University of Colombo
Department of physics

Hardware synthesis using VHDL

Chandima Kasun Edirisinghe
Index 12033
August, 2017

Abstract

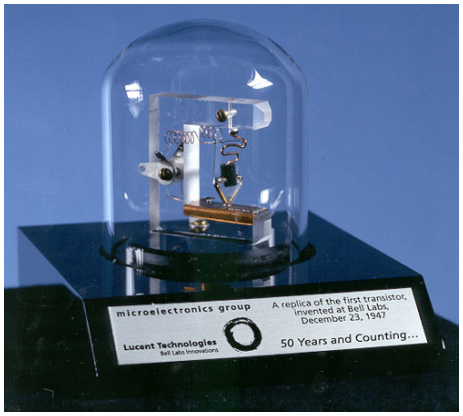
The purpose of this report is to synthesize digital circuits on complex programmable logic devices (CPLD). In this process, very high speed integrated circuit (VHSIC) hardware description language called VHDL was used. The first part of this practical is based on synthesizing combinational and sequential logics using VHDL. For this process, a Basys2 Spartan-3E FGPA board was used. In synthesizing logics, an AND gate, a BCD to SSD decoder and a mod 10 counter are synthesized. The second part of the practical is based on direct digital synthesis (DDS) using the Basys2 FPGA board combined with a R-2R ladder. In this process, a saw tooth signal, a triangular signal, a rectangular signal with adjustable duty cycle and a sinusoidal signal are synthesized.

Contents

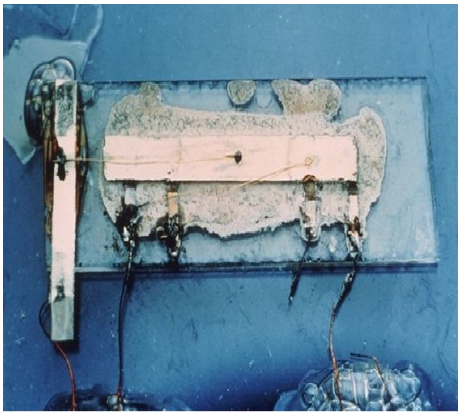
List of Figures

1 Introduction

The electronic world turned a new page in its advancement with the invention of the transistor by Physicists John Bardeen, Walter Brattain, and William Shockley. This was a revolution in the contemporary electronic world to an extent that it won the Nobel prize for Physics in 1956. The next big leap from the transistor was the integrated circuit (IC). It was the next big thing in the contemporary electronic market.



(a) A replica of the first working transistor



(b) The first working IC

Figure 1.1: The first working transistor and the IC in the world

From the development of the first working IC in 1958 by Jack Kilby, the IC technology has developed in leaps and bounds. Billions of components are contained inside modern ICs. With the exponential increase in usage of the ICs in a myriad of fields, the ICs were customized for suiting the need.

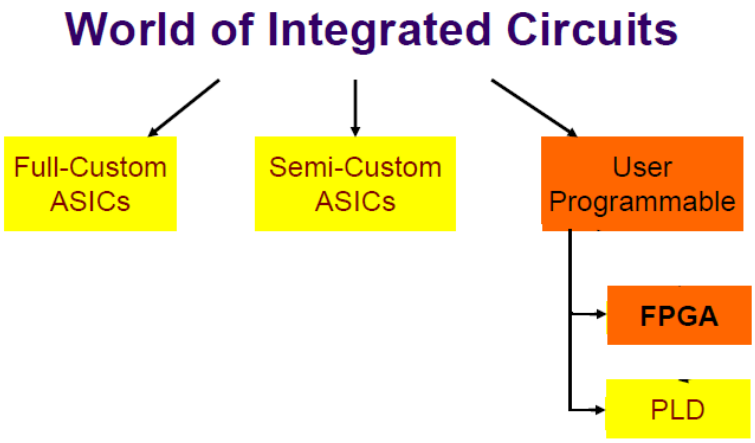


Figure 1.2: Classes of ICs

The main IC classes are the application specific type and the user programmable type. The application specific ICs (ASIC) are customized for a particular use, rather than for general-purpose use. The user programmable ICs like FPGA and PLD can be programmed by the user according to the requirement.

The main types of user programmable devices are the FPGA and PLD.

1. FPGA - field programmable gate array
2. PLD - programmable logic device

PLDs are electronic components used to build reconfigurable digital circuits.Prior to operating a PLD in a circuit, it must be programmed (reconfigured). Basically 2 categories of PLDs exist.

1. SPLD (simple programmable logic device).
2. CPLD (complex programmable logic device).

SPLDs are the simplest, smallest and least-expensive forms of PLDs. SPLDs can be used as an alternative to standard logic components (AND, OR and NOT gates).

CPLD is a combination of a fully programmable AND/OR array and a bank of macrocells. The AND/OR array is reprogrammable and can perform a multitude of logic functions. Macrocells are functional blocks that perform combinatorial or sequential logic, and also have the added flexibility for true or complement, along with varied feedback paths.

A field-programmable gate array (FPGA) is an IC designed to be configured by the user unlike an ASIC. advantageous comprise of the following components.

- CLBs (configurable logic blocks) - to perform logic.
- IOBs (input/output buffers) - for interfacing with the outside world.
- Programmable interconnections - for connecting CLBs and IOBs.
- RAM (random access memory).
- Multipliers.

FPGAs are more advantageous than ASICs due to the following reasons.

- Low development cost.
- Reconfigurability.
- Off-the-shelf.

The major vendors of FPGAs are as follows.

- Xilinx, Inc.
- Altera Corp.
- Atmel.
- Lattice Semiconductor.

A programming language is a formal language that specifies a set of instructions that can be used to produce various kinds of output. In FPGAs, a special type of programming language called the hardware description language (HDL) is used. It is a specialized computer language used to describe the structure and behavior of digital logic circuits. There are many HDLs like Abel, CUPL, Verilog, AHDL and VHDL.

VHDL is the programming language used in this practical. VHDL stands for VHSIC hardware description language. VHSIC is the abbreviation for very high speed integrated circuits. VHDL is intended for both circuit synthesis and circuit simulation.

VHDL is usually regarded as a code rather as a program due to the fact that the order of command execution is parallel (concurrent) unlike in regular computer programs. In VHDL, the statements placed inside PROCESS, FUNCTION or PROCEDURE are executed sequentially. The following figure depicts the design flow of a VHDL program.

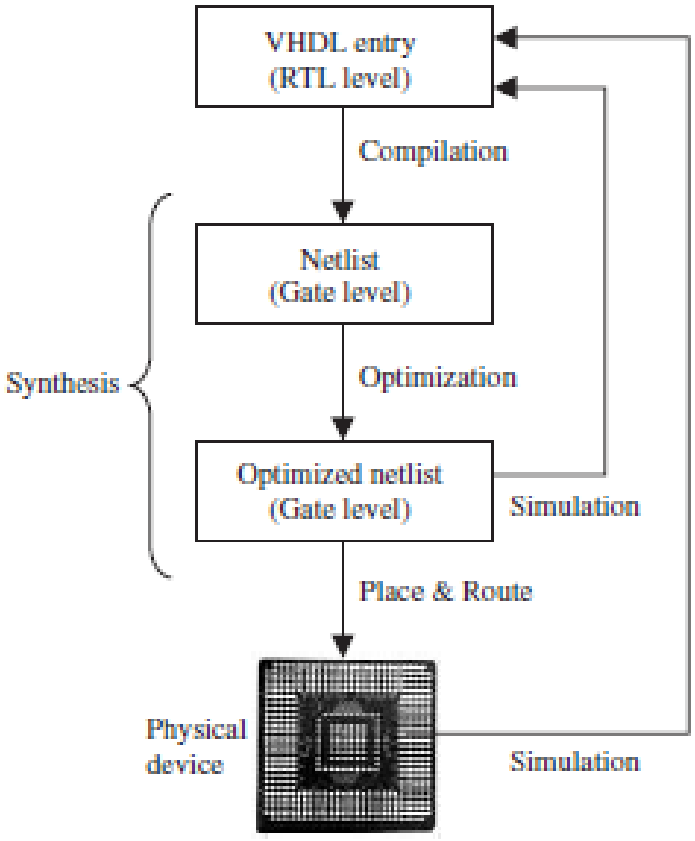


Figure 1.3: VHDL design flow

2 Theory

2.1 Combinational and sequential logic

2.1.1 Combinational logic

It's the type of logic in which the output depends only on the current inputs. In other words, it can be referred to as time-independent logic due to the fact that the output is a function dependent only on the present inputs. Logic gates are the building block of combinational logic circuits.

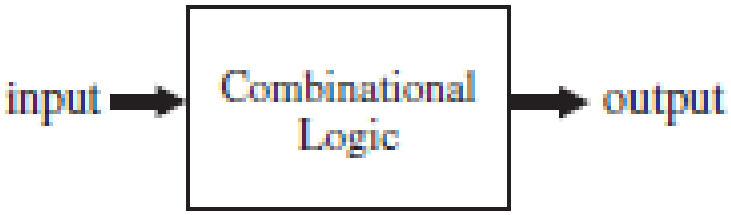


Figure 2.1: Combinational logic representation

Construction of combinational logic can be done using one of the following methods.

1. Sum of products (SOP) method - equations written as an OR of AND terms.
2. Product of sums (POS) method - equations written as an AND of OR terms.

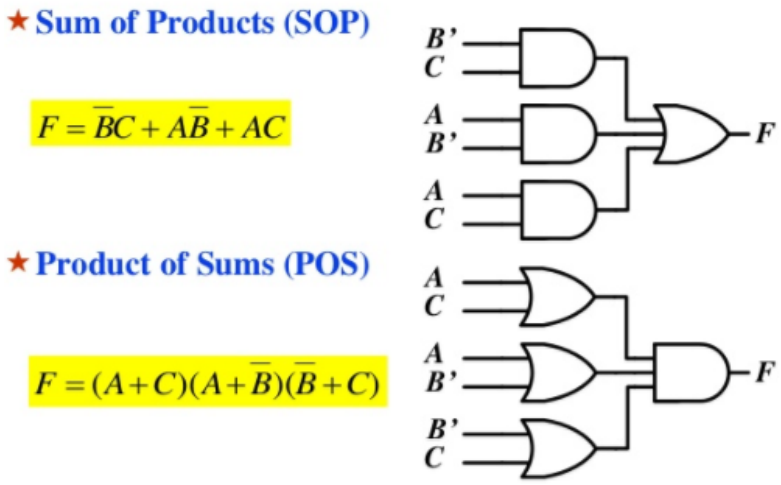


Figure 2.2: Implementation of the same equation using SOP and POS methods

2.1.2 Sequential logic

It's the type of logic in which the output does depend on the previous inputs (input history). In other words, the output is a function of not only the present input, but also of the past inputs. As there is a dependency of the output with past inputs, memory is required to store the past inputs. These memory (storage) units are connected to combinational logic blocks via feedback loops. Latches and flip-flops are the building block of combinational logic circuits.

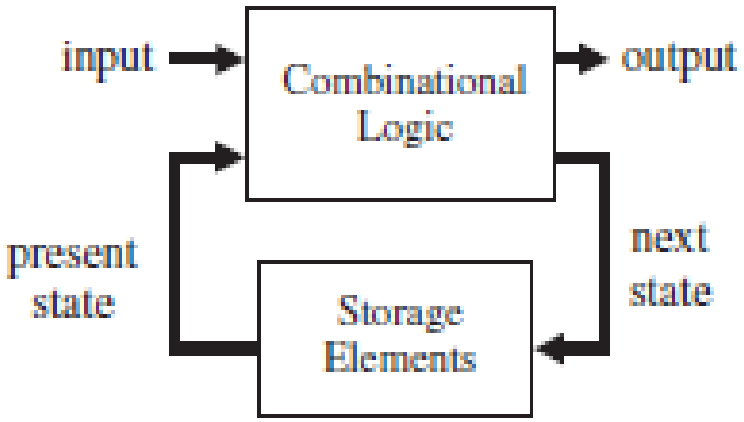


Figure 2.3: Sequential logic representation

There are 2 types of sequential logic.

1. Synchronous sequential logic.
2. Asynchronous sequential logic.

Synchronous sequential logic

In synchronous sequential circuits, the states and outputs change with respect to a clock signal. The basic memory element in sequential logic is the flip-flop. The outputs of all the flip-flops and their binary data are collectively called the state of the circuit. The next state is determined by the current state and the value of the input signals only when the clock pulse occurs. There are 2 main disadvantages of sequential logic circuits over the asynchronous logic circuits.

1. Maximum clock speed is dependent on the slowest path (critical path).
2. Distribution of the clock signal to each and every flip-flop.

Asynchronous sequential logic

In asynchronous sequential logic circuits, transition from one state to another is initiated by the change in the primary inputs, totally independent on a clock. Asynchronous sequential logic circuits are faster than the synchronous logic circuits as state changes aren't initiated by a clock. The speed of the circuit depends only on propagation delays in the logic gates used. The main disadvantage of asynchronous sequential logic circuits is the sensitivity to the order in which the signals arrive a latch or a flip-flop. This condition is called a race condition.

2.2 Latches and flip-flops

2.2.1 Latches

Logic gates are the building blocks of latches.Latches contain only 2 output states.

1. High-output.
2. Low-output.

Latches can act as memory devices by saving one bit of data until the device has power. This memory comes as a consequence of the feedback paths connecting the logic gates inside the latch. Latches are asynchronous (clock independent) unlike like flip-flops.

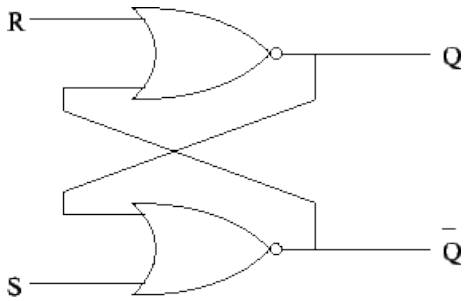


Figure 2.4: A simple SR latch

2.2.2 Flip-flops

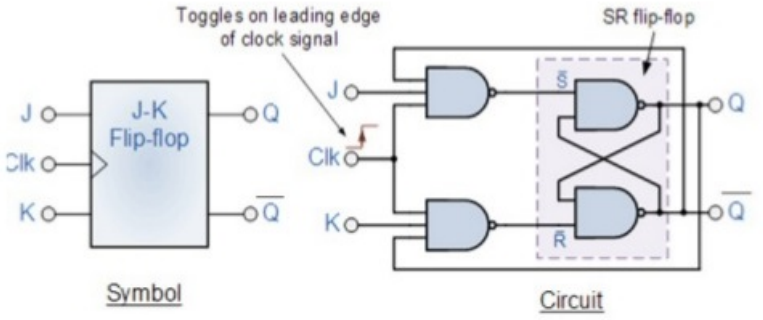
Flip-flops are comprised of latches. The main difference from latches is that flip-flops are synchronous.

