UNIVERSITY of DAYTON

ECE 501: CONTEMPORARY DIGITAL SYSTEMS

FINAL PROJECT

IR REMOTE CONTROL RECEIVER DESIGN & DEMO

4TH DECEMBER 2018

SUBMITTED BY

Venkataramani Kumar

Yassine Jaoudi

INTRODUCTION:

The main objective of this report is to demonstrate the working of IR Remote control on the DE2 115 board using VHDL code. In nutshell, this process works on the principle of reception of the IR signal from the remote, based on the signal it receives, the signal is decoded, and the value is displayed in the 7-segment display. As a continuation to this, the next stage deals with the command discrimination and execution using LEDs. For achieving this goal two different approaches were adapted: the first approach is based on the state machines given in the final project slides while the second approach was based on the concept given in the DE2-115 manual page 99. Both these approaches will succeed only if the counters, state machines and timing requirements for each these are properly managed and understood. More details and comments pertaining to both these approaches are provided in the design section.

The first step in this process is to understand the basic understanding of the process (as shown in Fig.1), followed by development of the VHDL code. Once, the code is developed it is tested for its complete operation using the testbench given in the ModelSim. The ModelSim gives an idea about the functioning of this IR Remote control in the form of waveforms after simulation. After verifying it based on the conceptual understanding, this code was targeted to the hardware DE2-115 board through the Quartus software. The process of targeting the hardware using Quartus begins with the compiling the code before and after the pins were allotted and finally the hardware gets programmed with this code. Now by using the remote control, different values can be changed. For instance, if the push button ‘1’ is pressed, the address, logical inverse of address, command and the logical inverse of command appears on the 7-segment display. In this way, the first part of the process gets completed. By either modifying the same code or by running a sub routine for this code, the second portion of the project that deals with the command execution using LEDs can be achieved. Once again this can be verified by using the operations mentioned in the assignment sheet. Also, pins must be assigned in such a way that all the LED’s may be considered for outputs and the remote-control push buttons serves as the input to this design.

This report along with the design and result section contains the error section where different kinds of errors encountered along its remedy is shown for the prevention of such errors in future. Though two approaches were used, the code first approach alone has been posted in Appendix-A owing to space constraints. This report also contains few points that must be kept in mind for the successful implementation of this project work. In the conclusion section, few suggestions and experience about doing this project too has been mentioned.

DESIGN

The design process of the IR Remote control begins with the development of the VHDL code for the design for two tasks. The first task deals with the demonstration of correct transmitted IR device address while the second task deals with command discrimination and execution which will make use of the LEDs.

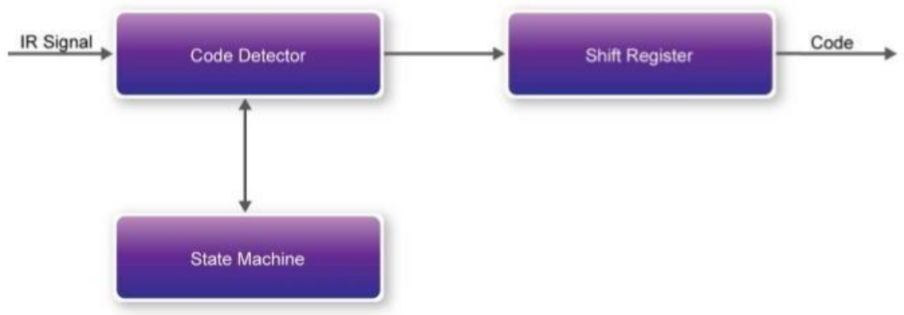


Fig.1: IR Receiver Controller

(Source: DE2-115 manual Page number 100)

The Fig.1 shows the basic block diagram of the IR Receiver controller that will be designed in this process. This process works on the principle of demodulating the signal to the code decoder block this in turn will check for the lead code and pass it on to the state machine. So, there are three different blocks that needs to be designed: Code detector, Shift Register and State Machine. Each of these blocks are individually designed and tested, then a logic connecting all the three is written to get a complete VHDL code. As mentioned in the introduction, two different approaches were utilized, and both these approaches are shown in this report. Either of these can be made use of for the complete working of the design.

This design section is split into two parts namely: Task-1 and Task-2 to highlight the methodologies adapted in realizing them in a succinct manner.

<u>Task-1: Demonstration of IR Device Address</u>

This task deals with the displaying of the IR device address on the 7-segment display when a push button from the remote control is pushed. Out of the eight 7 segment displays, as per the code developed, HEX4-HEX7 will display the custom code or the device address which remains the same irrespective of the push buttons pushed or pressed from the remote and HEX0-HEX3 will display the command or the key code. The key code changes when as per the push buttons in the remote control while the custom code (address and its logical inverse) appears in HEX4-HEX7. If ‘1’ is pushed, the key code of x’01 appears while ‘2’ is pushed the key code of x’02 will appear. The custom code or the device address or address varies depending on the IR remote being used. For instance, the custom code of the IR remote used in ECE 501 is x’68 x’66 (address and inverse logical addresses).

<u>Approach-1: Based on the codes and details given in the Final Project Slides:</u>

As mentioned before, the task-1 could be achieved using two approaches. This approach is based on the state machines and a portion of example code is given in the final project slides. The first step in using this approach is to understand the basic format of NEC IR transmission format (Fig.2) and state machines (Fig.3) very clearly along with their respective counters. There are different counters used here: LC_On_counter, LC_off_counter, clock counter and data_counter. Following this, is to design the various states as per the state machine and the pulse detection circuit. This the crux of this approach. The forthcoming paragraphs will explain these state machines and the NEC frame format precisely shown in Fig. 2. This basic design needs to be understood carefully esp. the timings of the leader codes and various counters with its logical operation. So, to make the process and the concept easy, a flow diagram of the entire process that was referred from the internet.

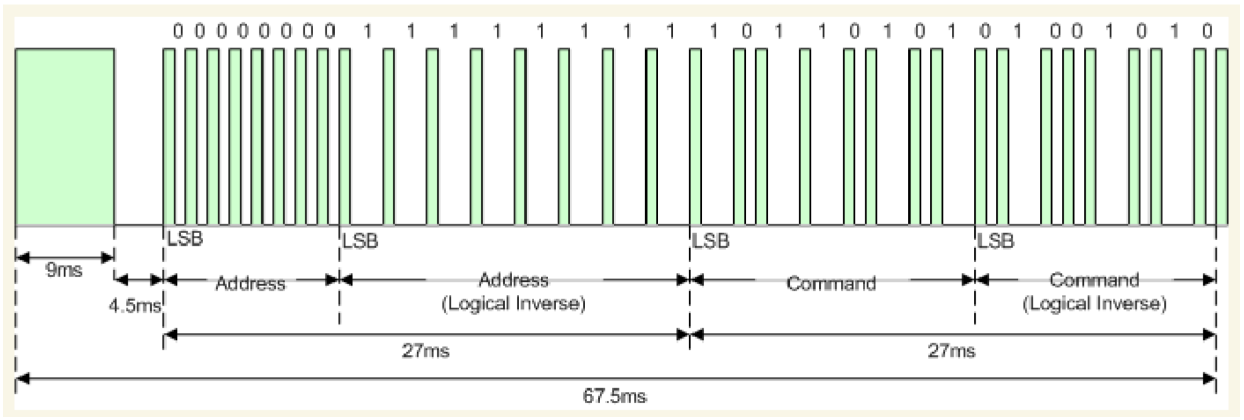


Fig.2: Format of NEC IR transmission frame

(Source: Final Project Notes)

Fig.2 shows the basic format of the NEC IR transmission format. The frame begins with a Leader code, that stays for about 9ms in ON state followed by 4.5ms of OFF state. This followed by 8 bits of remote-control address then its logical inverse which is succeeded by 8 bits of command and its corresponding logical inverse. A stop bit at the end marks the end of transmission. In this way the infrared signal from the remote control gets decoded.

