are defined below,

Functions of the Elevator and Assumptions made when designing the controller

- Let us assume that there are 7 floors **(floorg, floor1, floor2, floor3, floor4, floor5, floor6).** The inputs switches inside the elevator are **(0,1,2,3,4,5,6)** and it is assumed that the up and down buttons on each floor produce the same signal as the switches (corresponding to the respective floor) inside the elevator. In reality the controller should know which button (requesting to go up, or down) on each floor was pressed, and only visit that floor depending on the ascending or descending nature. However, it was noted that the elevator at SLIIT would still open its doors on a floor which has requested to go to a higher floor whilst the lift is descending.

  This is one critical assumption that was made in order to decrease the number of states and make the VHDL code less complex. There is also a timer input **(x)**. The timer will begin incrementing when there are no more requests to service, and turns input **x** high when the preset value is met. When **x** is high (when the lift is idle in a floor for some time) the controller sends the lift to a descending state corresponding with its current floor. The timer is reset and begins incrementing again. When there are no more further requests even in descending state, the controller sends the elevator to floorg. Again at floorg, if the timer preset is met, the elevator is set to an idle state. The flow diagram below illustrates the steps taken by the controller when the elevator is at floor n and there are no more requests to service.

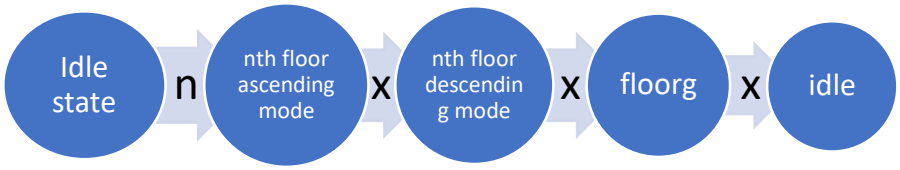

Figure 1 : Timer input determines the ascending or descending mode

When in an ascending mode, the controller will only service ascending requests. When in a descending mode, the controller will only service descending requests. The controller will always return to floorg before ascending again. This is another major assumption made.

- When any up or down button in any floor is pressed while in idle state(no down button in floorg and no up button in floor 6, the controller sends the elevator to the required floor. Pressing any buttons (0,1,2,3,4,5,6) inside the lift also results in a similar behaviour.
- When in any floor, the elevator will service the requests by **depending on the ascending or descending nature of the elevator.** If there are no requests to be stopped at a floor the elevator will skip it. In reality elevators maintain a **memory element** that changes with new requests, even while travelling to a floor. This can be easily implemented in a few lines of high level **sequential logic** running on a **processor**. (A similar process is demonstrated using python **using two states and recursion**) However this service routine becomes unnecessarily complicated and results in a large number of states and transitions when using VHDL. (Since sequential logic in an FPGA is implemented using states and transitions) Thus a much simpler lift controller was implemented on VHDL.
- It is also assumed that when the controller is turned on, the elevator is at floorg.

3. DFA Derivation

I. 5-tuple for the DFA

The 5 tuple is as follows.

- Q is a finite set called the states
- Σ is a finite set called the alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states

1. States (Q)

   Here q0 is idle, states with a at the end are in ascending mode, and states with d at the end are in descending mode.

Q = {I,q0,q1a,q2a,q3q,q4a,q5a,q6,q5d,q4d,q3d,q2d,q1d}

2. Alphabet

Σ = {0,1,2,3,4,5,6,x}

3. Table of Transition

δ	0	1	2	3	4	5	6	x
Q0	Q0	Q1a	Q2a	Q3a	Q4a	Q5a	Q6a	I
Q1a	Q1a	Q1a	Q2a	Q3a	Q4a	Q5a	Q6a	Q1d
Q2a	Q2a	Q2a	Q2a	Q3a	Q4a	Q5a	Q6a	Q2d
Q3a	Q3a	Q3a	Q3a	Q3a	Q4a	Q5a	Q6a	Q3d
Q4a	Q4a	Q4a	Q4a	Q4a	Q4a	Q5a	Q6a	Q4d
Q5a	Q5a	Q5a	Q5a	Q5a	Q5a	Q5a	Q6a	Q5d
Q6	Q0	Q1d	Q2d	Q3d	Q4d	Q5d	Q6a	Q0
Q5d	Q0	Q1d	Q2d	Q3d	Q4d	Q5d	Q5d	Q0
Q4d	Q0	Q1d	Q2d	Q3d	Q4d	Q4d	Q4d	Q0
Q3d	Q0	Q1d	Q2d	Q3d	Q3d	Q3d	Q3d	Q0
Q2d	Q0	Q1d	Q2d	Q2d	Q2d	Q2d	Q2d	Q0
Q1d	Q0	Q1d	Q1d	Q1d	Q1d	Q1d	Q1d	Q0

Table 1: Table of Transitions

I.e : Red colored state transitions indicate the disregarding of a descending request when in ascending mode, or an ascending request when in descending mode.