Latches	Flip Flops
Latches are building blocks of sequential circuits and these can be built from logic gates	Flip flops are also building blocks oof sequential circuits. But, these can be built from the latches.
Latch continuously checks its inputs and changes its output correspondingly.	Flip flop continuously checks its inputs and changes its output correspondingly only at times determined by clocking signal
The latch is sensitive to the duration of the pulse and can send or receive the data when the switch is on	Flipflop is sensitive to a signal change. They can transfer data only at the single instant and data cannot be changed until next signal change. Flip flops are used as a register.
It is based on the enable function input	It works on the basis of clock pulses
It is a level triggered, it means that the output of the present state and input of the next state depends on the level that is binary input 1 or 0.	It is an edge triggered, it means that the output and the next state input changes when there is a change in clock pulse whether it may a +ve or -ve clock pulse.

Figure 2.5: Differences between latches and flip-flops

2.3 Finite state machines (FSMs)

It's a mathematical model of computation in which a process is equated to an abstract machine whose state can be exactly one of a finite number of states at any given time. A FSM can make a transition from one state to another in response to one or more external inputs.



Clk	J	k	Q_n	$\overline{Q_n}$
↑	0	0	No change	
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	$\overline{Q_n}$	Q_n

Toggle

Figure 2.6: Truth table of a JK flip-flop

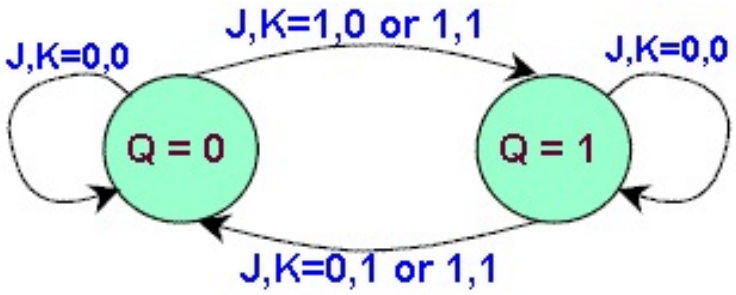


Figure 2.7: State diagram of a JK flip-flop

Based on the actions followed in order to generate an output based on a given input and/or a state using actions, FSMs can be divided to 2 categories.

1. Moore machine.
2. Mealy machine.

2.3.1 Moore machine

In this model, the output is dependent only on the current state of the machine. The outputs change synchronously with the state transition triggered by the active clock edge applied to the memory elements.

2.3.2 Mealy machine

In this model, the output depends on both the current state and the inputs to the machine. The output of a Mealy machine doesn't need to be governed by a clock pulse. It can change

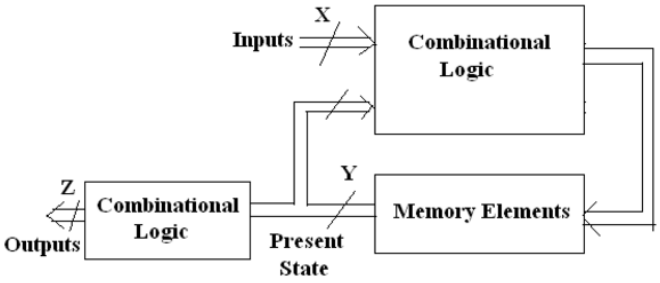


Figure 2.8: Moore machine

asynchronously in response to any change in the input.

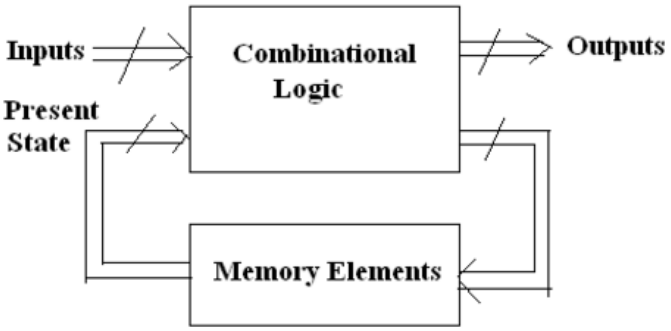


Figure 2.9: Mealy machine

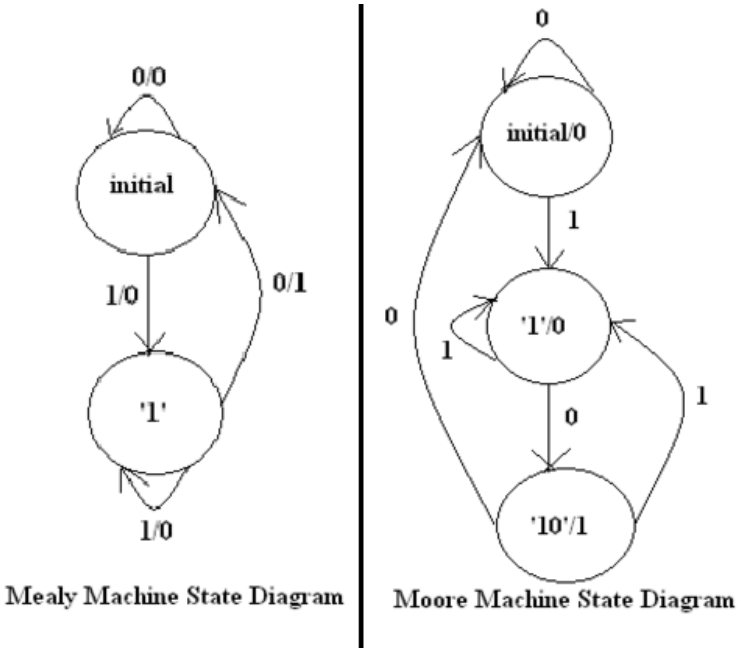


Figure 2.10: Comparison of Mealy and Moore Machines while designing '10' pattern detector.

2.4 Johnson counter

Johnson counter is a type of a ring counter used in hardware logic designs like FPGAs to create complex finite state machines. Ring counters are comprised of shift registers.

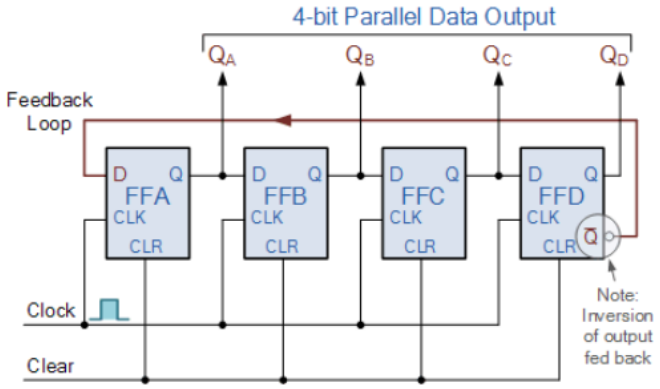


Figure 2.11: Implementation of a 4 bit Johnson counter

Clock Pulse No	FFA	FFB	FFC	FFD
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

Figure 2.12: Truth table of a 4 bit Johnson counter

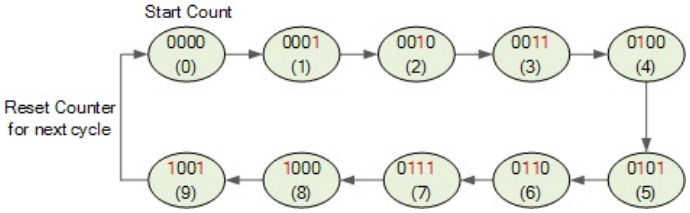


Figure 2.13: State diagram of a 4 bit Johnson counter

2.5 JTAG

JTAG is the abbreviation for the Joint Test Action Group. JTAG implements standards for on-chip instrumentation in electronic design automation (EDA) as a complementary tool to digital simulation. Processors often use JTAG to provide access to their debug functions. FPGAs and CPLDs use JTAG as the doorway to their programming functions.

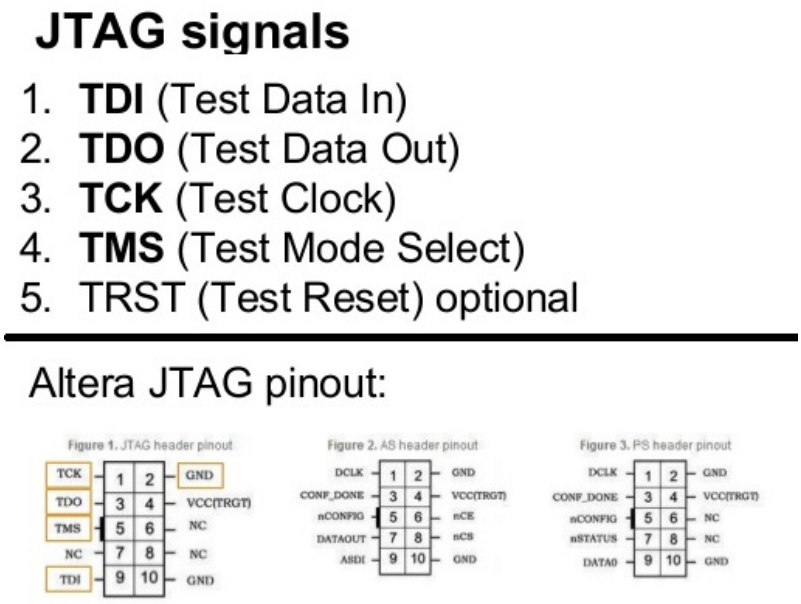


Figure 2.14: JTAG components

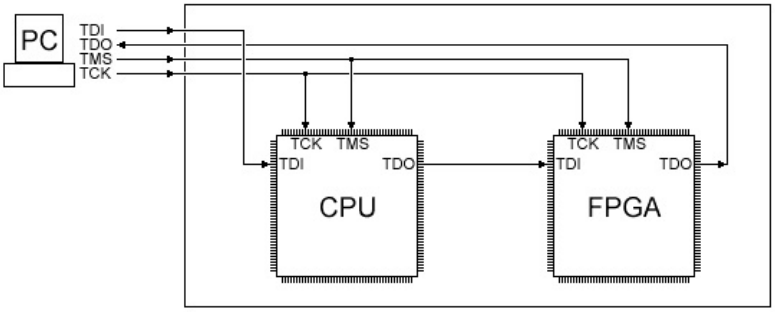


Figure 2.15: Programming through JTAG

2.6 Antifuse

An antifuse is a device which behaves in the opposite way to fuses. While fuses burn when the voltage through the fuse exceeds some threshold stopping the conduction of current, antifuse starts to conduct permanently when the voltage exceeds some threshold. Incorporating antifuse technology in PLDs enable the user to permanently program the device a single time. After that, the device can't be erased.

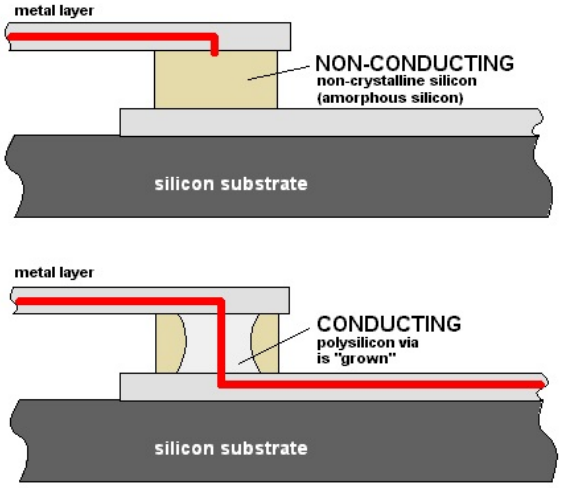


Figure 2.16: Antifuse surface

2.7 Direct digital synthesis (DDS)

DDS is the process of generating analog signals using digital techniques. The analog signals are synthesized from values stored in memory. A digital to analog converter is used to convert the needed digital signal to an analog signal. Some advantages of using DDS are as follows.

- The ability to generate arbitrary frequencies with accuracy and stability.
- The reliability and repeatability of the signals produced by DDS.
- High frequency resolution.

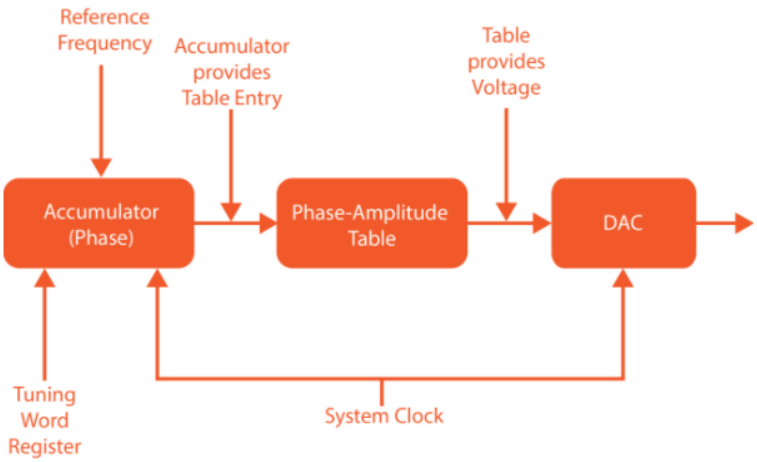


Figure 2.17: Block diagram representing the main components in a DDS system

3 Methodology

As the practical was based on FPGA which was a new technology, the primary step was to get familiar with the Basys2 trainer board and the ISE used to create bit files needed for programming the FPGA. So, the Basys2 FPGA board and its data sheet were studied first.