The IR signal from the remote control can be seen in the 7-segment display due to the action of the state machine. The state machine is defined in such a way that the code and the address corresponding to the push button being pushed appears correctly on the display. This helps in better understanding of the steps that happens when a button in TV remote is pushed. State machines can be understood when it is represented in the diagrammatic manner as shown in Fig.3. This figure explains how the state machine is moved from a state to another depending on the conditions laid for its promotion or de-promotion from state to another.

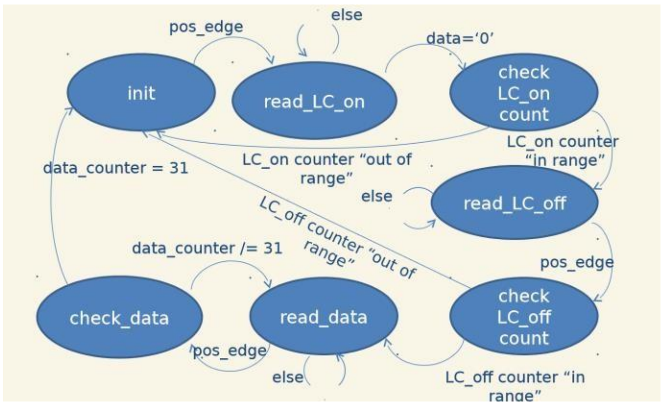


Fig.3: State machine for the IR Remote control

(Source: Final Project Slides)

Fig.3 explains the various states involved: initial stage that represents the beginning of the state machine, read_LC_on counter which reads the incoming IR signal at the pose_edge of the clock counter, LC_on counter that checks if all the bits from the IR signal is read completely or not, if so, then it succeeds to the read_LC_off state where the reading stops and the corresponding LC_off counter is checked. If this counter is out of range the state machine resets itself and reaches the initial state. Once the LC_off_counter meets the requirements, it moves to the next state which is the read_data where the reading of the data bits happen followed by check_data.

Fig.4 shows the logical operations happening in the state machine, the locations where the bits will be stored, and the kind of logical operations carried out at each step. This gives an option to verify the working manually although it is not advisable as it extremely confusing and an onerous task.

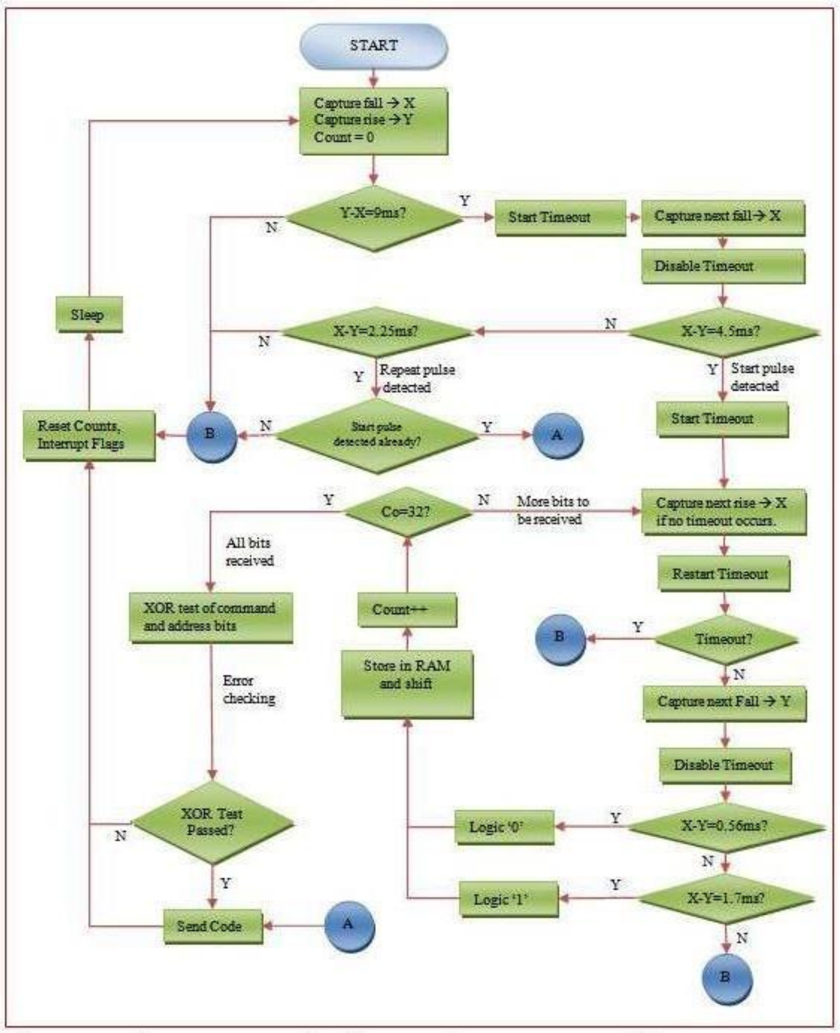


Fig.4: Flow chart of the NEC IR Protocol

(Source: Wikipedia)

NOTE:

1) The cyclic movement of the state machine from one state to another not only depends on the reading of the data and the leader code but also it depends on the counters. During all these processes, promotion to the next state is possible only if the corresponding counters i.e.., LC_on_counter and LC_off_counters are within the specified range. If they are out of range, then the state machine remains in the same state.
2) To generalize, along the functions performed by each state, the corresponding condition for its movement from one state to another needs to be satisfied. If they don't then the state machine will remain in the same state until the conditions are well satisfied.

These few concepts or requirements and ideas mentioned before was used in the further development of the code. The code was developed in a step-by-step sequence: initially the state machines were developed as per flow shown in the Fig.3, then the pulse detection circuit to detect the IR signals were developed followed by the development of the counters as per the requirements. 2% tolerance was given to both LC_on_counter and LC_off_counter as perfect working of counters at the prescribed requirements is not possible.

While developing the code, most of the time was spent on the development of the various counters. For instance, once the value of LC_on_counter exceeds LC_on_max + trans_max, the next state needs to be activated. Similarly, was the case with the other counters. The development of counters was very hard because of the buffer time involved in it. There is a 2% buffer (or allowance) that is given. In most of the cases, the proper roll-over was not happening. Even if it was happening, the counters such as data_counter and clock counters were not counting properly. Therefore, most of the time was allocated for the counters, and each counter were first developed one by one and tested. For instance, LC_on_counter and LC_off_counter were the first counters that were developed, once it was developed, its functionality was tested with the help of the testbench. In this way a slow and steady method was undertaken to ensure that the entire process works perfectly as per the requirements. Another advantage of this method is that, once the counters are perfectly developed and tested successfully, if any error occurs in future while compiling different process together, trouble-shooting or debugging the errors will be very simple. This because, all the counters and process are individually working fine, now the rectification needs to be done in the logic or the way in which they are implemented. In this way, proper and easy error tracking can be achieved.

The next important component to be designed is the shift register 'shift_reg'. The entire display depends on the bits in this shift register. The entire seven segment display depends on the bits in this shift register. It is being implemented in a way that it will be collecting and shifting data in, once the +2% and -2% tolerance satisfies the requirement of detecting the rising_edge of the data and depending on the LC_off_counter, it can be detected if pulse width represent a 0 in case of 1.125 ms or a 1 in case the value of the counter is 2.25 ms. Right after this bit detection, it is being shifted into the 32-bit shift_reg and then when the data_counter is equal to 31, the data inside shift register is ready to be mapped into the seven-segment display.

In this way, the entire VHDL code for this design was developed and tested on the DE2-115 board using the Quartus software. The eight 7-segments displays the custom code and the command respectively.