4. Starting State (Q)

   Q = (I)

5. Final State (F)

F = {I}

4. State Diagram

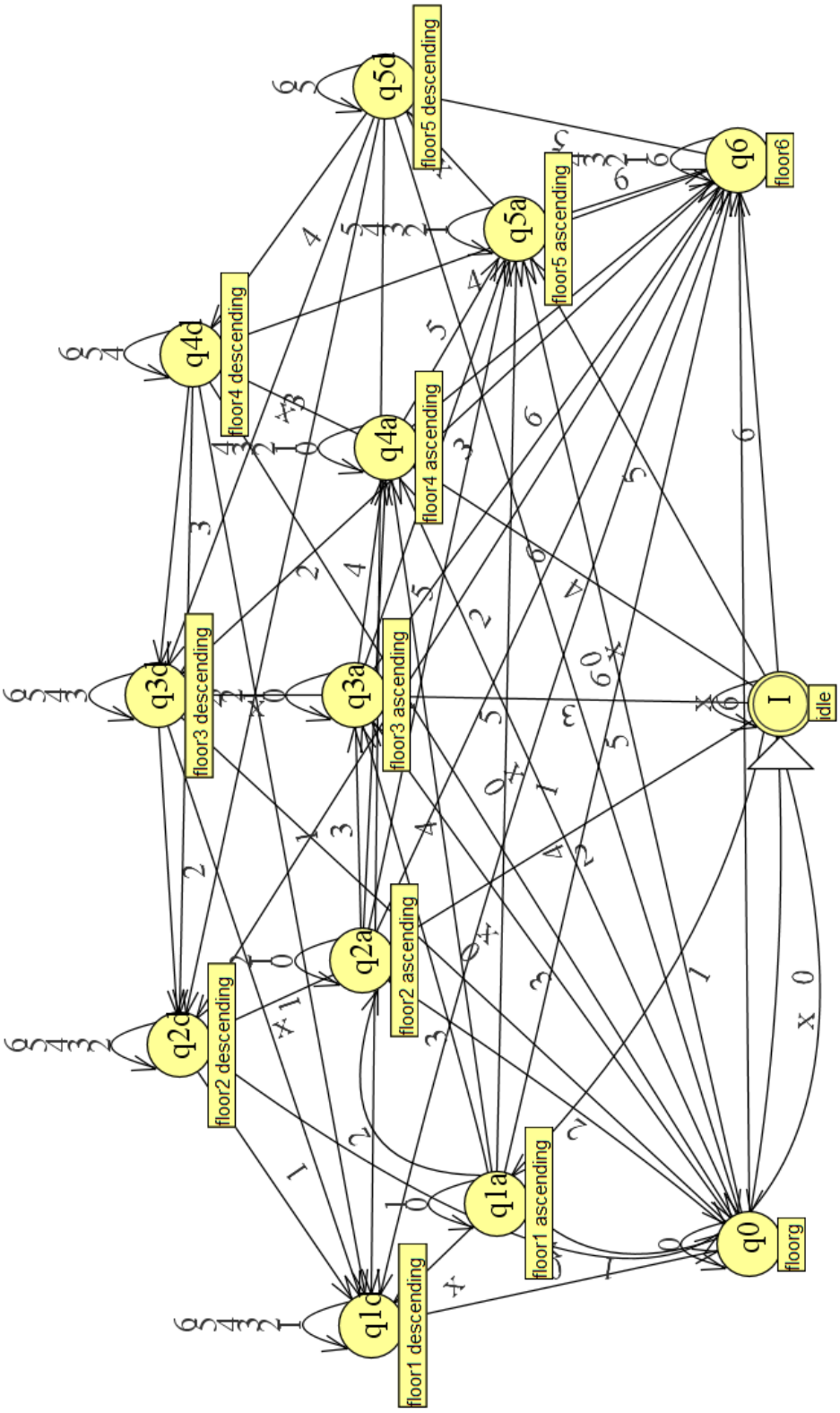

Figure 2 State Diagram

5. Test cases

I. JFLAP

1. Staying at Idle for ever

x x x x x x	Accept

Figure 3

2. Moving from idle to floor1, then returning to floorg and becoming idle due to no more requests. (Idling from timer input)