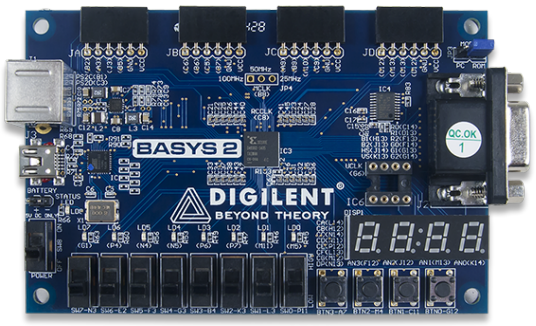


Figure 3.1: Front look of the Basys2 Spartan-3E FGPA board

FPGA pin definition table color key	
Grey	Not available to user
Green	User I/O devices
Yellow	Data ports
Tan	Pmod connector signals
Blue	USB signals

Basys2 Spartan-3E pin definitions

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
C12	JD1	P11	SW0	N14	CC	B2	JA1	P8	MODE0	M7	GND
A13	JD2	M2	USB-DB1	N13	DP	C2	USB-WRITE	N7	MODE1	P5	GND
A12	NC	N2	USB-DB0	M13	AN2	C3	PS2D	N6	MODE2	P10	GND
B12	NC	M9	NC	M12	CG	D1	NC	N12	CCLK	P14	GND
B11	NC	N9	NC	L14	CA	D2	USB-WAIT	P13	DONE	A6	VDDO-3
C11	BTN1	M10	NC	L13	CF	L2	USB-DB4	A1	PROG	B10	VDDO-3
C6	JB1	N10	NC	F13	RED2	L1	USB-DB3	N8	DIN	E13	VDDO-3
B6	JB2	M11	LD1	F14	GRN0	M1	USB-DB2	N1	INIT	M14	VDDO-3
C5	JB3	N11	CD	D12	JD4	L3	SW1	P1	NC	P3	VDDO-3
B5	JA4	P12	CE	D13	RED1	E2	SW6	B3	GND	M8	VDDO-3
C4	NC	N3	SW7	C13	JD3	F3	SW5	A4	GND	E1	VDDO-3
B4	SW3	M6	UCLK	C14	RED0	F2	USB-ASTB	A8	GND	J2	VDDO-3
A3	JA2	P6	LD3	G12	BTN0	F1	USB-DSTB	C1	GND	A5	VDDO-2
A10	JC3	P7	LD2	K14	AN3	G1	LD7	C7	GND	E12	VDDO-2
C9	JC4	M4	BTN2	J12	AN1	G3	SW4	C10	GND	K1	VDDO-2
B9	JC2	N4	LD5	J13	BLU2	H1	USB-DB6	E3	GND	P9	VDDO-2
A9	JC1	M5	LD0	J14	HSYNC	H2	USB-DB5	E14	GND	A11	VDDO-1
B8	MCLK	N5	LD4	H13	BLU1	H3	USB-DB7	G2	GND	D3	VDDO-1
C8	RCCLK	G14	GRN2	H12	CB	B14	TMS	H14	GND	D14	VDDO-1
A7	BTN3	G13	GRN1	J3	JA3	B13	TCK-FPGA	J1	GND	K2	VDDO-1
B7	JB4	F12	AN0	K3	SW2	A2	TDO-USB	K12	GND	L12	VDDO-1
P4	LD6	K13	VSYNC	B1	PS2C	A14	TDO-S3	M3	GND	P2	VDDO-1

Figure 3.2: Pin definitons of the Basys2 Spartan-3E FGPA board

Then, the next step was to install the ISE Design Suite and the Digilent Adept softwares. When installing the ISE Design Suite, the System Edition was chosen.

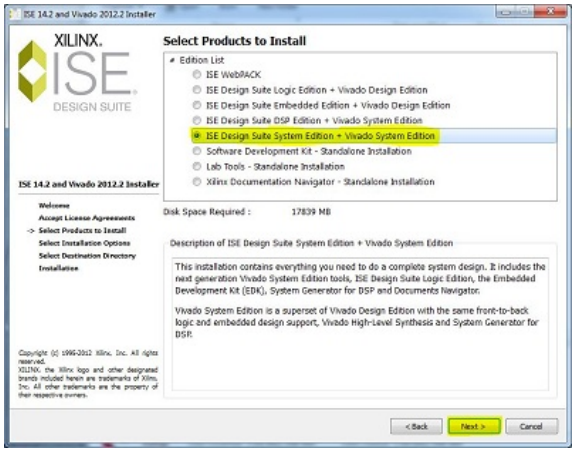


Figure 3.3: Installation of the ISE Design Suite: System Edition

After the correct version of the ISE was installed, the next objective was to get familar with GUI of the ISE. In this process, a new project was created. When creating the new project in the ISE, the following selections were done in order to choose the correct device and the design flow. After a new project was made the needed codes could be written in a new file by creating a new VHDL Module with any name. This could be achieved by creating a VHDL module in the new Source wizard panel. The assigning of the inputs and outputs of the created program could be done through a UCF file (Implementation Constraint File). A UCF file could be created by selecting the Implementation Constraint File in the new Source wizard panel.

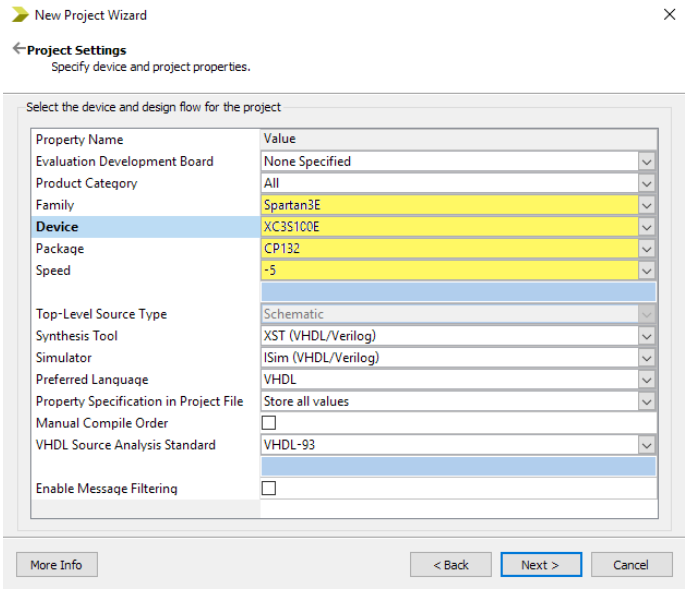


Figure 3.4: Selecting the correct device and the design flow

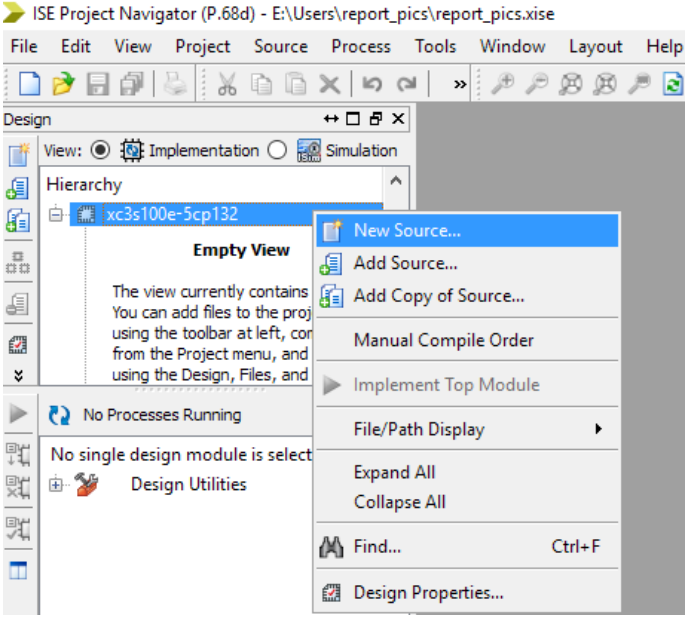
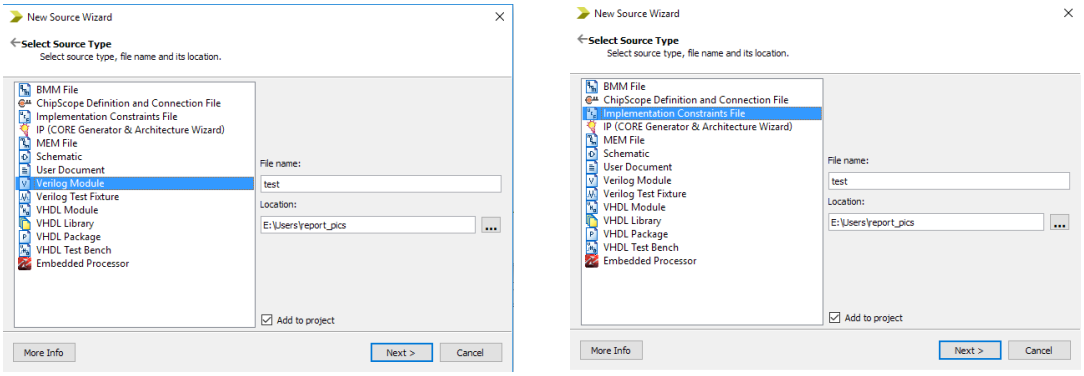


Figure 3.5: Obtaining the new Source wizard panel



(a) Creating a new VHDL module

(b) Creating a new UCF file

Figure 3.6: Using the new Source wizard panel

After creating a new project and the needed program was coded, the next step was to check the code for errors and generate the bit file needed. In checking the code for errors, the following procedure was followed.

1. Synthesis of the XST.
2. Implementation of the device.
3. Generation of the programming file.

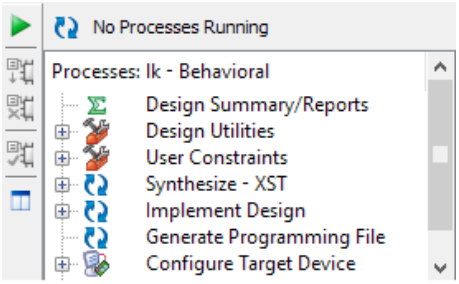


Figure 3.7: Compilation of the code

After the compilation and the generation of the programming file was successful, the created bit file could be uploaded to the FPGA board using the Adept software.

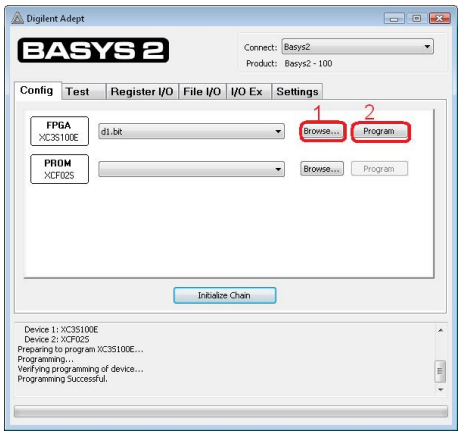


Figure 3.8: Selection of the bit file

In addition to synthesizing a program in a FPGA board, the created program can be simulated virtually. This could be done by the ISE Simulator (ISIM) which comes as an embedded feature of the ISE. The simulator can be accessed by selecting the Simulation view in the design menu.

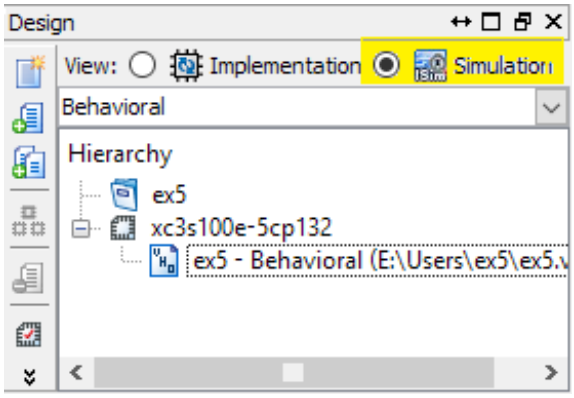


Figure 3.9: Selection of the ISIM

Then, the behaviour of the model can be simulated after completing the behavioural syntax check procedure.

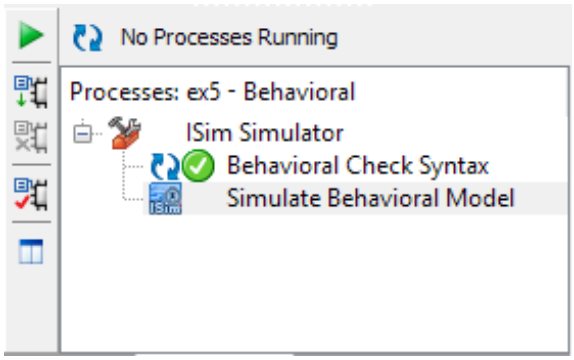


Figure 3.10: Accessing the ISIM

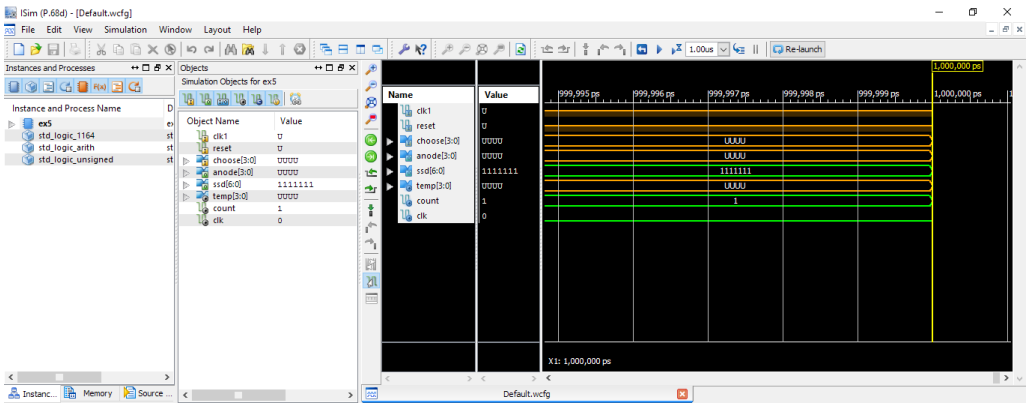


Figure 3.11: Interface of the ISIM

After getting familiar with the FPGA and the ISE, the next objective was to complete the specified tasks. These specified tasks were divided into 2 main sections.

1. Introduction to VHDL.
2. DDS using FPGA.

3.1 Introduction to VHDL

The first section of the practical included the following exercises.

1. Synthesis of a 2-input AND gate.
2. Synthesis of a BCD to SSD decoder.
3. Modification of the BCD to SSD decoder with a test button.
4. Synthesis of a mod 10 counter.
5. Combination of the mod 10 counter and the BCD to SSD decoder.

3.1.1 Synthesis of a 2-input AND gate

In this exercise, a 2-input AND gate was synthesized. In this process, the inputs were assigned to switches SW1 (L3) and SW2 (K3) and the output was assigned to the LED LD0 (M5). The code and the appropriate UCF file are attached in the Appendix A.