This is the first approach that was adapted to implement the IR remote control. The main drawback using this approach was the immense time spent on the counters as mentioned. Till the last week of the deadline, the counters were an impediment in the achieving the final goal. But parallelly approach-2 was also drafted and developed based on the information provided in the DE2-115 manual. In this manual, the pages preceding and succeeding 99 provided copious amounts of information pertaining to this work which is highlighted as the second approach. Keeping in view of the time constraints, the second approach was developed simultaneously if in case approach one fails, the second could be used for demonstration. Fortunately, both these approaches are working, and they are shown in precise manner for further references.

NOTE:

1) Custom Code- Custom code was once such term which is commonly used to refer the Address of the Remote control or device address which can be different for different remote control. For instance, the custom code for the NEC IR Remote control (that is used in this project work) is x68 x66 while for some other remote control it is x68 x50 in this way it is different for different remote control.
2) Also, one needs to remember this fact that the custom code always remains the same irrespective of the push buttons being pressed. Therefore, one must not get confused or baffled if the custom code remains the same despite different push buttons being used.
3) If the output is 0x0000 then it is called as repeat command and it happens since the user has kept the key always pressed all the time.
4) This project can be done in multiple ways as mentioned before: either all the processes can be called in a single file or through different separate VHDL files like the UART done before.
5) Care must be taken while mapping PIN_Y15 as it is the important pin which is needed for the IR Remote control to function else despite doing several processes or routines the output cannot be observed.
6) Better results were obtained when the tolerance of $\underline{+}$ 4% was used instead of 2%.
7) The timings for 1s and 0s if changed from 1.67ms and 0.56ms respectively to 1.56ms and 0.52ms brought better results.

Approach-2 Based on the state machine from Page 99 of the DE2-115 manual

For implementing this approach, the state machine as shown in Fig.5 was used. This approach from the DE2-115 manual is very simple because of limited number of states. When the number of states in the state machine is limited the complexity too decreases making it very easy to construe and realize it. In this approach, there are only 3 states namely: Idle, Dataread and Guidance.

The timing requirements are like the ones mentioned in the final project files, the only difference is the reduced number of states except that these two approaches do not have big difference. In this approach, the IR receiver works as follows: when the Lead code is detected the state, machine changes the state from IDLE to GUIDANCE and once the custom code or the device address or the address is detected by the code detector the present state changes to the next state that is the DATAREAD.

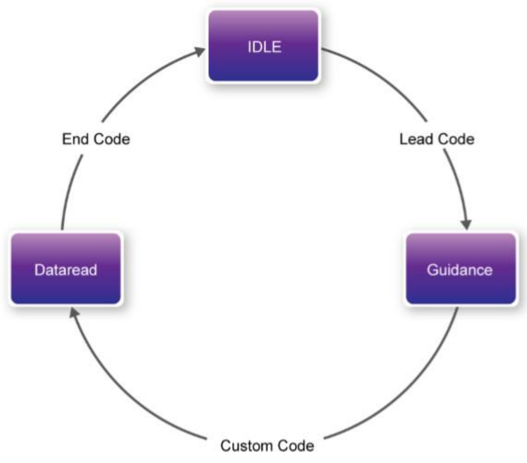


Fig.5: State machine of approach-2
(Source: DE2 115 manual)

In this way, this approach can be adapted. As mentioned before except for the reduced number of states there isn't any special advantage because when the number of states is reduced, the counter requirements is very low, and the coding process becomes very easy. This approach became very simple after struggling a lot with the first approach.

<u>Task-2 Command discrimination and execution</u>

This task becomes very simple after the successful implementation of the task-1. The most challenging portion was to design the Volume Up and Down as it involves developing the repeat codes for its functioning. This task involves the usage of LEDs to test the working of the buttons on the remote control. In other words, when any buttons on the IR remote is pushed the corresponding LED must ON or OFF. For instance, as per the specifications, when channel 2 is pushed the channel 7 LED must glow off and vice versa. In this way, different tasks are designed, and they can be carried out using the 'if-else' condition.

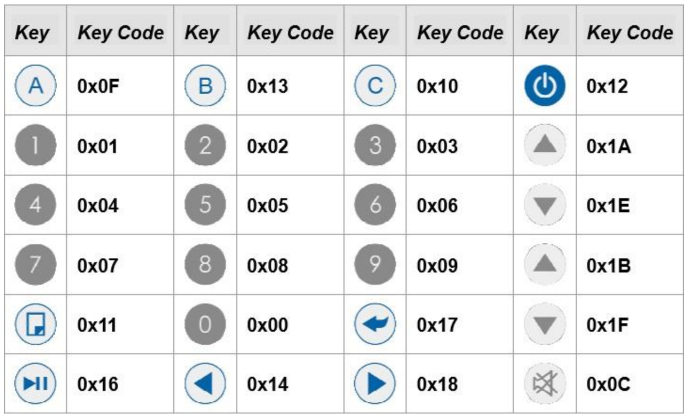
To be very clear, this task was defined based on the command. If x'12 is the command that is detected, then LEDG0 must ON when the power button is pushed and LEDR0 must be OFF when the power button is pushed again. In this way all the other tasks can be carried except for the Volume Up and Down, as they must remain ON as long as the button is pushed and must turn OFF on being released.

FAILURE ENCOUNTERED:

1) The Volume Up and Down was working as simple ON or OFF instead of being ON as long the push button is pushed and OFF on being released despite several trials. To make it work a new state called repeat state was formulated, and if the lead code is 1 then it will move to this state. This is how this new state was formulated but it did not work. Therefore, Volume Up and Down was implemented like any other push buttons such as power where in when the Up is pressed, LED10 glows ON otherwise LED11 glows ON on pushing the Down button.

NOTE:

1) The task-2 too was implemented using both the approaches using the same concept. In this way both the approaches were made use of for this project work.
2) The page 99 of the DE2-115 manual gives the key codes of various push buttons. This is extremely useful while developing and testing the code. Once the board is programmed with the code then using this (as shown in Fig.6) functionality can be tested properly.



Key	Key Code	Key	Key Code	Key	Key Code	Key	Key Code
A	0x0F	B	0x13	C	0x10	⏻	0x12
1	0x01	2	0x02	3	0x03	▲	0x1A
4	0x04	5	0x05	6	0x06	▼	0x1E
7	0x07	8	0x08	9	0x09	▲	0x1B
▣	0x11	0	0x00	↩	0x17	▼	0x1F
⏯	0x16	◀	0x14	▶	0x18	🔇	0x0C

Fig.6: Key code information for each key in the Remote Control
(Source: DE2-115 Manual Page number:99)

3) Also, the same manual has good amount of information in the pages preceding and succeeding page #99 pertaining to the detailed working of the IR Remote controller that can be made use of while working on this project.
4) There are two different ways of implementing the second task: (i) Installing another state in the state machine (ii) Based on the address of the key pressed. In this work, the second method was adapted, based on the command or the keycode of the key pressed, the LEDs were designed to be in ON or OFF state.