1 x x x	Accept

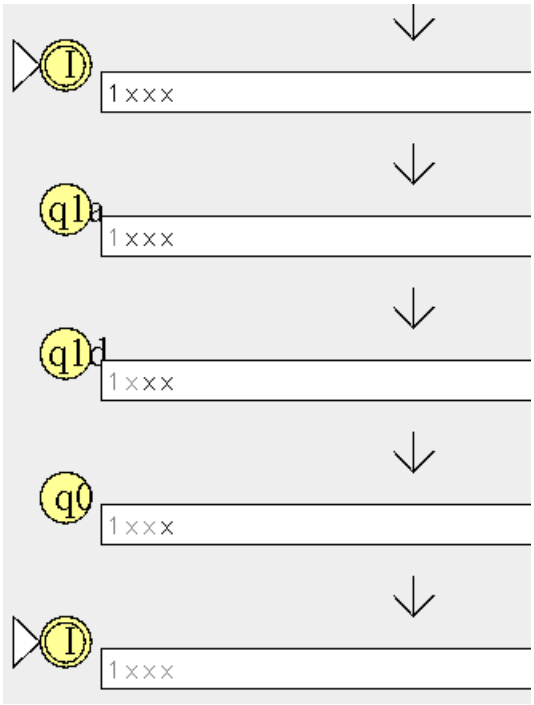

Figure 4

3. Moving up, then moving down after reaching floor6 or after getting no more ascend requests.

1 2 3 4 5 6 5 4 3 2 1 0 x	Accept
2 4 x 2 0 x	Accept
4 5 x 1 x x	Accept

Figure 5

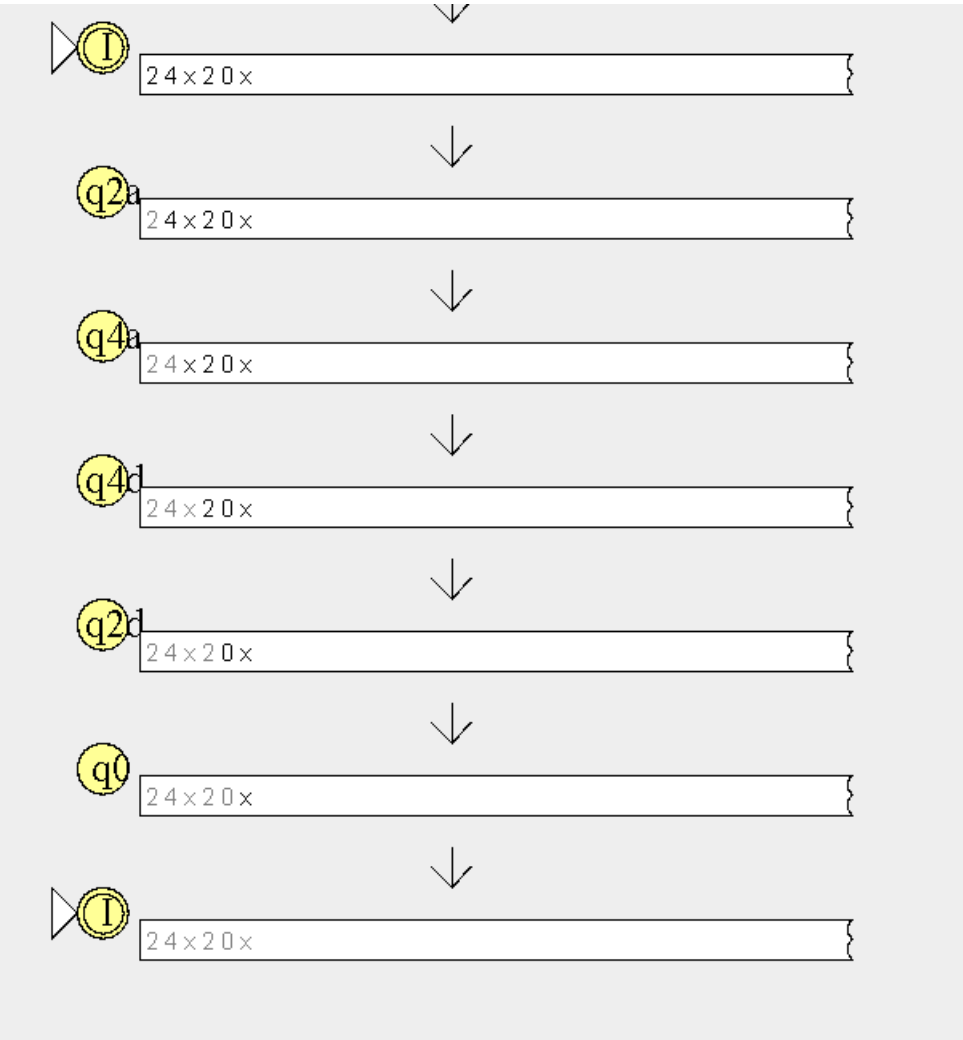

Figure 6 Trace for the second scenario

4. Starting from idle and ascending, then **disregarding** a descend request to floor3, and receiving timer inputs to go to idle due to no more requests.

1 2 4 3 x x x	Accept

Figure 7

When we observe the trace for this, we can see the controller disregarding the request to descend when in an ascending mode. Thus it is apparent that the **timer component x** is of utmost importance, since it stops the state transition from a deadlock or from 'choking'. By observing a successful ascending and descending scenario this can be further explained.