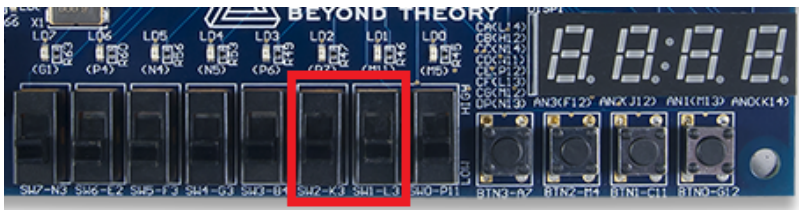


Figure 3.12: Switches which represents the inputs of the AND gate

3.1.2 Synthesis of a BCD to SSD decoder

In this exercise, a BCD to SSD decoder was synthesized. In this process, 4 switches were used as the inputs and the output was displayed through the SSD. Any binary number between 0 and 9 could be displayed by switching on the 4 switches SW0 (P11), SW1 (L3), SW2 (K3) and SW4 (B4) in which SW0 represented the least significant bit (LSB) and SW3 represented the most significant bit (MSB). The digit which represented the selected number could be changed using switches SW4, SW5, SW6 and SW7. Then, the program was simulated in ISIM to view the output. The code and the appropriate UCF file are attached in the Appendix B.

(a) Switches to change the number which is displayed on the SSD

(b) Swithces to select the digit which is lit

Figure 3.13: Action of switches in the BCD to SSD decoder

3.1.3 Modification of the BCD to SSD decoder with a test button

In this exercise, the BCD to SSD decoder constructed in the previous exercise was modified by adding a test button. The purpose of a test button was to override the current operation of the FPGA and illuminate all the digits of the SSD. For this operation, the push button BTN3 (A7) was used. The code and the appropriate UCF file are attached in the Appendix C.

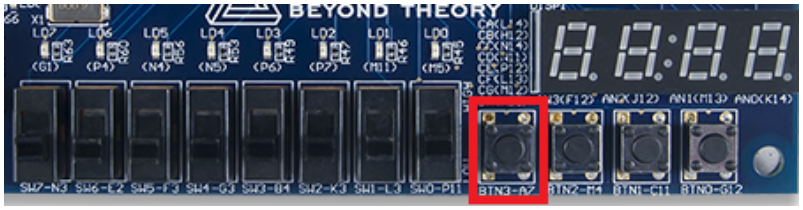


Figure 3.14: The push button used as the test button

3.1.4 Synthesis of a mod 10 counter

In this exercise, a mod 10 counter was synthesized. In this process, the LEDs LD0 (M5), LD1 (M11), LD2 (P7) and LD3 (P6) were used as the outputs. The LD3 was considered as the most significant bit and LD0 was considered as the least significant bit. As the clock frequency of the FPGA clock was 50 MHz, the action of the counter was not visible to the naked eye. So, a new clock of 1 Hz was formed and the counter was increased at the rising edge of the new clock of 1 Hz. A push button BTN3 (A7) was added to reset the mod 10 counter and initialize it. Then, the program was simulated in ISIM to view the output. The code and the appropriate UCF file are attached in the Appendix D.

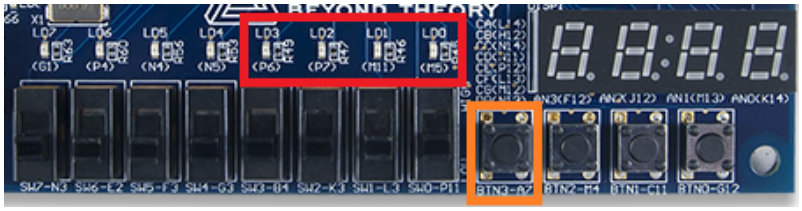


Figure 3.15: The output LEDs and the reset button

3.1.5 Combination of the mod 10 counter and the BCD to SSD decoder

In this exercise, the mod 10 counter and the BCD to SSD decoder which were synthesized earlier were combined together. In this process, the output (the numbers from 0 to 9) was displayed through the SSD. The digits which showed the output could be selected by using switches SW4 (G3), SW5 (F3), SW6 (E2) and SW7 (N3). The counter could be reset and initialized to 0 using the push button BTN3 (A7). The code and the appropriate UCF file are attached in the Appendix E.

Figure 3.16: The switches that select the needed digit in the SSD and the reset button

3.2 DDS using FPGA

This section comprised of 3 exercises.

1. Production of a ramp signal.
2. Production of a square signal.
3. Production of a sinusoidal signal.

3.2.1 Production of a ramp signal

This exercise consisted of 3 steps.

1. Production of a saw tooth signal using an up counter
2. Production of a square tooth signal using a down counter
3. Production of a triangular signal using an up and down counter

Production of a saw tooth signal using an up counter

Here, as the first step, a 8 bit up counter was constructed. Then, the outputs of that counter were connected to a R-2R ladder. When connecting the outputs of the counter to the ladder, care had to be taken so that the bit order of the counter and the ladder were the same (LSB to LSB and MSB to MSB). Then, the output of the R-2R ladder was observed through the DSO.

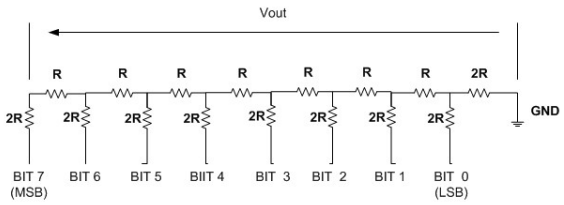


Figure 3.17: Connecting a R-2R ladder

Production of a saw tooth signal using an up counter

Here, as the first step, a 8 bit down counter was constructed. Then, the outputs of that counter were connected to a R-2R ladder. Then, the output of the R-2R ladder was observed through the DSO.

Production of a triangular signal using an up and down counter

Here, as the first step, a 8 bit up/down counter which counts from 0 to 255 and then fron 255 to 0 was constructed. Then, the outputs of that counter were connected to a R-2R ladder. Then, the output of the R-2R ladder was observed through the DSO.

Finally, both the up and down counters were incorporated together so that any of up, down or up and down counters could be selected thereby selecting between a saw tooth signal or a triangular signal. For this process the switches SW1 (P11) were SW1 (L3) used. The selection of the counter type was done as follows.

1. If only SW0 was switched on, the up counter was selected forming a saw tooth signal.
2. If only SW1 was switched on, the down counter was selected forming a saw tooth signal.
3. If both SW0 and SW1 were switched on, the up and down counter was selected forming a triangular signal.

Additionally, a reset button was added to reset the counters. For this purpose, the push button BTN3 (A7) was used. The code and the appropriate UCF file are attached in the Appendix F.

Figure 3.18: Switches and the button used in the selection of the counter type and resetting of the counter respectively

3.2.2 Production of a square signal

In this exercise, a square signal was produced. For this process, a 8 bit up and down counter coupled with a R-2R ladder and a comparator were used. The purpose of the counter and the R-2R ladder was to produce a triangular wave. The comparator was used to make a square signal from the supplied triangular signal.

As the comparator, a ua 741 op amp operating in single rail (3.3 V and 0 V) was used. To supply power to the op amp, the VCC and GND pins of the FPGA board were used. The triangular signal produced by the 8 bit up and down counter coupled with a R-2R ladder was connected to the non-inverting input and the reference voltage was connected to the inverting input.

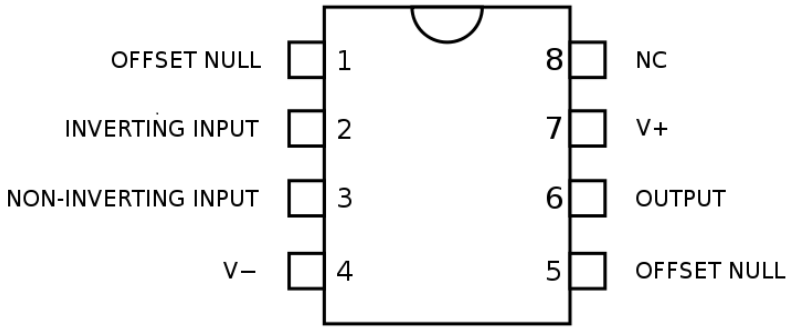


Figure 3.19: The pin configuration of the ua 741 op amp

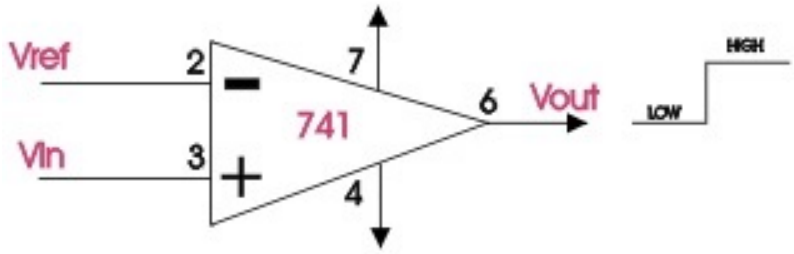


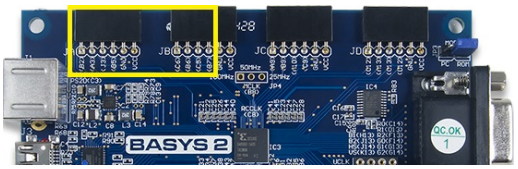
Figure 3.20: The ua 741 op amp as the comparator

As the duty cycle of the produced square signal from the comparator was dependent on the reference voltage connected to the inverting input of the op amp, 4 switches were used to produce a variable reference voltage. Here, the switches SW4 (G3), SW5 (F3), SW6 (E2) and SW7 (N3) were used. The SW4 was considered the LSB and SW7 was considered the MSB. By using the switches, a 4 bit binary number from 0 to 15 could be generated. Then, the outputs of these 4 switches were connected to another R-2R ladder for the purpose of converting the digital signal to an analog voltage. The output of this R-2R ladder was connected to the inverting input. So, by

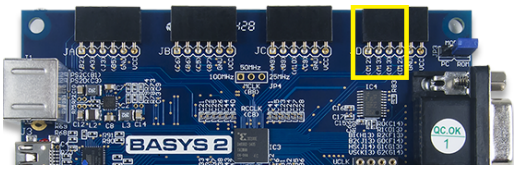
using the switches, the duty cycle of the square signal could be controlled.

A reset button was used to reset the counters. For this purpose, the push button BTN3 (A7) was used. The code and the appropriate UCF file are attached in the Appendix G.

Figure 3.21: The switches used to control the reference voltage and the reset button



(a) The outputs of the up and down counter connected to the R-2R ladder



(b) Outputs to the other R-2R ladder to create the reference voltage

Figure 3.22: The outputs of the FPGA board

3.2.3 Production of a sinusoidal signal

In this exercise, a sinusoidal signal was produced. For this process, a look up table had to be used in order to generate the amplitudes of a sinusoidal signal. Then, the values in the look up table were input to a 8 bit R-2R ladder in order to create the needed sinusoidal signal.

The needed lookup table was generated using the Microsoft Excel software. In this process, the following steps were followed.

1. Generation of numbers from 0 to 180 with unit increments.
2. Conversion of the numbers to radians.
3. Getting the sin values of the radian values.
4. Multiplication of the sin values by 255.
5. Conversion of the numbers obtained in step 4 to 8 bit binary numbers.

Then, the produced binary numbers were incorporated with the vhdl code through a select case. The 8 bit binary pattern produced by the FPGA was provided to the afore mentioned R-2R ladder and the output of the ladder was viewed through the DSO. As the input,a reset button was used to reset the counters. For this purpose, the push button BTN3 (A7) was used. The code and the appropriate UCF file are attached in the Appendix H.

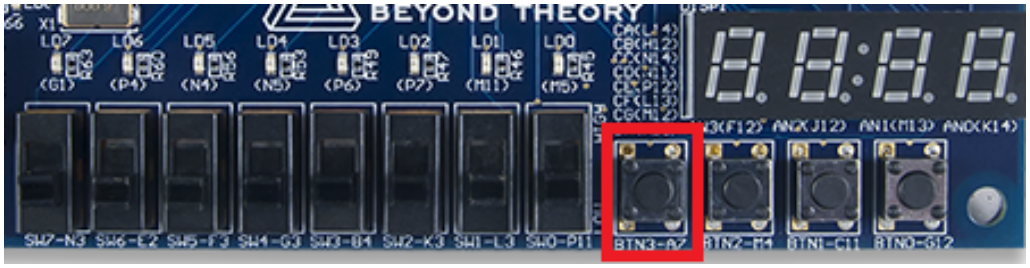


Figure 3.23: The push button used as the reset button

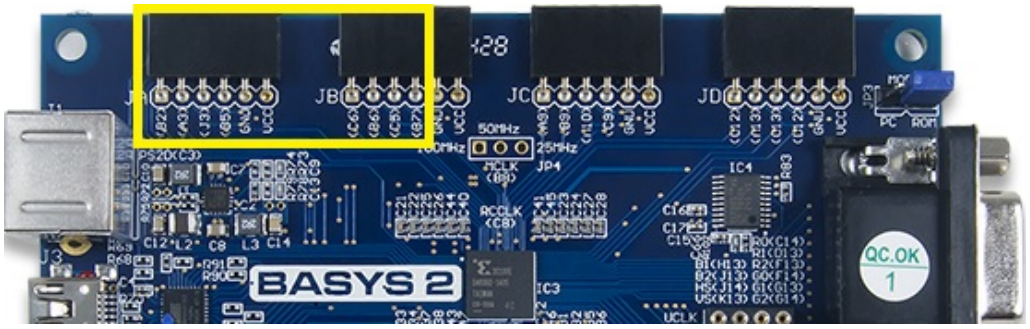


Figure 3.24: The pins of the FPGA connected to the R-2R ladder

4 Results and Analysis

4.1 Introdiction to VHDL

4.1.1 Synthesis of a 2-input AND gate

The AND gate when simulated from ISIM showed the following behaviour.

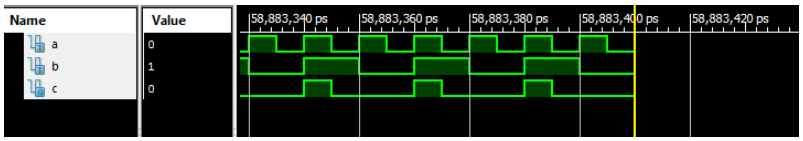


Figure 4.1: The output of the AND gate when simulated through ISIM

The AND gate when synthesized through the FPGA board showed the following behaviour.