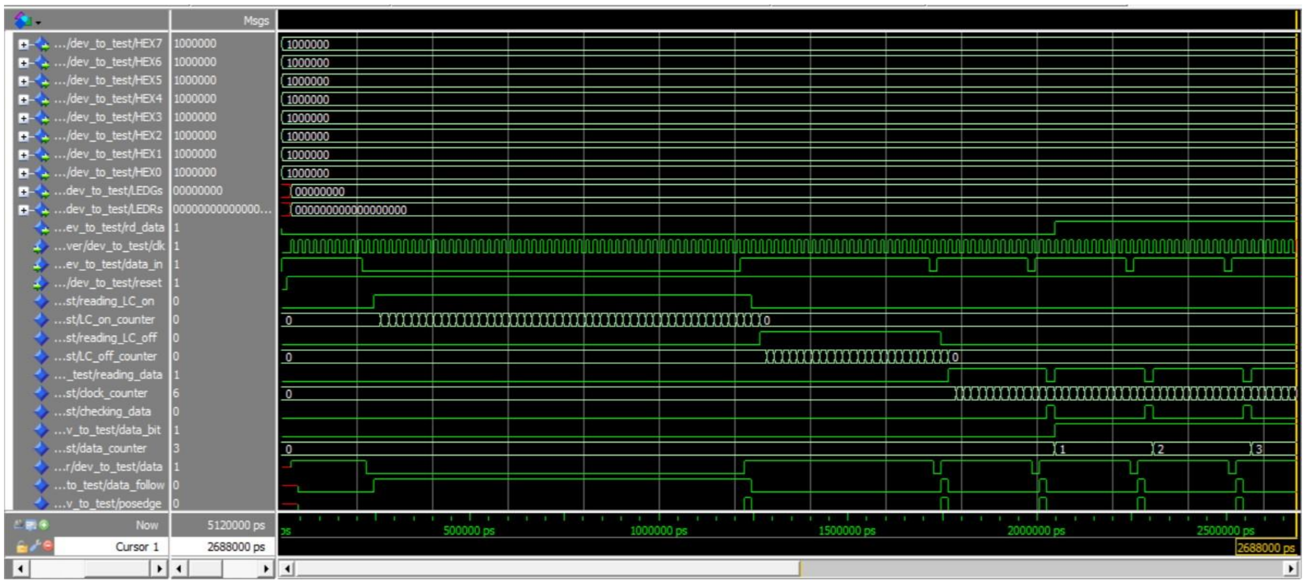
RESULTS

The design section in detail highlighted the different approaches and the tasks related to the IR Remote controller. This section will deal with the results that were obtained after passing the code through the testbench and in Quartus after compilation.

Here the state machine is developed properly with the counters meeting the appropriate requirements mentioned in the design section. Once, those standards are met, automatically code passes the testbench properly. But as usual there are several points which needs to be kept in mind while working on this project which is also mentioned in the forthcoming paragraphs.

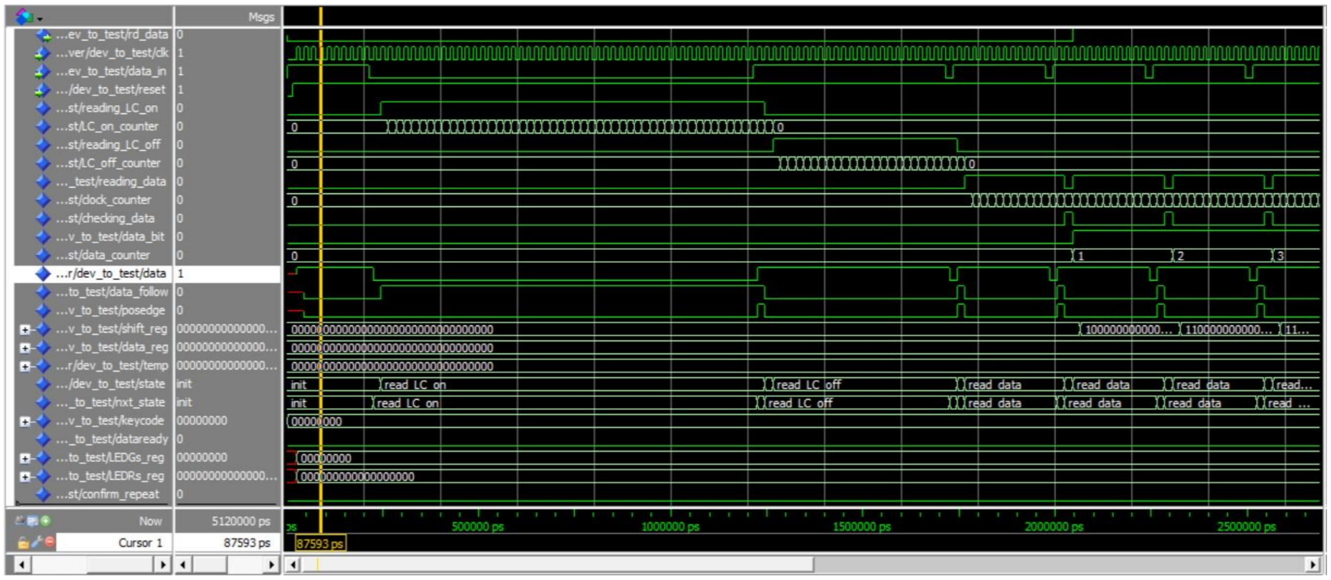
Due to the space constraints, the ModelSim simulation results obtained from approach-1 only is shown here and in a similar manner the simulation waveforms for approach-2 too can be obtained.

The explanation corresponding to the simulation waveforms using approach-1 is done here in the forthcoming para.



	Msgs
.../dev_to_test/HEX7	1000000
.../dev_to_test/HEX6	1000000
.../dev_to_test/HEX5	1000000
.../dev_to_test/HEX4	1000000
.../dev_to_test/HEX3	1000000
.../dev_to_test/HEX2	1000000
.../dev_to_test/HEX1	1000000
.../dev_to_test/HEX0	1000000
...dev_to_test/LEDGs	00000000
...dev_to_test/LEDRs	0000000000000...
...ev_to_test/rd_data	1
...ver/dev_to_test/clk	1
...ev_to_test/data_in	1
.../dev_to_test/reset	1
...st/reading_LC_on	0
...st/LC_on_counter	0
...st/reading_LC_off	0
...st/LC_off_counter	0
..._test/reading_data	1
...st/clock_counter	6
...st/checking_data	0
...v_to_test/data_bit	1
...st/data_counter	3
...r/dev_to_test/data	1
...to_test/data_follow	0
...v_to_test/posedge	0
Now	5120000 ps
Cursor 1	2688000 ps

Fig.7a: ModelSim Simulation waveform

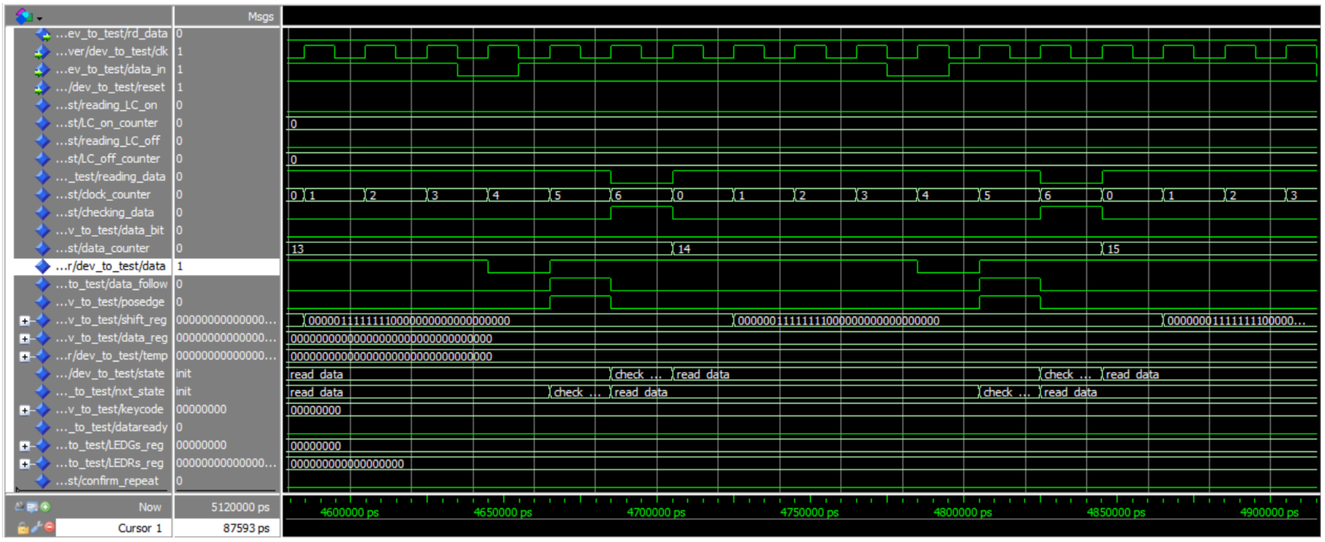


	Msgs
...ev_to_test/rd_data	0
...ver/dev_to_test/clk	1
...ev_to_test/data_in	1
.../dev_to_test/reset	1
...st/reading_LC_on	0
...st/LC_on_counter	0
...st/reading_LC_off	0
...st/LC_off_counter	0
..._test/reading_data	0
...st/clock_counter	0
...st/checking_data	0
...v_to_test/data_bit	0
...st/data_counter	0
...r/dev_to_test/data	1
...to_test/data_follow	0
...v_to_test/posedge	0
...v_to_test/shift_reg	0000000000000...
...v_to_test/data_reg	0000000000000...
...r/dev_to_test/temp	0000000000000...
.../dev_to_test/state	init
..._to_test/nxt_state	init
...v_to_test/keycode	00000000
..._to_test/dataready	0
...to_test/LEDGs_reg	00000000
...to_test/LEDRs_reg	0000000000000...
...st/confirm_repeat	0
Now	5120000 ps
Cursor 1	87593 ps