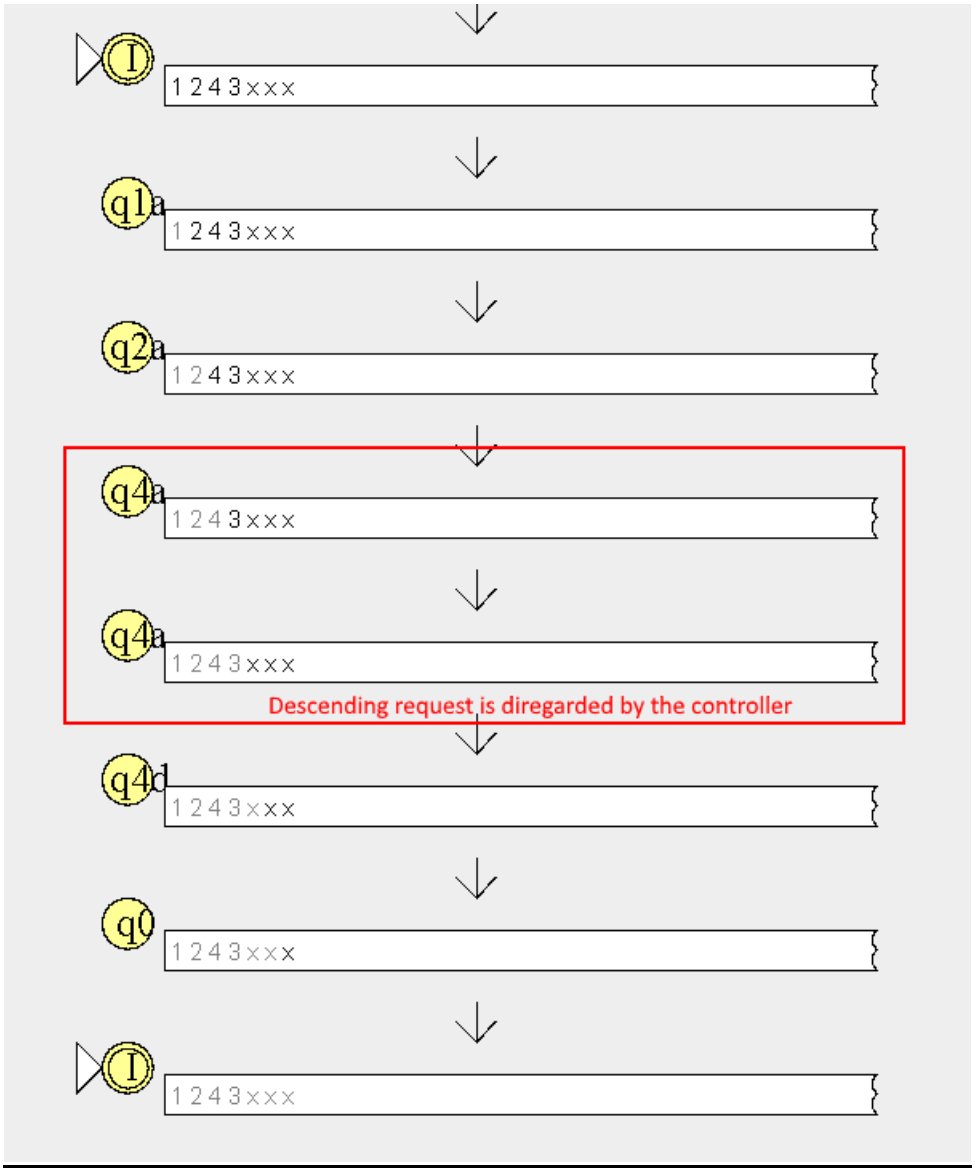

Figure 8

5. Ascending, then switching to descending mode due to no more ascend requests, then descending down to floorg due to user requests, then idling.

1 2 3 4 5 6 5 4 3 2 1 0 x	Accept

Figure 9

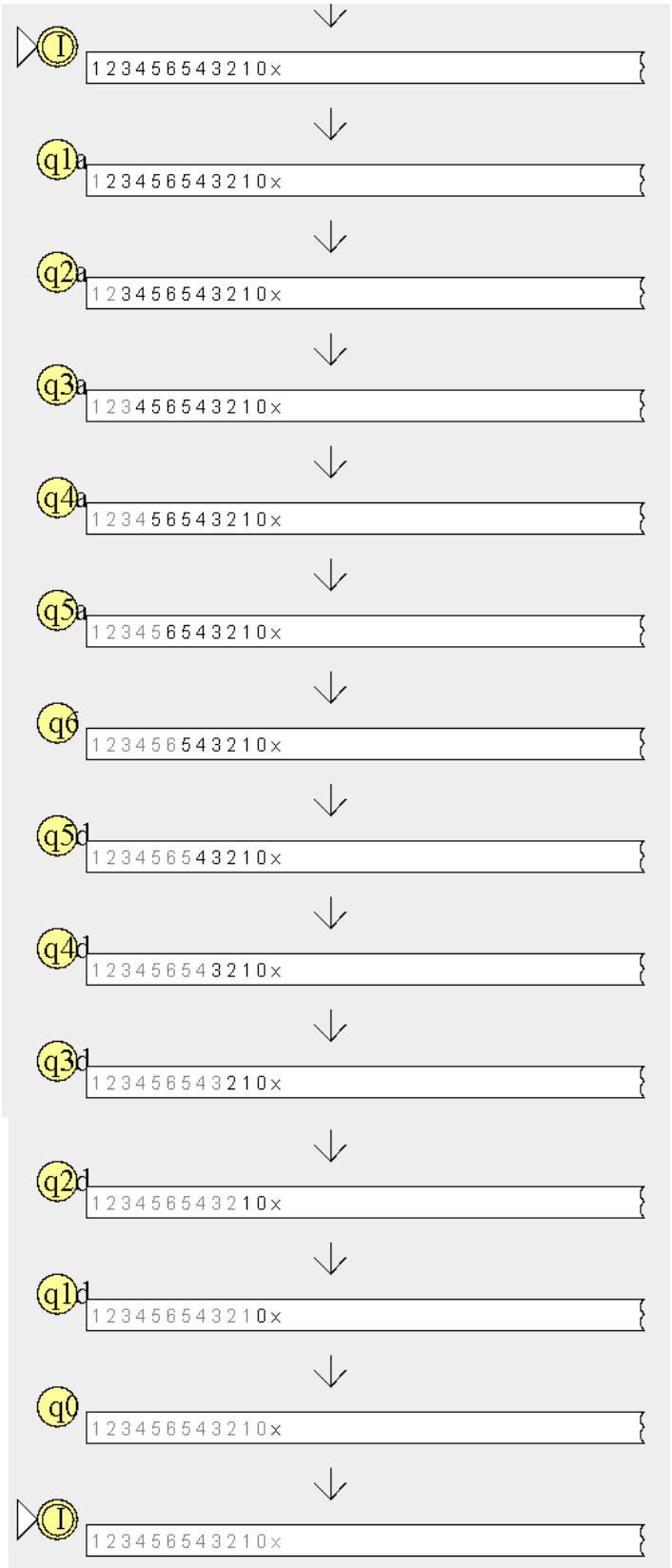

Figure 10 Trace for successful ascend and descend operation

6. Ascending, descending then receiving an ascend request. The ascend request is disregarded and the controller keeps descending according to user input and timer input.