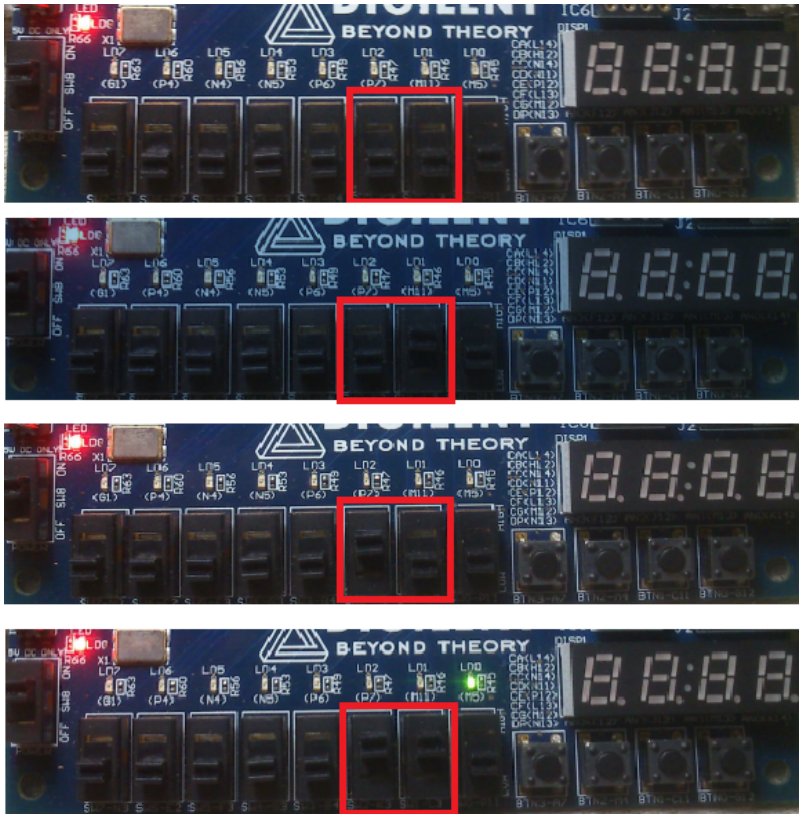


Figure 4.2: The output of the AND gate for various input states

As expected from an AND gate, the output became high only when all the inputs were high. This fact was clearly visible in the simulation as well as in the synthesized program. In the simulation, the signal denoted by c had become high only when both inputs a and b were being high. In the synthesis, the LED lit only when both the switches were high. So, it could be verified that this program imitated the behaviour expected of an AND gate and hence it can be concluded that it's an AND gate.

4.1.2 Synthesis of a BCD to SSD decoder

The BCD to SSD decoder when simulated from ISIM showed the following behaviour.

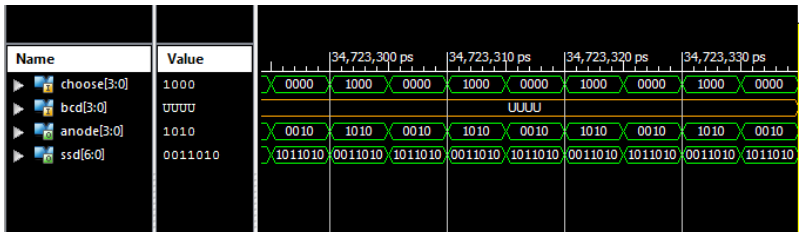


Figure 4.3: The output of the BCD to SSD decoder when simulated through ISIM

The BCD to SSD decoder when synthesized through the FPGA board showed the following behaviour.

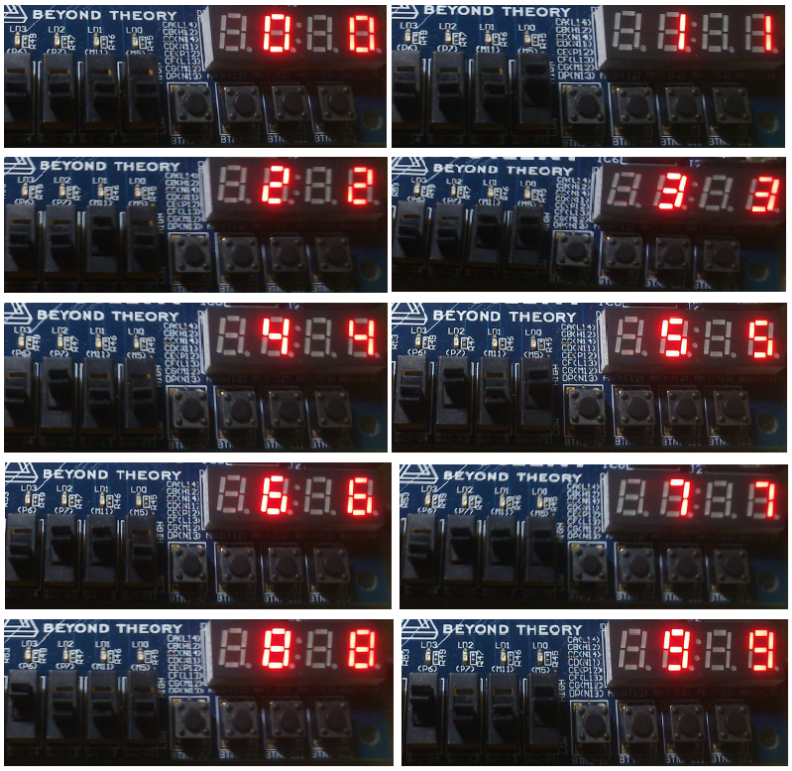


Figure 4.4: The outputs of the BCD to SSD decoder

By using the switches SW0 to SW3, the numbers from 0 to 9 could be displayed on the SSD. The switches SW4 to SW7 could be used to select the digit of the SSD which needed to be activated.

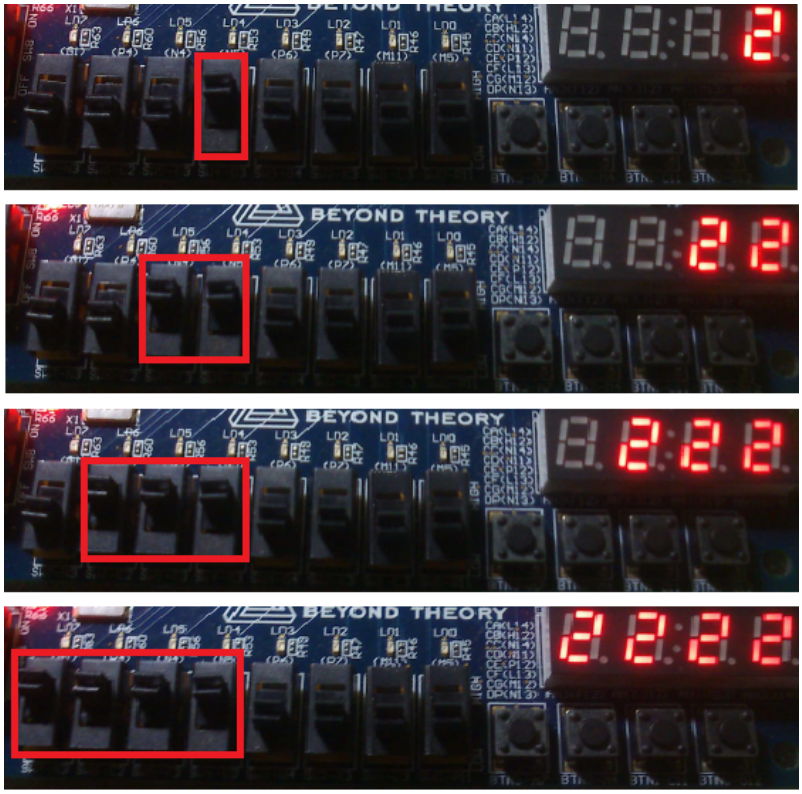


Figure 4.5: Controlling of the digits in the SSD

In this selection process, selection of the digits in the SSD could be easily done by biasing the transistors AN0 (K14), AN1 (M13) , AN2 (J12) and AN3 (F21) connected to the SSD working in common anode mode.

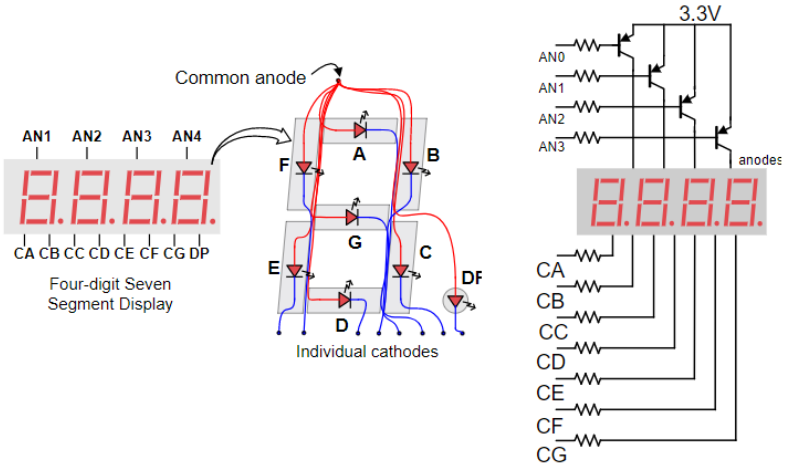
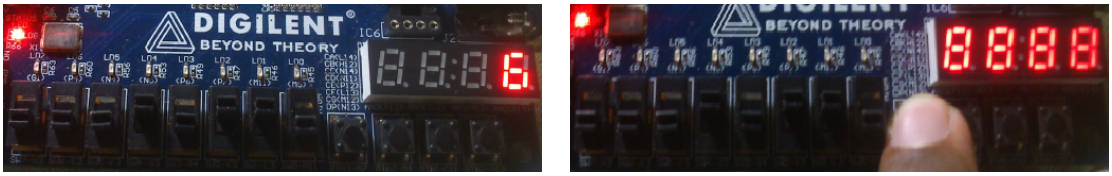


Figure 4.6: SSD diagram and the schematic

4.1.3 Modification of the BCD to SSD decoder with a test button

When the reset button was pressed, the all the digits in the SSD were lit.



(a) Output of the SSD when the test button wasn't pressed

(b) Output of the SSD when the test button was pressed

Figure 4.7: Action of the test button

4.1.4 Synthesis of a mod 10 counter

The mod 10 counter when simulated from ISIM showed the following behaviour.

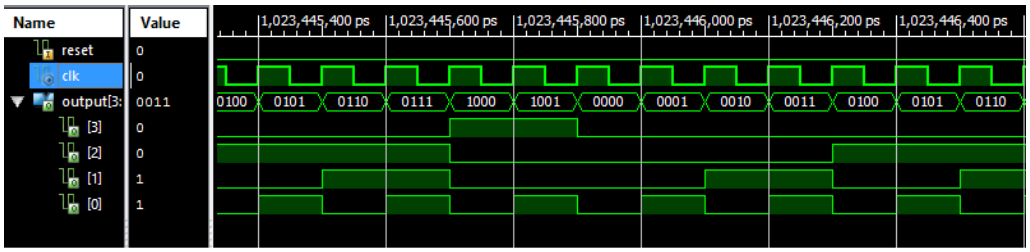


Figure 4.8: The output of the counter when simulated through ISIM

From the above figure, it could be clearly observed that the output varies from 0000 to 1001 which indeed depicts a mod 10 counter action. The binary values of the 4 bits of the output signal and their states were clearly visible in the ISIM simulation.
When the mod 10 counter was synthesized through the FPGA board, the outputs observed were as follows.

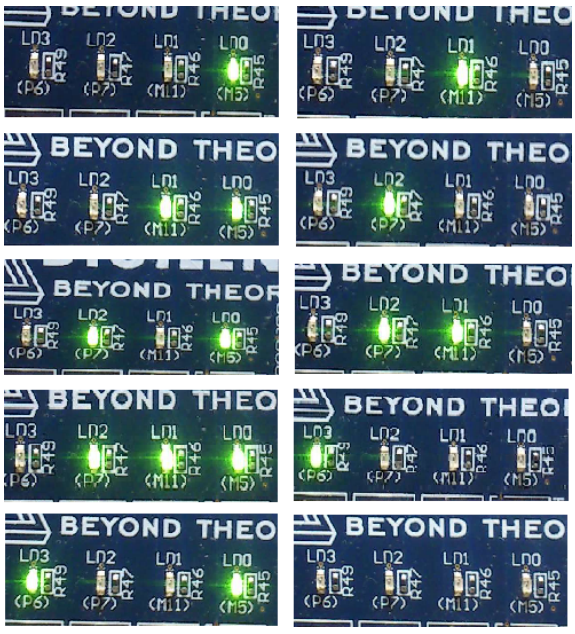


Figure 4.9: The output of the counter displayed by LEDs when synthesized through the FPGA

4.1.5 Combination of the mod 10 counter and the BCD to SSD decoder

The combination of the mod 10 counter and the BCD to SSD decoder when simulated from ISIM showed the following behaviour. In this case, it was not easy to decipher the signal patterns and their binary values of the SSD output.

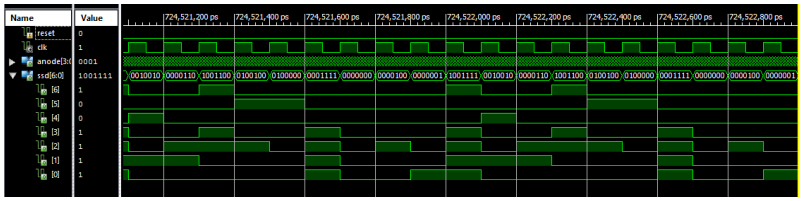


Figure 4.10: The output when simulated using ISIM

When the program was synthesized through the FPGA board, the outputs observed were as follows. It is clear from the below outputs that the SSD acts as a mod 10 which counted from 0 to 9.

Figure 4.11: The output when synthesized through the FPGA

The switches SW4 to SW7 could be used to select the digit of the SSD which needed to be activated.

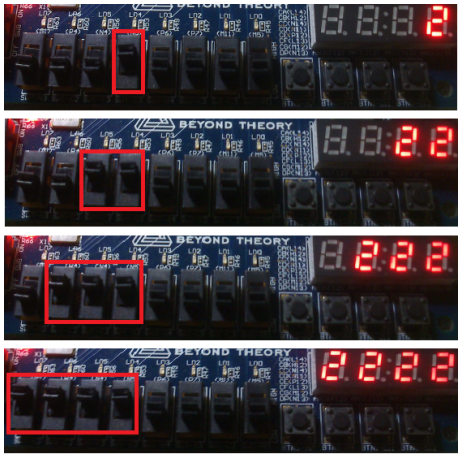


Figure 4.12: Controlling of the digits in the SSD

4.2 DDS usign FPGA

4.2.1 Production of a ramp signal

Production of a saw tooth signal using an up counter

The up counter when simulated from ISIM showed the following behaviour. The action of the up counter was clearly visible in the simulation. In this process, the *choose_up* variable was assigned 1 and *choose_down* variable was assigned 0. This was done as the code was written in such a way that when $choose\_up = 1$ and $choose\_down = 0$, the program worked as an up counter (line 84 in the code in Appendix F).