Fig.7b: ModelSim simulation waveform

Fig.7a,7b and 7c shows the simulation waveform that was obtained after testing the developed code with the testbench. Since the VHDL code is huge with several states and each state must be tested properly, the simulation waveform has been appropriately truncated and posted here for clear view. From these figures one can observe that all the counters are counting, the shift registers are shifting the values periodically as how it is supposed to do, the different states are visible very clearly vindicating that the state machine is working very fine.

When the simulation waveforms (Fig.7a through 7c) are observed, the state machines move from one state to another depending on the conditions satisfied. For instance, at the positive edge or when the data_in is HIGH the state machine moves from initial state to the read_LC_on state and similar when the next condition is satisfied (as per Fig.3) it moves to the next state. In this way each state is implemented.



	Msgs
...ev_to_test/rd_data	0
...ver/dev_to_test/clk	1
...ev_to_test/data_in	1
.../dev_to_test/reset	1
...st/reading_LC_on	0
...st/LC_on_counter	0
...st/reading_LC_off	0
...st/LC_off_counter	0
..._test/reading_data	0
...st/clock_counter	0
...st/checking_data	0
...v_to_test/data_bit	0
...st/data_counter	0
...r/dev_to_test/data	1
...to_test/data_follow	0
...v_to_test/posedge	0
...v_to_test/shift_reg	00000000000000...
...v_to_test/data_reg	00000000000000...
...r/dev_to_test/temp	00000000000000...
.../dev_to_test/state	init
..._to_test/nxt_state	init
...v_to_test/keycode	00000000
..._to_test/dataready	0
...to_test/LEDGs_reg	00000000
...to_test/LEDRs_reg	00000000000000...
...st/confirm_repeat	0
Now	5120000 ps
Cursor 1	87593 ps

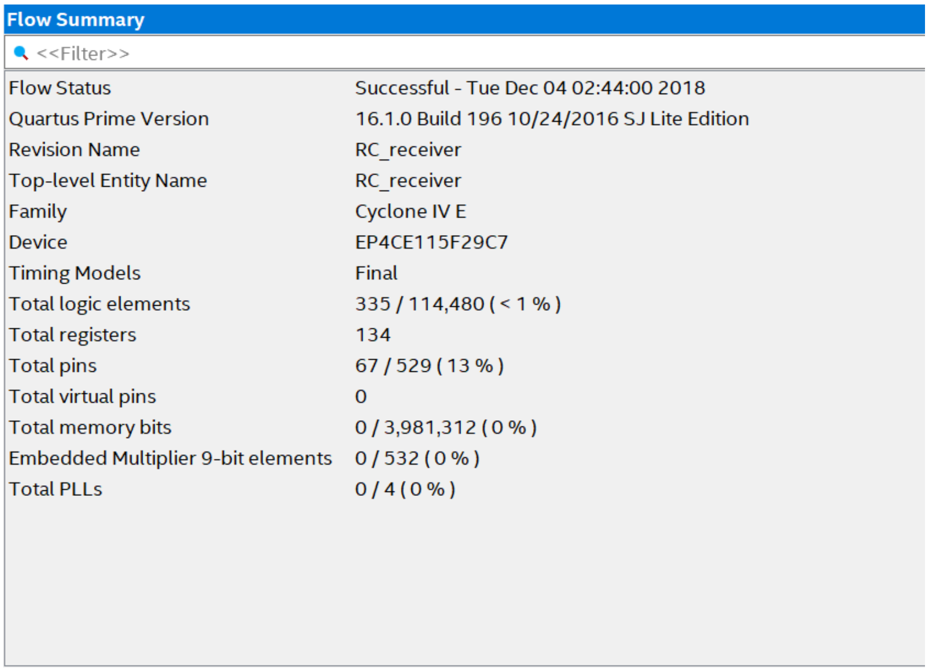
Fig.7c: ModelSim simulation waveform

The same can be observed in these waveforms too. The different states it traverses at each value of the clock and data counter too can be verified from this waveform. Next is the shift register, the 32-bit shift register is always shifting the values based on the data_counter. These values are the ones which are displayed in the 7-segment display. The first four segments are the custom codes (address and its logical inverse) and the rest four segments are the data (or the command) that is pushed and its logical inverse.

The working process is highly tricky due to the presence of different clocks present in the code. The clock counter is continuously counting the clock pulses, the LC_on_counter ensures that the time period of 9ms of ON state and LC_off_counter notices the time period of 4.5ms of OFF state is always maintained. Similarly, once this condition is satisfied, or if the reading data reaches '1' then the data counter starts counting. Depending on the last bit of data, the values of the shift register is decided. Then the 32 bits will be properly mapped to the 7-segment display.

In this way, the explanation provided in the design is diagrammatically shown in the waveform from ModelSim. This gives an assurance for the complete and perfect working of the code. Once, code is tested properly, it can be programmed on the kit using the Quartus software. This gives a detailed view of the various logical elements and pins are made use of by this design.

Fig.8 and Fig.9 shows the compilation results and a portion of RTL view of the design respectively as the design is extremely huge and it cannot be fitted in a single page. The RTL view gives the gate level design based on this developed code. This is very helpful as it gives an idea about the various gates being used and how they relate to each other. From Fig.8 we can see that this design utilizes **almost 335 out of 114480 (which <1%) logic elements, 67 out of 529 pins (which is about 13%).** Such a utilization is expected as it as several process associated, and LEDs are used to check for the proper working of the states during the development stage.



Flow Summary

<<Filter>>

Flow Status	Successful - Tue Dec 04 02:44:00 2018
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	RC_receiver
Top-level Entity Name	RC_receiver
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	335 / 114,480 (< 1 %)
Total registers	134
Total pins	67 / 529 (13 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Fig.8: Compilation results