Figure 11

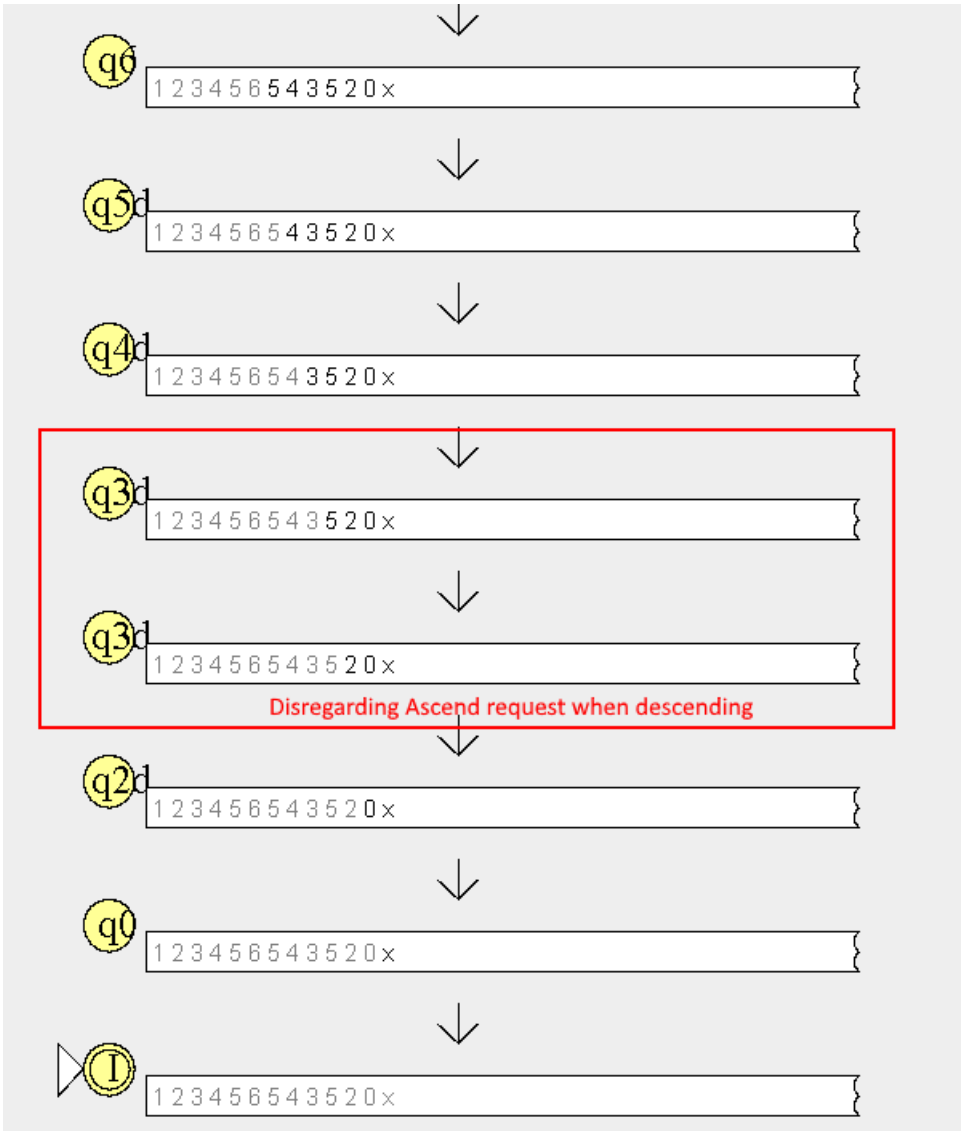

Figure 12

II. FPGA

The state diagram developed above was implemented on a FPGA board using VHDL code. The outputs were observed using ModelSim and with the use of LEDs. The ModelSim Simulations are shown below. The code is available in the appendix.