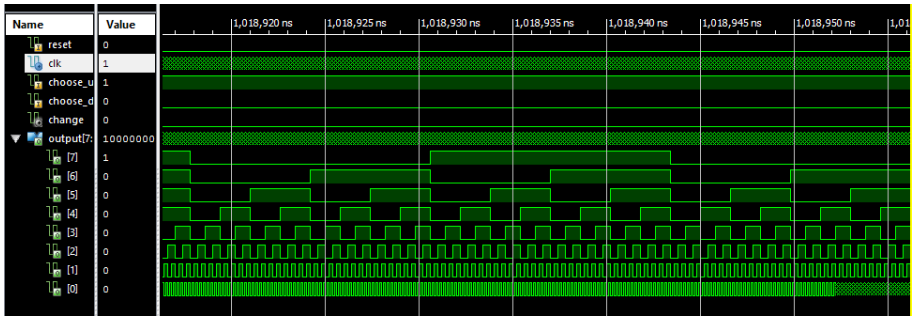


Figure 4.13: Up counter simulated by ISIM

The waveform obtained by the DSO connected to the R-2R ladder when the outputs of the up counter were connected to the R-2R ladder was as follows.

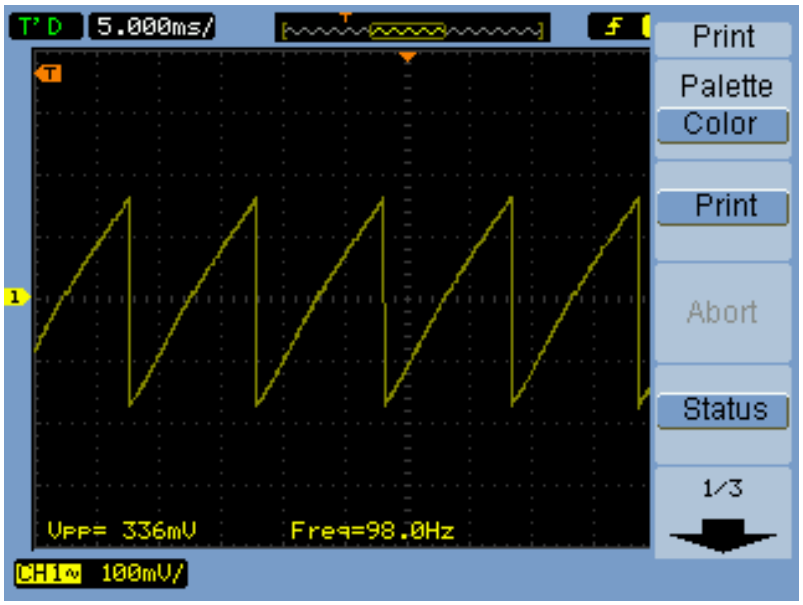


Figure 4.14: Waveform of the up counter

The frequency of the saw tooth signal produced had a frequency of 98 Hz and an amplitude (Vpp) of 336 mV. The frequency of the produced signal could be adjusted by changing the clock frequency of the "clk" signal. This could be done by adjusting the "count" value (line 72 of the code in Appendix F).

Production of a saw tooth signal using a down counter

The down counter when simulated from ISIM showed the following behaviour. The action of the down counter was clearly visible in the simulation. In this process, the *choose_up* variable was assigned 0 and *choose_down* variable was assigned 1. This was done as the code was written in such a way that when *choose_up* = 0 and *choose_down* = 1, the program worked as a down counter (line 89 in the code in Appendix F).

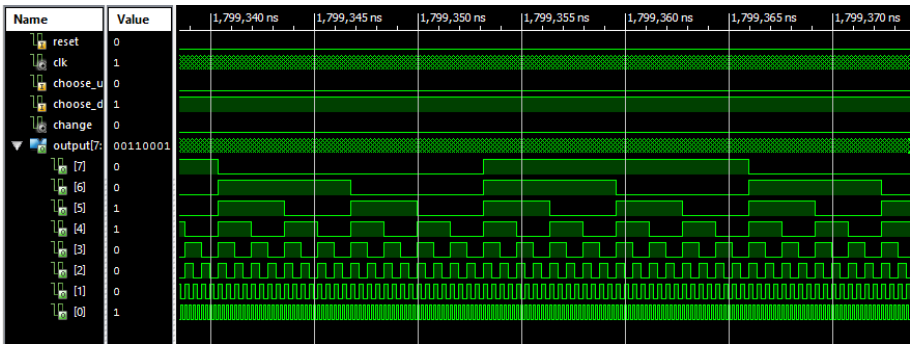


Figure 4.15: Down counter simulated by ISIM

The waveform obtained by the DSO connected to the R-2R ladder when the outputs of the down counter were connected to the R-2R ladder was as follows.

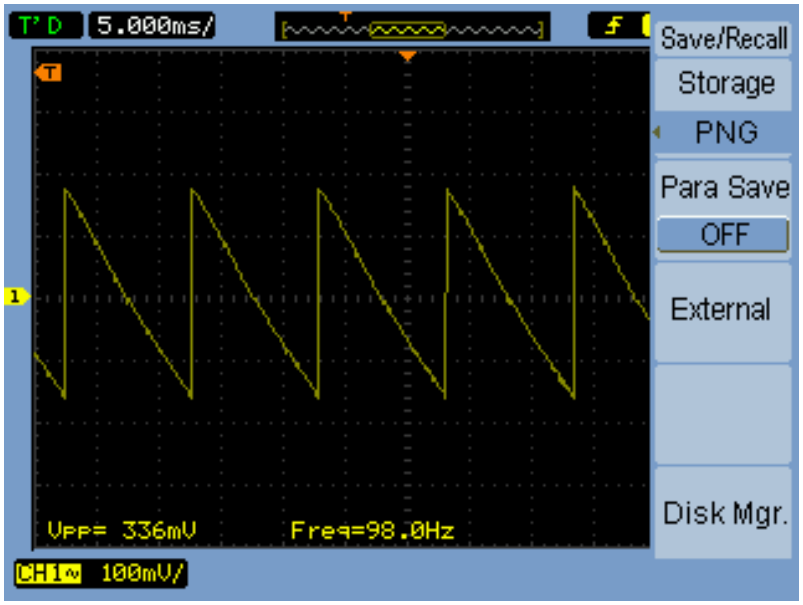


Figure 4.16: Waveform of the saw tooth signal formed by the down counter

The frequency of the saw tooth signal produced had a frequency of 98 Hz and an amplitude (Vpp) of 336 mV. The frequency of the produced signal could be adjusted by changing the clock frequency of the "clk" signal. This could be done by adjusting the "count" value (line 72 of the code in Appendix F).

Production of a triangular signal using an up and down counter

The up and down counter when simulated from ISIM showed the following behaviour. The action of the up and down counter was clearly visible in the simulation. In this process, the *choose_up* variable was assigned 1 and *choose_down* variable was assigned 1. This was done as the code was written in such a way that when $choose\_up = 1$ and $choose\_down = 1$, the program worked as an up and down counter (line 94 in the code in Appendix F).

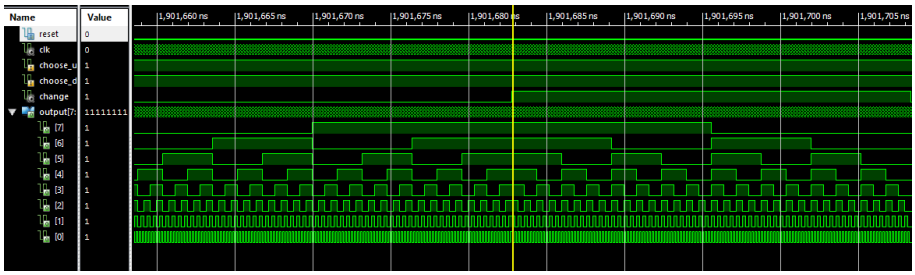


Figure 4.17: Up and down counter simulated by ISIM

The waveform obtained by the DSO connected to the R-2R ladder when the outputs of the up and down counter were connected to the R-2R ladder was as follows.

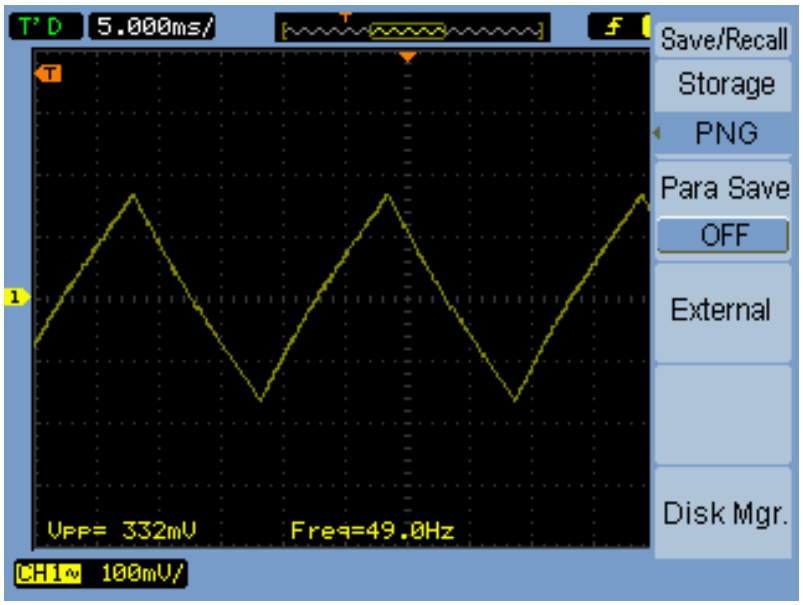


Figure 4.18: Waveform of the triangular signal formed by the up and down counter

The frequency of the triangular signal produced had a frequency of 49 Hz and an amplitude (Vpp) of 332 mV. The frequency of the produced signal could be adjusted by changing the clock frequency of the "clk" signal. This could be done by adjusting the "count" value (line 72 of the code in Appendix F).

4.2.2 Production of a square signal

As the same program used in the production of the ramp signal was used in this exercise, the same result was obtained when simulated from ISIM.

The waveform obtained by the DSO connected to the R-2R ladder when the outputs FPGA board were connected to the R-2R ladder was as follows.

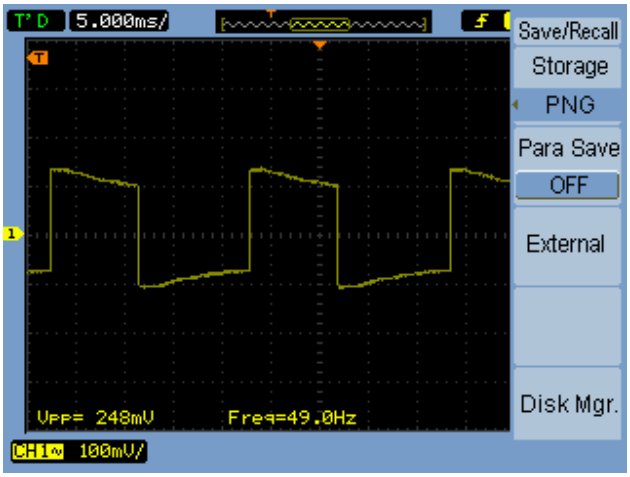
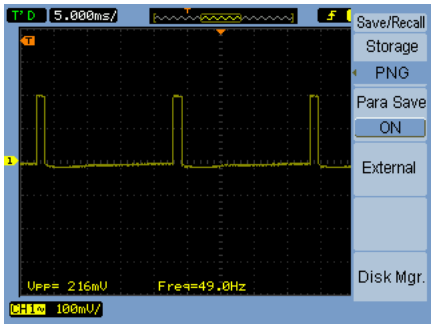
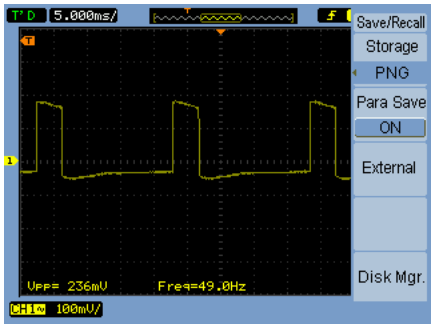


Figure 4.19: Waveform of the square signal formed

When the switches from SW4 to SW7 were adjusted, the duty cycle of the produced square signal could be adjusted. SW4 represented the LSB and SW7 represented the MSB. Values from 0000 to 1111 could be created from the 4 switches and analog voltages corresponding to the created values were created from the 4 bit R-2R ladder. The output of this 4 bit R-2R ladder was connected to as the reference voltage of the comparator.



(a) 0001 provided



(b) 0010 provided

(c) 0100 provided

(d) 1000 provided

Figure 4.20: Waveforms obtained when the switches were used

Although a square wave was created, it didn't appear to be exactly square shaped. This is due to the fact that the frequency of the counter used was not high enough. This could be averted by decreasing the "count" value from 1000 to a lower value (line 55 of the code in Appendix G).

4.2.3 Production of a sinusoidal signal

The program when simulated from ISIM showed the following behaviour.

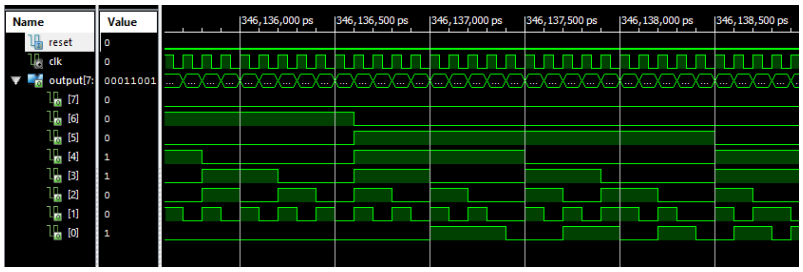


Figure 4.21: The output when synthesized through the FPGA

The waveform obtained by the DSO connected to the R-2R ladder when the outputs FPGA board were connected to the R-2R ladder was as follows.