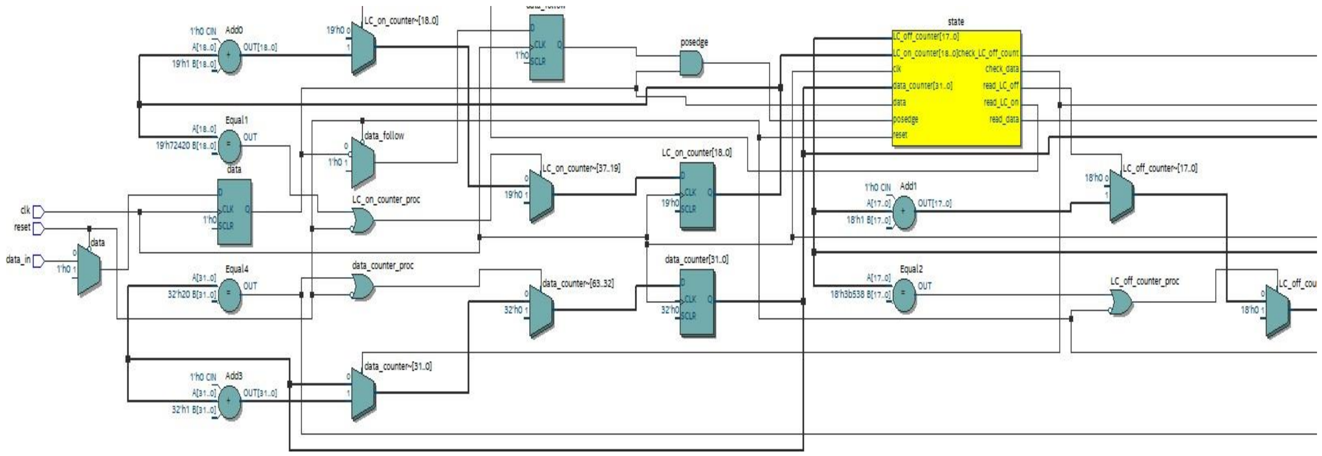


Fig.9: RTL View

Now, little explanation on the working of LEDs needs to be mentioned. Although there wasn’t any specific testbench made use of to verify this task-2, as it can be verified visually after targeting on the board. For controlling the LEDs, it is very important to be aware of the command of each button pressed from the remote control. Based on this the LEDs can be controlled: illuminated and extinguished alternatively. The key codes associated with each push button can be seen from Fig.6. Keeping this in mind, the second task was executed properly except for the Volume Up and Down (that was designed as simple ON and OFF) as mentioned in the failures section, though the concept was clear.

NOTE:

1) The LEDs can be turned ON or OFF using the ‘NOT’ logic statement and for toggling or alternatively switching between two LEDs ‘XOR’ gate can be made use of.

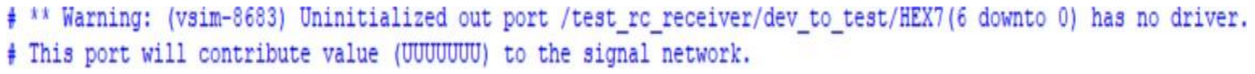
2) Port mapping of every element is very important esp. when there are several VHDL files containing different processes.
3) Failure to map properly results in different errors which are very hard to solve.

Therefore, the result section gave a holistic view of the working of the code in both theoretical and practical manner. It also provided different thinking involved while developing the code. Several errors were committed and rectified of which few important ones are provided in the next section to give an insight and prevent such things from happening again.

ERRORS AND CORRECTIONS

In this section, collection of most occurring errors has been shown. Other than this there were many others too, but these were very significant and needs to be addressed very quickly for better and clear functioning of the design.

1) **Uninitialized output port/output port has no driver:**



```
# ** Warning: (vsim-8683) Uninitialized out port /test_rc_receiver/dev_to_test/HEX7(6 downto 0) has no driver.
# This port will contribute value (UUUUUUU) to the signal network.
```

Fig.10: Uninitialized Output port error

As shown in Fig.10, this happens to be one of the most recurring errors encountered during the design process. This error can be alleviated by properly mapping the ports from the entity in the VHDL code to that of the component present in the testbench. In other words, by port-mapping the input and output ports this error can be rectified.

2) **Changing the outputs to the buffer mode**
Such kinds of errors happen during the compilation in the Quartus, this happens when a port is marked to be a signal instead of buffer, esp. while working with the 2nd task that deals with the control of the LEDs. By understanding the functions and the operations of the LEDs this sort of error can be rectified very easily.

3) **Entity does not exist**
This type of error occurs in the Quartus when the name given to the project file does not match with that of the entity. The same error happens in the ModelSim too as ‘missing design file’. Both these errors mean the same. Checking the names, the spelling and rectifying it can solve the issue else, the port mapping in the code and the testbench needs to be verified in case of the ModelSim and correcting the spelling will solve the issue in the Quartus.

4) **The output has 2 elements; expecting 9 elements**
This is another popular error that pops up while working with the LEDs. The LEDs must be mapped in accordance to the numbers. For instance, if there are 8 outputs, they can be mapped to 8 different LEDs. But if only two LEDs say, LEDG0 and LEDG1 are declared but 9 output bits are assigned to it, such errors happen.

5) **Other common errors**
There are few errors which generally happen if the outputs are assigned to the ports that are contrary to the way they are defined. For instance, if an input or output port is defined as ‘std_logic’, the values that can take is 0 or 1. Integers cannot be assigned to such ports. Care must be taken to ensure that these ports are well defined, and they are assigned properly.

CONCLUSION

In the all the previous mentioned sections, a detailed review of the steps involved in the design were mentioned as detailly as possible for better understanding and to highlight the entire thought process involved while designing this project. The report was drafted in such a way that there is a logical flow within each section and from one section to another to get a clear picture of how this project was handled. As mentioned in the design section, step-by-step method was adapted so that each step is covered properly, and only then can this project be successful. It is worth mentioning here that this project is extremely sequential in nature, i.e.., one process succeeds another. Only if the signal is decoded in the 7-segment display can the next task of controlling the LEDs happen. No bypassing of steps can be done. If the first task is unsuccessful so shall the entire project. Two approaches were used, and both the tasks were completed using both the approaches except for Volume Up and Down, wherein it was made to work like the power button. Sometimes toggling was observed wrt the Volume button but not always and this posed a big challenge as the concept was well understood but was not able to implement it.

Like the previous assignments, from this project work too, a massive amount of knowledge was learnt. Of the many concepts learnt, one such is the working of the IR Remote control. Besides using the remote control provided, different remotes were used and tested on this kit and device address was seen in the 7-segment display. For instance, a random remote control when used on this board displayed an address of x’68 x’50 while the remote provided with the kit has a device address of x’68 x’66. In this way, the working of the remote control was learnt properly and got amused with the various results that were obtained.

Also, the errors that were confronted while developing the code was immense. So many errors were found. The onerous responsibility of rectifying the errors were done and, in this way, new concepts and deeper concepts of VHDL, ModelSim and Quartus were learnt. By collectively incorporating the knowledge gained from the errors and studying related concepts, resulted in working of the DE2-115 board. It was exciting and happy to see the results after a spate of failures.

In toto, in this report, the IR Remote control receiver design was implemented, the outputs associated with two tasks too were observed. The details about the same is recorded here with full transparency to highlight the difficulty level of this project. Besides the technical knowledge gained, the spirit of working as a team and to lead as team was also learnt.