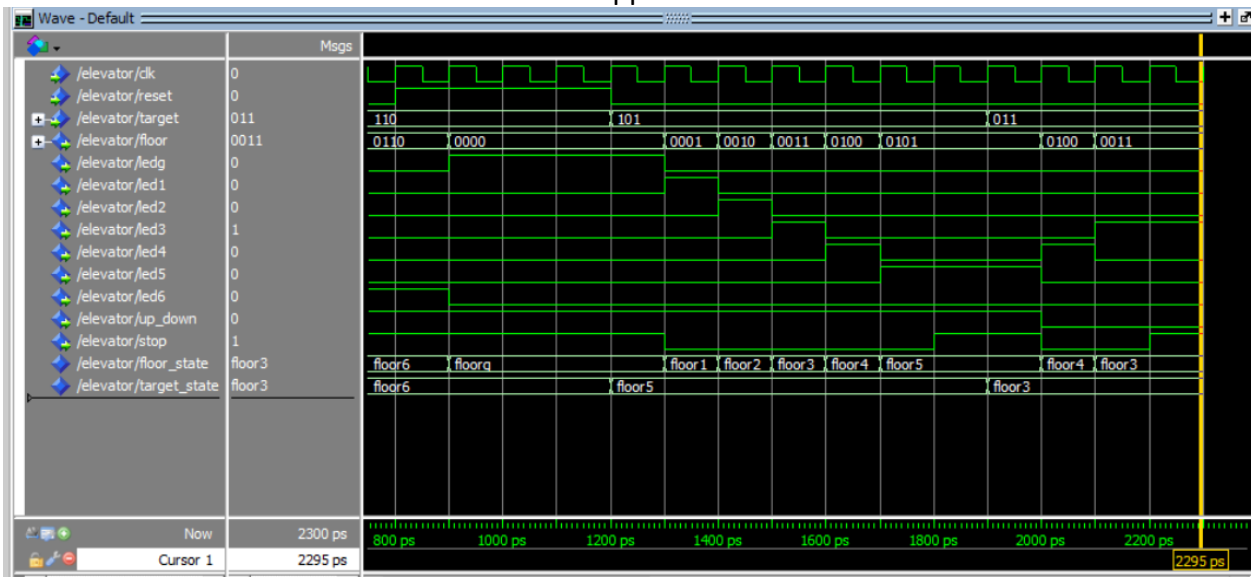

Figure 13 Modelsim Ascending Scenario

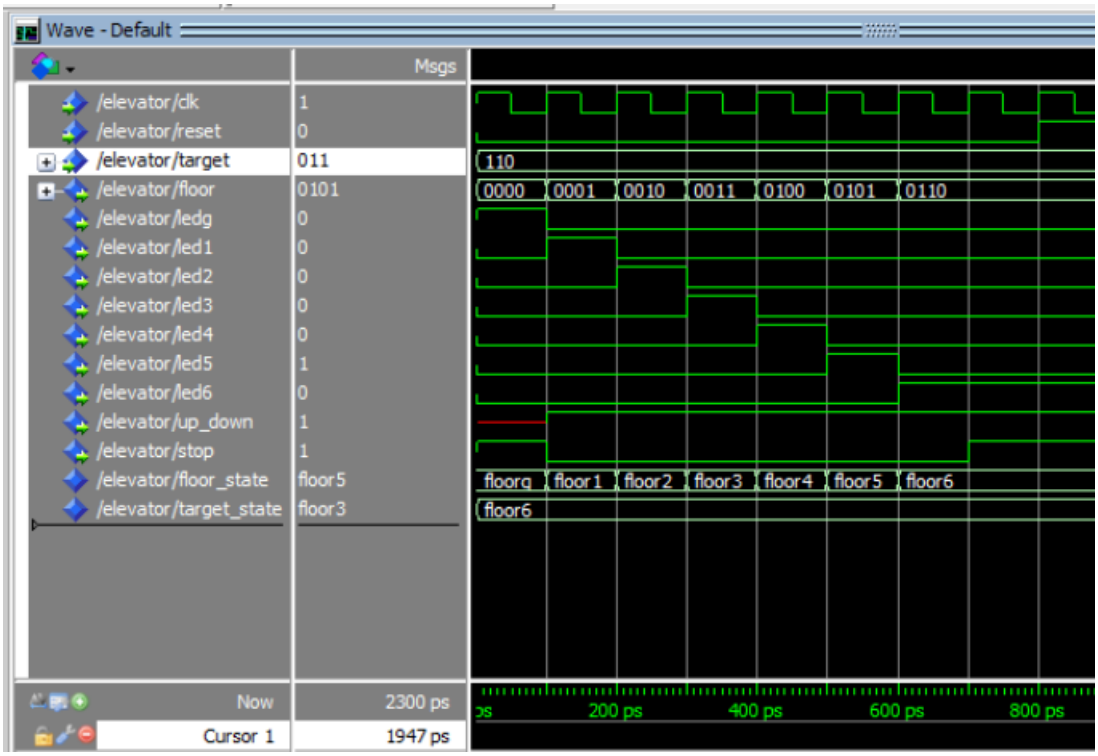

Figure 14 Modelsim Descending Scenario

A correlation can be observed between the results observed in JFLAP and the results from modelsim.

6. Regular Expression for the system

Q0 = x*I + 0*q0+ 0*q1d + 0*q2d+ 0*q3d + 0*q4d + 0*q5d + 0*q6

Q1a = 1*q0 + 1*I + (0+1)q1a

Q2a = 2*q0 + 2q1a + (0+1+2)q2a

Q3a = 3q0+3q1a+3q2a+(0+1+2+3)q3a

Q4a = 4q0+4I+4q3q+4q4a+(0+1+2+3+4)q4a

Q5a = 5q0+5q1a+5q2a+5q3a+5q4a+5q5a+(0+1+2+3+4+5)q5a

Q6 = 6q0+6q1a+6q2a+6q3a+6q4a+6q5a+6q6 + (0+1+2+3+4+6)q6

Q5d = 5q6 + x5a + (5+6)q5d

Q4d = xq4a+(4+5+6)q4d

Q3d = xq3a+(3+4+5+6)q3d

Q2d = xq2a+(2+3+4+5+6)q2d

Q1d = xq1d + (1+2+3+4+5+6)q1d

I = xq0

Then,

RE = E + q0 u q1a u q2a u q3a u q4a u q5a u q6 u q5d u q4d u q3d u q2d u q1d u I

7. Conclusion

The steps that need to be taken when implementing a DFA in real life were observed. Even though an algorithm can be explained in pseudocode easily, it becomes difficult when implementing a without a proper structure. State machines help to provide a stream lined method of defining the flow of data in an algorithm. Since the controlling schemes utilized in modern elevators are very complex, **certain assumptions were made in order to reduce the number of states and transitions**. Implementing the DFA on VHDL was very straightforward, and the results obtained from the DFA and VHDL code were consistent, which indicated that the conversion to VHDL was a success. Ultimately a contrast was drawn between VHDL and a sequential logic code running on a processor by implementing the same lift controller using python. (The code and results are provided in the appendix) This was achieved with a few lines of code and recursion. Finally it can be said that even though there may be lot of steps involved, developing a DFA for a real life application creates a proper structure that can be followed when implementing it using code.