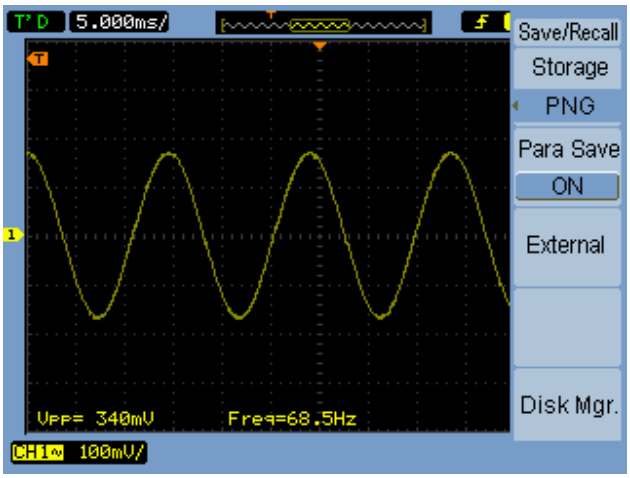


Figure 4.22: Waveform of the sinusoidal signal formed

The frequency of the sinusoidal signal produced had a frequency of 68.5 Hz and an amplitude (Vpp) of 340 mV. The frequency of the produced signal could be adjusted by changing the clock frequency of the "clk" signal. This could be done by adjusting the "count" value (line 55 of the code in Appendix H).

5 Discussion

All through out this practical, the FPGA was programmed as a RAM. So, the program was only temporarily saved in the FPGA and executed. Once the power to the FPGA is removed, the FPGA clears its programming memory. If the FPGA is to be used in an application where the use of a ROM is needed, one could program the device permanently. This could be achieved by selecting the PROM option in the Adept software instead of FPGA.

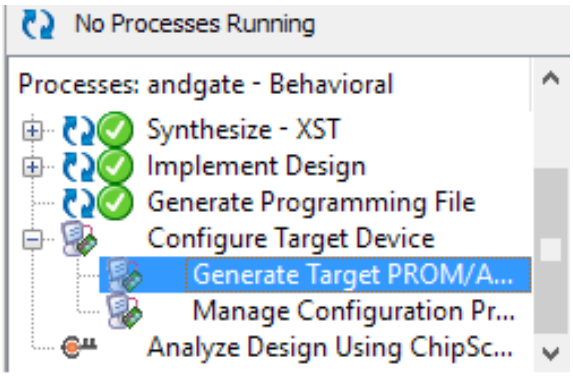


Figure 5.1: Generation of the PROM file

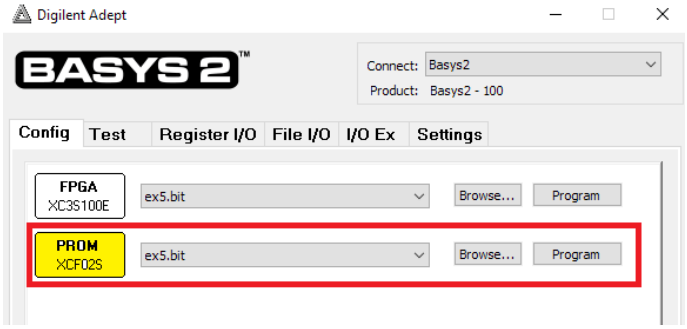


Figure 5.2: Changing from FPGA to PROM

A very useful feature of the Linux ISE is the ability to view the schematic of a program. There are 2 types of schematics.

1. RTL schematic - To view a schematic representation of the pre-optimized design in terms of generic symbols like logic gates, adders, multipliers and counters.
2. Technology schematic - To view a schematic representation of the design in terms of logic elements optimized for the target device (technology).

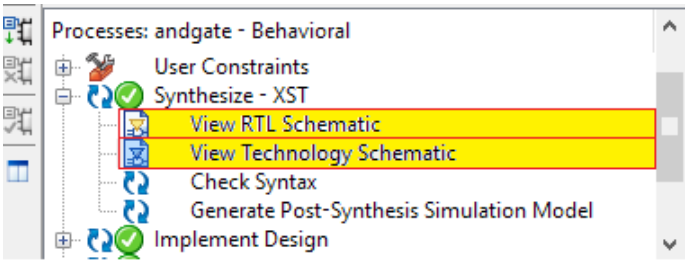
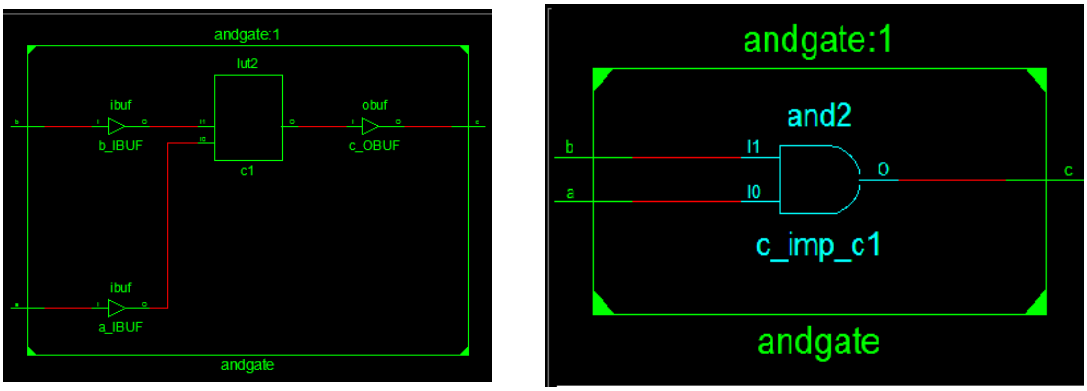


Figure 5.3: Selecting the schematic view



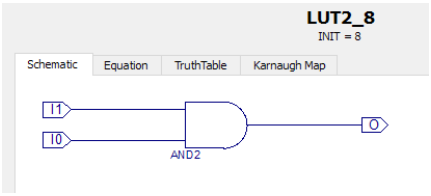
(a) Technology schematic

(b) RTL schematic

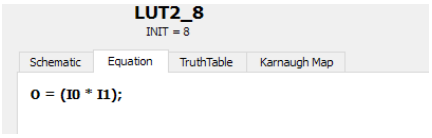
Figure 5.4: The schematics of an AND gate

By double clicking on the instance handling the logic gate obtained by a technology schematic, the following details of the selected instance can be obtained (instance dialog).

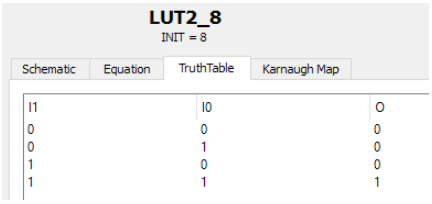
- Schematic.
- Equation.
- Truth table.
- K-map.



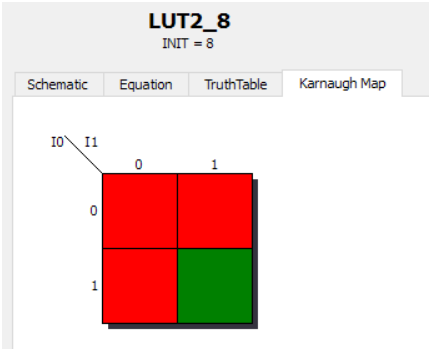
(a) Schematic



(b) Truth table



(c) 0100 provided



(d) K-map

Figure 5.5: Instance dialog

In applications where very high speed information transfers are involved, timing delays play a major role in the accuracy and precision of the outputs. Hence, calculating signal delays is of utmost importance. By using the static timing option in the design overview, signal delay information can be obtained.

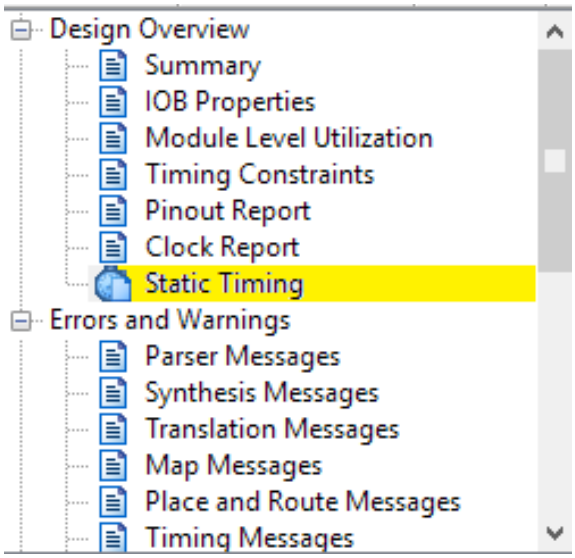


Figure 5.6: Selection of static timing

```
All values displayed in nanoseconds (ns)

Pad to Pad
---------------+---------------+---------+
Source Pad     |Destination Pad|  Delay  |
---------------+---------------+---------+
a              |c              |    6.778|
b              |c              |    6.276|
---------------+---------------+---------+
```

Figure 5.7: Delay information of an AND gate where a and b are inputs and c is the output

The Basys2 FPGA board contains a primary, user-settable silicon oscillator that produces 25MHz, 50MHz, or 100MHz based on the position of the clock select jumper at JP4. The default frequency is 50 MHz. The primary oscillator is a silicon oscillator whose pros are its flexibility and cheapness. Like any device, it has its own cons as well. In the case of a silicon oscillator, that being the frequency instability. So in applications where frequency stability is needed for example, driving of a VGA monitor, using a crystal oscillator installed in the IC6 socket gives a much more defined and an accurate output.

6 Conclusion

In this practical, hardware synthesis using VHDL was studied. Theoretical and practical knowledge needed in using the Basys2 FPGA board were acquired through this practical. Also, the capabilities and the practical limitations of the FPGA board were practically understood. So, all in all, it can be concluded that through this practical, all the basic knowledge needed in the use of VHDL in the programming of a FPGA board in any needed application was achieved successfully.

7 References

[1] V. Pedroni, *Circuit design and simulation with VHDL.*,2017. Cambridge [EE. UU.]: The MIT Press, 2010.

[2] "VHDL Tutorial: Learn by Example", *Esd.cs.ucr.edu,* 2017. [Online]. Available: `http://esd.cs.ucr.edu/labs/tutorial/.` [Accessed: 01- Aug- 2017].

[3] "Basys 2 Reference Manual [Reference.Digilentinc]", *Reference.digilentinc.com,* 2017. [Online]. Available: `https://reference.digilentinc.com/reference/programmable-logic/basys-2/reference-manual.` [Accessed: 05- Aug- 2017].

[4] "VHDL for FPGA Design/4-Bit BCD Counter with Clock Enable - Wikibooks, open books for an open world", *En.wikibooks.org,* 2017. [Online]. Available: `https://en.wikibooks.org/wiki/VHDL_for_FPGA_Design/4-Bit_BCD_Counter_with_Clock_Enable`. [Accessed: 07- Aug- 2017].

[5] "Counters", *Csit-sun.pub.ro,* 2017. [Online]. Available: `http://www.csit-sun.pub.ro/courses/Masterat/Xilinx%20Synthesis%20Technology/toolbox.xilinx.com/docsan/xilinx4/data/docs/xst/hdlcode6.html.` [Accessed: 07- Aug- 2017].

[6] J. Deschamps, G. Sutter and E. Cantó, *Guide to FPGA Implementation of Arithmetic Functions.*, Dordrecht: Springer Netherlands, 2012.

Appendices

Appendix A - Code and the UCF file for exercise 1 (AND gate)

Code

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5
6  entity andgate is
7    port ( a, b : in  std_logic;
8         c    : out std_logic
9         );
10 end andgate;
11
12 architecture Behavioral of andgate is
13
14 begin
15  c <= a and b;
16
17 end Behavioral;
```

UCF file

```
1 net "a" loc = "K3";
2 net "b" loc = "L3";
3 net "c" loc = "M5";
```

Appendix B - Code and the UCF file for exercise 2 (BCD to SSD decoder)

Code

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

entity ex2 is
port (
    choose : in std_logic_vector(3 downto 0);
    bcd : in std_logic_vector(3 downto 0);
    anode : out std_logic_vector(3 downto 0);
    ssd : out std_logic_vector(6 downto 0)

);
end ex2;

architecture Behavioral of ex2 is
begin
  process(choose)
  begin
    anode<= not choose;
  end process;

  process(bcd)
  begin
    case bcd is
      when "0000"=> ssd <="0000001";  -- '0'
      when "0001"=> ssd <="1001111";  -- '1'
      when "0010"=> ssd <="0010010";  -- '2'
      when "0011"=> ssd <="0000110";  -- '3'
      when "0100"=> ssd <="1001100";  -- '4'
      when "0101"=> ssd <="0100100";  -- '5'
      when "0110"=> ssd <="0100000";  -- '6'
      when "0111"=> ssd <="0001111";  -- '7'
      when "1000"=> ssd <="0000000";  -- '8'
      when "1001"=> ssd <="0000100";  -- '9'
      when others=> ssd <="1111111";
    end case;
  end process;

end Behavioral;
```

UCF file

```
net "bcd(0)" loc = "P11";
net "bcd(1)" loc = "L3";
net "bcd(2)" loc = "K3";
net "bcd(3)" loc = "B4";

net "ssd(0)" loc = "M12";
net "ssd(1)" loc = "L13";
net "ssd(2)" loc = "P12";
net "ssd(3)" loc = "N11";
net "ssd(4)" loc = "N14";
net "ssd(5)" loc = "H12";
net "ssd(6)" loc = "L14";


net "choose(0)" loc = "G3";
net "choose(1)" loc = "F3";
net "choose(2)" loc = "E2";
net "choose(3)" loc = "N3";

net "anode(0)" loc = "F12";
net "anode(1)" loc = "J12";
net "anode(2)" loc = "M13";
net "anode(3)" loc = "K14";
```

Appendix C - Code and the UCF file for exercise 3 (BCD to SSD decoder with test button)

Code

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

entity ex3 is
port (
    choose : in std_logic_vector(3 downto 0);
    bcd : in std_logic_vector(3 downto 0);
    anode : out std_logic_vector(3 downto 0);
    reset : in std_logic;
    ssd : out std_logic_vector(6 downto 0)

);
end ex3;

architecture Behavioral of ex3 is
begin
  process(reset,choose,bcd)
  begin
    if reset='1' then
      anode <= "0000";
      ssd <="0000000";
    else
      anode<= not choose;

      case bcd is
      when "0000"=> ssd <="0000001";  -- '0'
      when "0001"=> ssd <="1001111";  -- '1'
      when "0010"=> ssd <="0010010";  -- '2'
      when "0011"=> ssd <="0000110";  -- '3'
      when "0100"=> ssd <="1001100";  -- '4'
      when "0101"=> ssd <="0100100";  -- '5'
      when "0110"=> ssd <="0100000";  -- '6'
      when "0111"=> ssd <="0001111";  -- '7'
      when "1000"=> ssd <="0000000";  -- '8'
      when "1001"=> ssd <="0000100";  -- '9'
      when others=> ssd <="1111111";
      end case;
    end if;
  end process;
end Behavioral;
```

UCF file