Appendix-A

The VHDL codes of the first approach alone is shown here for reference

<u>Approach-1 VHDL Code</u>

(i) The VHDL Code

```
-- RC_receiver
-- implement a data receiver for the DE2 remote control
--Team Members: Venkataramani and Yassine
--4th December 2018
--Approach-1
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity RC_receiver is
generic (
      -- number of clocks for the leader code_on signal (assume 50MHZ clock)
      LC_on_max                       : integer := 450000
      );
port(
      -- outputs to the 8 7-segment displays. The remote control
      -- outputs 32 bits of binary data (each byte display as
      -- 2 7-segment displays)
      HEX7                            : out std_logic_vector(6 downto 0);
      HEX6                            : out std_logic_vector(6 downto 0);
      HEX5                            : out std_logic_vector(6 downto 0);
      HEX4                            : out std_logic_vector(6 downto 0);
      HEX3                            : out std_logic_vector(6 downto 0);
      HEX2                            : out std_logic_vector(6 downto 0);
      HEX1                            : out std_logic_vector(6 downto 0);
      HEX0                            : out std_logic_vector(6 downto 0);

      LEDG                            : out std_logic_vector(7 downto 0);
      LEDR                            : out std_logic_vector(17 downto 0);
      -- output to display when receiver is receiving data
      rd_data                         : out std_logic;
      -- clock, data input, and system reset
      clk                             : in std_logic;
      data_in                         : in std_logic;
      reset                                 : in std_logic);
end RC_receiver;

architecture arc of RC_receiver is
-- leader code off duration
-- lengths of symbols '1' and '0'
-- length of transition time (error)
constant LC_off_max                   : integer := LC_on_max/2;
constant one_clocks                   : integer := LC_on_max/4;
constant zero_clocks                  : integer := LC_on_max/8;
constant trans_max                    : integer := LC_on_max/50;
constant LC_off_repeat_max            : integer := LC_off_max/2;

------------------------------------------------------------------------------------------------
constant max_bits                   : integer := 32;
```

```
-- counter for measuring the duration of the leader code-on signal
signal reading_LC_on             : std_logic := '0';
signal LC_on_counter             : integer range 0 to LC_on_max+trans_max;
-- counter for measuring the duration of the leader code-off signal
signal reading_LC_off            : std_logic := '0';
signal LC_off_counter            : integer range 0 to LC_off_max+trans_max;
-- counter for measuring the duration of the data signal
signal reading_data              : std_logic := '0';
signal clock_counter             : integer range 0 to one_clocks+trans_max;
signal checking_data             : std_logic := '0';
-- signal which determine the bit that is communicated
signal data_bit                  : std_logic := '0';

-- counter to keep track of the number of bits transmitted
signal data_counter              : integer := 0;
-- signals for edge detection circuitry
signal data                      : std_logic;
signal data_follow               : std_logic;
signal pos_edge                  : std_logic;
-- shift register which holds the transmitted bits
signal shift_reg: std_logic_vector(max_bits-1 downto 0) := (others => '0');
signal data_reg : std_logic_vector(max_bits-1 downto 0) := (others => '0');
signal temp : std_logic_vector(max_bits-1 downto 0) := (others => '0');

-- state machine signals
type state_type is (init, read_LC_on, check_LC_on_count, read_LC_off,
            check_LC_off_count, read_data, check_data);
signal state, nxt_state       : state_type;

-- The code is divided into different process: (i) LED process (ii) 7 Segment
display (iii)State machine process
-- LED signals
signal command                : std_logic_vector(7 downto 0);
signal dt_rdy                 : std_logic := '0'; -- Data ready
signal LG_reg                 : std_logic_vector(7 downto 0); -- Green LEDs
signal LR_reg                 : std_logic_vector(17 downto 0); -- Red LEDs
-- 7 segment display circuitry
component hex_to_7_seg  is
port (
      seven_seg         : out std_logic_vector(6 downto 0);
      hex               : in std_logic_vector(3 downto 0));
end component;

begin
      -- state machine processes
      -- Defining clock
      state_proc : process(clk)
      begin
            if(rising_edge(clk)) then
                  if(reset = '0') then
                        state <= init;   -- Based on the assignment sheet
                  else
                        state <= nxt_state;
                  end if;
            end if;
      end process state_proc;
```

```
        nxt_state_proc : process(state, pos_edge, data,
    LC_on_counter,LC_off_counter, clock_counter, data_counter)
        begin
              nxt_state <= state;-- Initialization of the various states
              reading_LC_on <= '0';
              reading_LC_off <= '0';
              reading_data <= '0';
              checking_data <= '0';

        --- The entire state machine was developed based
        --- on the details given in the assignment
        --- Nothing new was added
              case state is
                    when init =>
                          if(data = '0') then
                                nxt_state <= read_LC_on;
                          else
                                nxt_state <= init;
                          end if;
                    when read_LC_on =>
                          reading_LC_on <= '1';
                          if(pos_edge = '1') then
                                nxt_state <= check_LC_on_count;
                          else
                                nxt_state <= read_LC_on;
                          end if;
                    when check_LC_on_count =>
                       if ((LC_on_counter < LC_on_max+trans_max) and
              (LC_on_counter > LC_on_max-trans_max)) then
                                nxt_state <= read_LC_off;
                          else
                                nxt_state <= init;
                          end if;
                    when read_LC_off =>
                          reading_LC_off <= '1';
                          if(data = '0') then
                                nxt_state <= check_LC_off_count;
                          else
                                nxt_state <= read_LC_off;
                          end if;
                    when check_LC_off_count =>
                          if ((LC_off_counter < LC_off_max+trans_max) and
                          (LC_off_counter > LC_off_max-trans_max)) then
                                nxt_state <= read_data;
                          elsif ((LC_off_counter < LC_off_repeat_max+trans_max)
                          and (LC_off_counter > LC_off_repeat_max-trans_max))
                          then
                                nxt_state <= init;
                          else
                                nxt_state <= init;
                          end if;
                    when read_data =>
                          reading_data <= '1';
                          if(pos_edge = '1') then
                                nxt_state <= check_data;
                          else
                                nxt_state <= read_data;
```

```
                        end if;
                  when check_data =>
                        checking_data <= '1';
                        if(data_counter /= 31) then
                              nxt_state <= read_data;
                        else
                              nxt_state <= init;
                        end if;

                  when others =>
                        nxt_state <= init;
            end case;
      end process nxt_state_proc;


      -- Pulse detection circuitry
      pos_edge <= data and data_follow;
      pos_edge_proc : process(clk)
      begin
            if(rising_edge(clk)) then
                  if(reset = '0') then
                        data <= '0';
                        data_follow <= '0';
                  else
                        data <= data_in;
                        data_follow <= not data;
                  end if;
            end if;
      end process pos_edge_proc;



      LC_on_counter_proc : process(clk)
      begin
            if(rising_edge(clk)) then
                  if((reset = '0') or (LC_on_counter = LC_on_max+trans_max))
then
                        LC_on_counter <= 0;
                  elsif(reading_LC_on = '1') then
                        LC_on_counter <= LC_on_counter + 1;
                  else
                        LC_on_counter <= 0;
                  end if;
            end if;
      end process LC_on_counter_proc;

      -- LC_off counter
      -- Based on the state machine
      -- Either reset or in the buffer mode (2% tolerance)
      LC_off_counter_proc : process(clk)
      begin
            if(rising_edge(clk)) then
                  if((reset = '0') or(LC_off_counter = LC_off_max+trans_max))
                  then
                        LC_off_counter <= 0;
                  elsif(reading_LC_off = '1') then
                        LC_off_counter <= LC_off_counter + 1;
            else
                  LC_off_counter <= 0;
            end if;
      end if;
end process LC_off_counter_proc;

-- clock counter can be written as process :
cc_proc : process(clk)
begin
      if(rising_edge(clk)) then
            if((reset = '0') or (clock_counter = one_clocks+trans_max)
            or checking_data = '1')) then
                  clock_counter <= 0;
            elsif(reading_data = '1') then
                  clock_counter <= clock_counter + 1;
            else
                  clock_counter <= 0;
            end if;
      end if;
end process cc_proc;

--To find the nature of the bit that is transmitted
rd_data <= data_bit;
data_bit_proc : process(clk)
begin
      if(rising_edge(clk)) then
            if(reset = '0') then
                  data_bit <= '0';
            elsif(checking_data = '1') then
                        if((clock_counter < one_clocks+trans_max) and
                        (clock_counter > one_clocks-trans_max)) then
                              data_bit <= '1';
                        elsif((clock_counter < zero_clocks+trans_max)
                        and (clock_counter > zero_clocks-trans_max))
                        then
                              data_bit <= '0';
                        end if;
            end if;
      end if;
end process data_bit_proc;

-- This is the process for the data counter
dc_proc : process(clk)
begin
      if(rising_edge(clk)) then
            if(reset = '0' or data_counter = max_bits) then
                  data_counter <= 0;
            elsif(checking_data = '1') then
                  data_counter <= data_counter + 1;
            end if;
      end if;
end process dc_proc;

--This is the process for the working of shift register:
shift _proc : process(clk)
begin
      if(rising_edge(clk)) then
```

```
            if(reset = '0') then
                  shift_reg <= (others => '0');
            elsif(clock_counter = 0 and data_counter /= 31)
                  shift_reg <= data_bit & shift_reg(31 downto 1);
                  --shift_reg(data_counter-1) <= data_bit;
            end if;
      end if;
end process shift_proc;


--final check and store 32 bits data, data reg process
dr_reg_proc : process(clk)
begin
      if(rising_edge(clk)) then
            if(reset = '0') then
                  data_reg <= (others => '0');
                  temp <= (others => '0');
            elsif(data_counter = 32) then
            data_reg <= shift_reg(1 downto 0) & shift_reg(31 downto 2);

                  dt_rdy <= '1';
            else
               dt_rdy <= '0';
            end if;
      end if;
end process data_reg_proc;


-- 7 segment displays
Seg0: hex_to_7_seg port map(HEX7,data_reg(31 downto 28)) ;
Seg1: hex_to_7_seg port map(HEX6,data_reg(27 downto 24)) ;
Seg2: hex_to_7_seg port map(HEX5,data_reg(23 downto 20)) ;
Seg3: hex_to_7_seg port map(HEX4,data_reg(19 downto 16)) ;
Seg4: hex_to_7_seg port map(HEX3,data_reg(15 downto 12)) ;
Seg5: hex_to_7_seg port map(HEX2,data_reg(11 downto 8)) ;
Seg6: hex_to_7_seg port map(HEX1,data_reg(7 downto 4)) ;
Seg7: hex_to_7_seg port map(HEX0,data_reg(3 downto 0)) ;

--- Task-2 Command Discrimination and Execution
LEDG <= LG_reg;
LEDR <= LR_reg;
command <= data_reg(23 downto 16);

task2_proc : process(clk)
begin
      if(rising_edge(clk)) then
            if(reset = '0') then
                  LR_reg <= (others => '0');
                  LG_reg <= (others => '0');
            else
   if(dt_rdy = '1') then   --Data ready signal
            case command is
                        --For implementing Power on and Off
      when x"12" =>
           if(LG_reg(0) = '0' and LR_reg(0) = '0') then
                        LG_reg(0) <= '1';
                        LG_reg(7 downto 1) <= "0000000";
```

```
                              LR_reg(17 downto 0) <= "000000000000000000";

                  elsif(LG_reg(0) = '1' and LR_reg(0) = '0') then
                              LR_reg(0) <= '1'
                              LR_reg(17 downto 1) <= "00000000000000000";
                              LG_reg(7 downto 0) <= "00000000";

                        elsif(LG_reg(0) = '0' and LR_reg(0) = '1') then
                              LG_reg(0) <= '1';
                              LG_reg(7 downto 1) <= "0000000";
                              LR_reg(17 downto 0) <= "000000000000000000";
                                    end if;
                              --For ensuring that when channel 2 is ON
Channel 7 is OFF and vice versa
                  when x"02" =>
                        LR_reg(17 downto 0) <= "000000000000000100";

                  when x"07" =>
                        LR_reg(17 downto 0) <= "000000000010000000";
                              --Working of the mute button
                  when x"0C" =>
                                    LR_reg(12) <= not LR_reg(12);
                                    LR_reg(11 downto 0) <= "000000000000";
                                    LR_reg(17 downto 13) <= "00000";
      --Volume up/down resetting it and trying to implement the repeat code.
                  when x"1B" =>
                        LR_reg(17 downto 0) <= "000000010000000000";
                  when x"1F" =>
                        LR_reg(17 downto 0) <= "000000100000000000";

                  when others =>
                        LG_reg <= "00000000";
                        LR_reg <= "000000000000000000";
                  end case;
                  end if;
                  end if;
            end if;
      end process task2_proc;
end arc;
```