8. References

[1] S. Dewasurendra, S. Ekanayake, R. Ekanayake, S. Sanjayan and S. G. Abeyratne, “FPGA Based Elevator Controller with Improved Reliability,” 2013.

[2] J. E. Savage, “Models of Computation,” in *Exploring the Power of Computing*, Addison-Wesley (1998), p. 672.

[3] “XILINX Triple Module Redundancy Design Techniques for Virtex FPGAs,” Application Note (Virtex Series).

[4] Z. Yuhang and M. Muyan, Implementation of six-layer automatic elevator controller based on FPGA , 2010 IEEE, 2010.

9. Appendices

VHDL Code

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

entity liftsim is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           target : in  STD_LOGIC_Vector (2 downto 0);
           floor : out  STD_LOGIC_VECTOR (3 downto 0);
              ledg,led1,led2,led3,led4,led5,led6 : out STD_LOGIC;
              up_down: out STD_LOGIC; --Indicates the direction of the elevator
              stop : out STD_LOGIC); --Says if the elevator is stopped
end liftsim;


architecture Behavioral of liftsim is

type floor_state_type is (floorg, floor1, floor2, floor3, floor4, floor5,
floor6);

signal floor_state, target_state : floor_state_type := floorg;


begin

target_state <=     floorg when (target = "000") else
                        floor1 when (target = "001") else
                        floor2 when (target = "010") else
                        floor3 when (target = "011") else
                        floor4 when (target = "100") else
                        floor5 when (target = "101") else
                        floor6 when (target = "110") else
                        floorg;


floor_state_machine: process(clk)
begin

    if clk'event and clk='1' then


        if reset='1' then
```

```
            floor_state <= floorg;

        else
            if (floor_state < target_state) then
                stop <= '0';
                up_down <= '1';
            elsif(floor_state > target_state)then
                stop <= '0';
                up_down <= '0';
            else
                stop <= '1';
            end if;

            case floor_state is

                when floorg =>

                    if (floor_state < target_state) then

                        floor_state <= floor1;

                    else
                        floor_state <= floorg;
                    end if;

                when floor1 =>

                    if (floor_state < target_state) then
                        floor_state <= floor2;

                    elsif (floor_state > target_state) then
                        floor_state <= floorg;

                    else
                        floor_state <= floor1;
                    end if;

                when floor2 =>

                    if (floor_state > target_state) then

                        floor_state <= floor1;

                    elsif (floor_state < target_state) then
```

```
                floor_state <= floor3;

            else
                floor_state <= floor2;
            end if;


        when floor3 =>

            if (floor_state > target_state) then

                floor_state <= floor2;

            elsif (floor_state < target_state) then

                floor_state <= floor4;

            else
                floor_state <= floor3;
            end if;

        when floor4 =>

            if (floor_state > target_state) then

                floor_state <= floor3;

            elsif (floor_state < target_state) then

                floor_state <= floor5;

            else
                floor_state <= floor4;
            end if;

        when floor5 =>

            if (floor_state > target_state) then

                floor_state <= floor4;

            elsif (floor_state < target_state) then

                floor_state <= floor6;
```

```
                    else
                        floor_state <= floor5;
                    end if;


                when floor6 =>

                    if (floor_state > target_state) then

                        floor_state <= floor5;

                    else
                        floor_state <= floor6;
                    end if;

                when others =>
                    floor_state <= floorg;
            end case;
        end if;
    end if;
end process;




floor <= "0000" when (floor_state = floorg) else --0
            "0001" when (floor_state = floor1) else --1
            "0010" when (floor_state = floor2) else --2
            "0011" when (floor_state = floor3) else --3
            "0100" when (floor_state = floor4) else --4
            "0101" when (floor_state = floor5) else --5
            "0110" when (floor_state = floor6) else --6
            "0000"; --otherwise reset output to floor 0

with floor_state select
        ledg <= '1' when floorg,
            '0' when others;

with floor_state select
        led1 <= '1' when floor1,
            '0' when others;

with floor_state select
        led2 <= '1' when floor2,
            '0' when others;
```

```
with floor_state select
        led3 <= '1' when floor3,
            '0' when others;

with floor_state select
        led4 <= '1' when floor4,
            '0' when others;

with floor_state select
        led5 <= '1' when floor5,
            '0' when others;

with floor_state select
        led6 <= '1' when floor6,
            '0' when others;


end Behavioral;
```

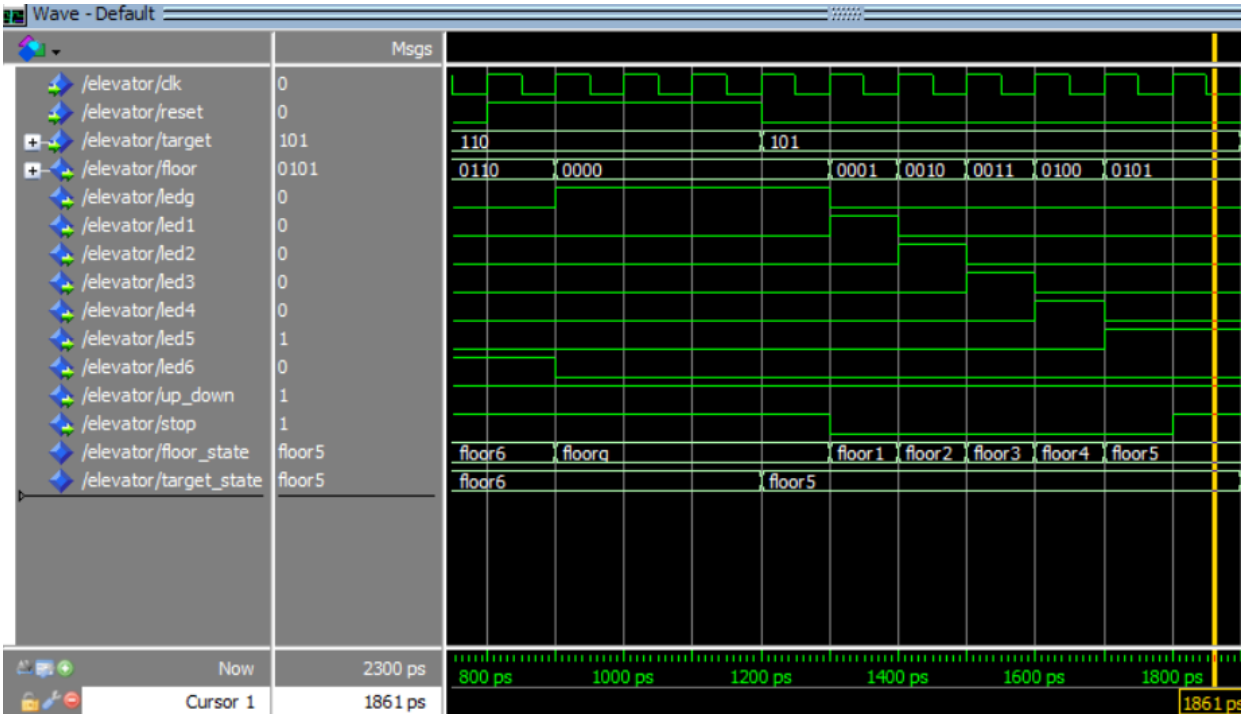

Figure 15 More simulation results

Python Code for a similar Scenario

```
import time

currentfloor = 0
floors = [0,0,0,0,0,0]
reqfloor = 0
highestreq = -1
lowestreq = 10

def movestate() :
    global highestreq
    global floors
    global reqfloor
    global highestreq
    global lowestreq
    global currentfloor
    while(1):
        inputstate()
        if(highestreq != -1):
            for x in range(currentfloor, highestreq+1):
                time.sleep(1)
                if(floors[x] == 1):
                    print "The Lift stops at Floor %d"%(x)
                    floors[x] = 0
                    currentfloor = x
                    movestate()
                else:
                    print "The Lift skips Floor %d"%(x)
                    currentfloor = x

        if(lowestreq != -1):
            for n in range(currentfloor, lowestreq-1,-1):
                time.sleep(1)
                if(floors[n] == 1):
                    print "The Lift stops at Floor %d"%(n)
                    floors[n] = 0
                    currentfloor = n
                    movestate()


                else:
                    print "The Lift skips Floor %d"%(n)
                    currentfloor = n
        highestreq = -1
```

```
        lowestreq = 10



def inputstate() :
    global highestreq
    global lowestreq
    print "The Lift is in Floor %d"%(currentfloor)
    while(1):
        reqfloor = raw_input('Enter Floors To Go To, C to Close Doors: ')
        if(reqfloor == 'C'):
            break
        elif(int(reqfloor) < 6 and int(reqfloor) >-1):
                floors[int(reqfloor)] = 1
                if(int(reqfloor)>currentfloor and highestreq <= int(reqfloor)):
                    highestreq = int(reqfloor)
                if(int(reqfloor)<currentfloor and lowestreq >= int(reqfloor)):
                    lowestreq = int(reqfloor)


movestate()
```

Output

```
C:\Users\Samira\Desktop>python lift.py
The Lift is in Floor 0
Enter Floors To Go To, C to Close Doors: 2
Enter Floors To Go To, C to Close Doors: 3
Enter Floors To Go To, C to Close Doors: C
The Lift skips Floor 0
The Lift skips Floor 1
The Lift stops at Floor 2
The Lift is in Floor 2
Enter Floors To Go To, C to Close Doors: 1
Enter Floors To Go To, C to Close Doors: 5
Enter Floors To Go To, C to Close Doors: C
The Lift skips Floor 2
The Lift stops at Floor 3
The Lift is in Floor 3
Enter Floors To Go To, C to Close Doors: C
The Lift skips Floor 3
The Lift skips Floor 4
The Lift stops at Floor 5
The Lift is in Floor 5
Enter Floors To Go To, C to Close Doors: C
The Lift skips Floor 5
The Lift skips Floor 5
The Lift skips Floor 4
The Lift skips Floor 3
The Lift skips Floor 2
The Lift stops at Floor 1
The Lift is in Floor 1
Enter Floors To Go To, C to Close Doors:
```