(ii) Testbench

```
--Team Members: Venkataramani and Yassine
--Testbench
--4th December 2018
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use std.textio.all ;
use ieee.std_logic_textio.all ;

use work.sim_mem_init.all;

entity test_RC_receiver is
end;

architecture test of test_RC_receiver is
```

```
component RC_receiver
generic (
      LC_on_max                           : integer := 450000);
port (
      HEX7                                : out std_logic_vector(6 downto 0);
      HEX6                                : out std_logic_vector(6 downto 0);
      HEX5                                : out std_logic_vector(6 downto 0);
      HEX4                                : out std_logic_vector(6 downto 0);
      HEX3                                : out std_logic_vector(6 downto 0);
      HEX2                                : out std_logic_vector(6 downto 0);
      HEX1                                : out std_logic_vector(6 downto 0);
      HEX0                                : out std_logic_vector(6 downto 0);

      rd_data                                   : out std_logic;

      clk                                       : in std_logic;
      data_in                                   : in std_logic;
      reset                               : in std_logic);
end component;

constant LC_on_max                        : integer := 50;

signal rd_data                            : std_logic := '0';
signal clk                                : std_logic := '0';
signal reset                              : std_logic := '0';
signal data_in, n_data                    : std_logic := '0';

constant num_segs                         : integer := 8;
constant seg_size                         : integer := 7;
type seg_arr is array(0 TO num_segs-1) of std_logic_vector(seg_size-1 downto
0);
signal seg, expected                : seg_arr := ((others=> (others=>'0')));

constant in_fname                         : string := "input.csv";
constant out_fname                        : string := "output.csv";
file input_file,output_file               : text;

begin
      n_data <= not data_in; -- data comes into the FPGA inverted
      dev_to_test:  RC_receiver
            generic map(LC_on_max)
                  port map(seg(7), seg(6), seg(5), seg(4), seg(3), seg(2),
      seg(1), seg(0),rd_data, clk, n_data, reset);

      stimulus:  process
    variable input_line                 : line;
    variable WriteBuf                   : line;
    variable in_char                    : character;
    variable in_bit                     : std_logic_vector(7 downto 0);
    variable out_slv                    : std_logic_vector(7 downto 0);
    variable ErrCnt                     : integer := 0 ;
    variable read_rtn                   : boolean;

      begin
            file_open(output_file, out_fname, read_mode);
```

```
        for k in 0 to num_segs-1 loop
                readline(output_file,input_line);
                for i in 0 to 1 loop
                        read(input_line, in_char, read_rtn);
out_slv := std_logic_vector(to_unsigned(character'pos(in_char),8));

                        if(i = 0) then
        expected(k)(6 downto 4) <= ASCII_to_hex(out_slv)(2 downto 0);
                        else
                        expected(k)(3 downto 0) <= ASCII_to_hex(out_slv);
                        end if;
                end loop;
        end loop;
        file_close(output_file);

        file_open(input_file, in_fname, read_mode);
        wait for 15 ns;
        reset <= '1';

        while not(endfile(input_file)) loop
                -- read an input bit
                readline(input_file,input_line);
                while true loop
                        read(input_line,in_char, read_rtn);
                        if(read_rtn = false) then
                                exit;
                        end if;
in_bit := std_logic_vector(to_unsigned(character'pos(in_char),8));
                        if(in_bit /= x"30" and in_bit /= x"31") then
                        exit;
                        end if;
                        data_in <= in_bit(0);
                        clk <= '0';
                        wait for 10 ns;
                        clk <= '1';
                        wait for 10 ns;
                end loop;
        end loop;

        file_close(input_file);
        wait for 10 ns;

        for k in 0 to num_segs-1 loop
                if (expected(k) /= seg(k)) then
        write(WriteBuf, string'("ERROR:  7 seg display failed at k = "));
        write(WriteBuf, k);
        write(WriteBuf, string'("expected = "));
        write(WriteBuf, expected(k));
        write(WriteBuf, string'(", seg = "));
        write(WriteBuf, seg(k));

                        writeline(Output, WriteBuf);
                        ErrCnt := ErrCnt+1;
                end if;
        end loop;

      if (ErrCnt = 0) then
```

```
                        report "SUCCESS!!!  RC receiver Test Completed";
                else
                        report "The RC receiver device is broken" severity warning;
                end if;
        end process;
end test;
```

References:

[1] Final Project slides

[2] http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2002_w/vhdl/IR_Receiver/infrared.htm