```
net "bcd(0)" loc = "P11";
net "bcd(1)" loc = "L3";
net "bcd(2)" loc = "K3";
net "bcd(3)" loc = "B4";

net "ssd(0)" loc = "M12";
net "ssd(1)" loc = "L13";
net "ssd(2)" loc = "P12";
net "ssd(3)" loc = "N11";
net "ssd(4)" loc = "N14";
net "ssd(5)" loc = "H12";
net "ssd(6)" loc = "L14";


net "choose(0)" loc = "G3";
net "choose(1)" loc = "F3";
net "choose(2)" loc = "E2";
net "choose(3)" loc = "N3";

net "anode(0)" loc = "F12";
net "anode(1)" loc = "J12";
net "anode(2)" loc = "M13";
net "anode(3)" loc = "K14";
```

```
24
25 net "reset" loc= "A7";
```

Appendix D - Code and the UCF file for exercise 4 (Mod 10 counter)

Code

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

entity test4 is
   port(
   clk1: in std_logic;
   Reset: in std_logic;
   Output: out std_logic_vector(3 downto 0)
   );
end test4;

architecture Behavioral of test4 is
signal temp: std_logic_vector(3 downto 0);
signal count : integer :=1;
signal clk : std_logic :='0';
begin

process(clk1)
begin
  if rising_edge(clk1) then
  count <=count+1;
  if(count = 50000000) then
  clk <= not clk;
  count <=1;
  end if;
  end if;
end process;

process(clk,Reset,temp)
   begin
      if Reset='1' then
         temp <= "0000";
      elsif rising_edge(clk) then
      temp <= temp + '1';
         if temp="1001" then
               temp<="0000";
      end if;
    end if;
   end process;
   Output <= temp;
end Behavioral;
```

UCF file

```
net "Output(0)" loc = "M5";
net "Output(1)" loc = "M11";
net "Output(2)" loc = "P7";
net "Output(3)" loc = "P6";

net "Reset" loc = "A7";

net "clk1" loc = "B8";
```

Appendix E - Code and the UCF file for exercise 5 (Mod 10 counter with BCD to SSD decoder)

Code

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

entity ex5 is
port(
    clk1: in std_logic;
    reset: in std_logic;
    choose: in std_logic_vector(3 downto 0);
    anode: out std_logic_vector(3 downto 0);
    ssd : out std_logic_vector(6 downto 0)
   );
end ex5;

architecture Behavioral of ex5 is
signal temp:std_logic_vector(3 downto 0);
signal count: integer :=1;
signal clk: std_logic :='0';
begin

  process(clk1)
  begin
  if rising_edge(clk1) then
    count <= count + 1;
  if (count= 100000000) then
    clk <= not clk;
    count <= 1;
  end if;
  end if;
  end process;

  process(clk,reset,temp)
  begin
    if reset= '1' then
      temp <= "0000";
    elsif rising_edge(clk) then
      temp <= temp + '1';
      if temp= "1001" then
        temp<= "0000";
      end if;
    end if;
  end process;

  process(choose)
  begin
    anode <= not choose;
--    case (choose) is
--      when "0001" => anode <= "1110";
--      when "0010" => anode <= "1101";
--      when "0100" => anode <= "1011";
--      when "1000" => anode <= "0111";
--      when others => anode <= "1111";
--    end case;
  end process;

  process(temp)
  begin
    case temp is
      when "0000"=> ssd <="0000001";  -- '0'
      when "0001"=> ssd <="1001111";  -- '1'
      when "0010"=> ssd <="0010010";  -- '2'
      when "0011"=> ssd <="0000110";  -- '3'
      when "0100"=> ssd <="1001100";  -- '4'
      when "0101"=> ssd <="0100100";  -- '5'
      when "0110"=> ssd <="0100000";  -- '6'
```

```
67        when "0111"=> ssd <="0001111";  -- '7'
68        when "1000"=> ssd <="0000000";  -- '8'
69        when "1001"=> ssd <="0000100";  -- '9'
70        when others=> ssd <="1111111";
71        end case;
72    end process;
73
74 end Behavioral;
```

UCF file

```
1  net "ssd(0)" loc = "M12";
2  net "ssd(1)" loc = "L13";
3  net "ssd(2)" loc = "P12";
4  net "ssd(3)" loc = "N11";
5  net "ssd(4)" loc = "N14";
6  net "ssd(5)" loc = "H12";
7  net "ssd(6)" loc = "L14";
8
9  net "reset" loc = "A7";
10
11 net "clk1" loc = "B8";
12
13 net "choose(0)" loc = "G3";
14 net "choose(1)" loc = "F3";
15 net "choose(2)" loc = "E2";
16 net "choose(3)" loc = "N3";
17
18 net "anode(0)" loc = "F12";
19 net "anode(1)" loc = "J12";
20 net "anode(2)" loc = "M13";
21 net "anode(3)" loc = "K14";
```

Appendix F - Code and the UCF file for exercise 6 (Ramp signal generation)

Code

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

entity dds_ex1 is
port(
    clk1: in std_logic;
    reset:in std_logic;
    choose_up: in std_logic;
    choose_down: in std_logic;
--    freq: in std_logic_vector(3 downto 0);
     led: out std_logic_vector(7 downto 0);
    output: out std_logic_vector(7 downto 0)
);
end dds_ex1;

architecture Behavioral of dds_ex1 is
signal temp:std_logic_vector(7 downto 0);
signal count: integer :=1;
--signal frequency: integer :=1000000;
signal clk: std_logic :='0';
signal change: std_logic :='0';
begin

--  process(clk1,freq)
--  begin
--  if rising_edge(clk1) then
--    case freq is
--      when "0001"=> frequency <=1000000;
--      when "0010"=> frequency <=10000;
--      when "0100"=> frequency <=100;
--      when "1000"=> frequency <=1;
--      when others=> frequency <=50000000;
--    end case;
--  end if;
--  end process;

  process(clk1)
  begin
    if rising_edge(clk1) then
        count <= count + 1;
--        frequency<=10000000;
      if (count= 1000) then
        clk <= not clk;
        count <= 1;
      end if;
    end if;
   end process;

  process(clk,reset,temp,choose_up,choose_down,change)
  begin
    if (reset= '1') then
      temp<= (others=> '0');
    elsif rising_edge(clk) then
      if choose_up='1' and choose_down= '0' then
        temp<= temp + '1';
        if temp="11111111" then
          temp<="00000000";
        end if;
      elsif choose_up='0' and choose_down= '1' then
        temp<= temp - '1';
        if temp="00000000" then
          temp<="11111111";
        end if;
      elsif choose_up='1' and choose_down= '1' then
```

```
        if change='0' then
          if temp="11111111" then
            change<='1';
          else
            temp<= temp + '1';
          end if;
        elsif change='1' then
            if temp="00000000" then
              change<='0';
            else
              temp<= temp - '1';
            end if;
        end if;
      end if;
    end if;
  end process;
output<=temp; --change the outputs in the ucf file from the LED to the JA and JB
    ports when connecting the R-2R ladder
led<=temp;
end Behavioral;
```

UCF file

```
net "reset" loc = "A7";

net "clk1" loc = "B8";

net "output(0)" loc = "M5";
net "output(1)" loc = "M11";
net "output(2)" loc = "P7";
net "output(3)" loc = "P6";
net "output(4)" loc = "N5";
net "output(5)" loc = "N4";
net "output(6)" loc = "P4";
net "output(7)" loc = "G1";

net "led(0)" loc = "B7";
net "led(1)" loc = "C5";
net "led(2)" loc = "B6";
net "led(3)" loc = "C6";
net "led(4)" loc = "B5";
net "led(5)" loc = "J3";
net "led(6)" loc = "A3";
net "led(7)" loc = "B2";

net "choose_up" loc = "P11";
net "choose_down" loc = "L3";

#net "freq(0)" loc = "G3";
#net "freq(1)" loc = "F3";
#net "freq(2)" loc = "E2";
#net "freq(3)" loc = "N3";
```

Appendix G - Code and the UCF file for exercise 7 (Square signal generation)

Code

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

entity dds_ex2 is
port(
    bcd : in std_logic_vector(3 downto 0);
    clk1: in std_logic;
    reset:in std_logic;
    choose_up: in std_logic;
    choose_down: in std_logic;
    output: out std_logic_vector(7 downto 0);
    comp : out std_logic_vector(3 downto 0) -- comparator out
);
end dds_ex2;

architecture Behavioral of dds_ex2 is
signal temp:std_logic_vector(7 downto 0);
signal count: integer :=1;
signal clk: std_logic :='0';
signal change: std_logic :='0';
begin
  process(clk1)
  begin
  if rising_edge(clk1) then
    count <= count + 1;
  if (count= 250) then
    clk <= not clk;
    count <= 1;
  end if;
  end if;
  end process;

  process(clk,reset,temp,choose_up,choose_down,change)
  begin
    if (reset= '1') then
      temp<= (others=> '0');
    elsif rising_edge(clk) then
      if choose_up='1' and choose_down= '0' then
        temp<= temp + '1';
        if temp="11111111" then
          temp<="00000000";
        end if;
      elsif choose_up='0' and choose_down= '1' then
        temp<= temp - '1';
        if temp="00000000" then
          temp<="11111111";
        end if;
      elsif choose_up='1' and choose_down= '1' then
        if change='0' then
          if temp="11111111" then
            change<='1';
          else
            temp<= temp + '1';
          end if;
        elsif change='1' then
            if temp="00000000" then
              change<='0';
            else
              temp<= temp - '1';
            end if;
        end if;
      end if;
    end if;
  end process;
```

```
67
68    process (bcd)
69    begin
70      case bcd is
71        when "0000"=> comp <="0000";  -- '0'
72        when "0001"=> comp <="0001";  -- '1'
73        when "0010"=> comp <="0010";  -- '2'
74        when "0011"=> comp <="0011";  -- '3'
75        when "0100"=> comp <="0100";  -- '4'
76        when "0101"=> comp <="0101";  -- '5'
77        when "0110"=> comp <="0110";  -- '6'
78        when "0111"=> comp <="0111";  -- '7'
79        when "1000"=> comp <="1000";  -- '8'
80        when "1001"=> comp <="1001";  -- '9'
81        when "1010"=> comp <="1010";  -- '10'
82        when "1011"=> comp <="1011";  -- '11'
83        when "1100"=> comp <="1100";  -- '12'
84        when "1101"=> comp <="1101";  -- '13'
85        when "1110"=> comp <="1110";  -- '14'
86        when others=> comp <="1111"; -- '15'
87      end case;
88    end process;
89
90  output<=temp; --change the outputs in the ucf file from the LED to the JA and JB
        ports when connecting the R-2R ladder
91
92  end Behavioral;
```

UCF file

```
1  net "reset" loc = "A7";
2
3  net "clk1" loc = "B8";
4
5  net "output(0)" loc = "B7";
6  net "output(1)" loc = "C5";
7  net "output(2)" loc = "B6";
8  net "output(3)" loc = "C6";
9  net "output(4)" loc = "B5";
10 net "output(5)" loc = "J3";
11 net "output(6)" loc = "A3";
12 net "output(7)" loc = "B2";
13
14 net "choose_up" loc = "P11";
15 net "choose_down" loc = "L3";
16
17 net "bcd(0)" loc = "G3";
18 net "bcd(1)" loc = "F3";
19 net "bcd(2)" loc = "E2";
20 net "bcd(3)" loc = "N3";
21
22 net "comp(0)" loc = "D12";
23 net "comp(1)" loc = "C13";
24 net "comp(2)" loc = "A13";
25 net "comp(3)" loc = "C12";
```

Appendix H - Code and the UCF file for exercise 8 (Sinusoidal signal generation)

Code

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

entity dds_ex3 is
port(
    clk1: in std_logic;
    reset:in std_logic;
     led: out std_logic_vector(7 downto 0);
    output: out std_logic_vector(7 downto 0)
);
end dds_ex3;

architecture Behavioral of dds_ex3 is
signal temp:std_logic_vector(7 downto 0);
signal c_up: std_logic :='1';
signal count: integer :=1;
signal clk: std_logic :='0';
signal val: integer :=1;
signal sin: std_logic_vector(7 downto 0);
begin
  process(clk1)
  begin
    if rising_edge(clk1) then
        count <= count + 1;
      if (count= 1000) then
        clk <= not clk;
        count <= 1;
      end if;
    end if;
   end process;

  process(val)
  begin
    case val is
      when  1 =>  sin <=  "10000000";
      when  2 =>  sin <=  "10000010";
      when  3 =>  sin <=  "10000100";
      when  4 =>  sin <=  "10000110";
      when  5 =>  sin <=  "10001000";
      when  6 =>  sin <=  "10001011";
      when  7 =>  sin <=  "10001101";
      when  8 =>  sin <=  "10001111";
      when  9 =>  sin <=  "10010001";
      when  10  =>  sin <=  "10010011";
      when  11  =>  sin <=  "10010110";
      when  12  =>  sin <=  "10011000";
      when  13  =>  sin <=  "10011010";
      when  14  =>  sin <=  "10011100";
      when  15  =>  sin <=  "10011110";
      when  16  =>  sin <=  "10100000";
      when  17  =>  sin <=  "10100011";
      when  18  =>  sin <=  "10100101";
      when  19  =>  sin <=  "10100111";
      when  20  =>  sin <=  "10101001";
      when  21  =>  sin <=  "10101011";
      when  22  =>  sin <=  "10101101";
      when  23  =>  sin <=  "10101111";
      when  24  =>  sin <=  "10110001";
      when  25  =>  sin <=  "10110011";
      when  26  =>  sin <=  "10110101";
      when  27  =>  sin <=  "10110111";
      when  28  =>  sin <=  "10111001";
      when  29  =>  sin <=  "10111011";
      when  30  =>  sin <=  "10111101";
```

```
67        when  31  =>  sin <=  "10111111";
68        when  32  =>  sin <=  "11000001";
69        when  33  =>  sin <=  "11000011";
70        when  34  =>  sin <=  "11000101";
71        when  35  =>  sin <=  "11000111";
72        when  36  =>  sin <=  "11001000";
73        when  37  =>  sin <=  "11001010";
74        when  38  =>  sin <=  "11001100";
75        when  39  =>  sin <=  "11001110";
76        when  40  =>  sin <=  "11001111";
77        when  41  =>  sin <=  "11010001";
78        when  42  =>  sin <=  "11010011";
79        when  43  =>  sin <=  "11010100";
80        when  44  =>  sin <=  "11010110";
81        when  45  =>  sin <=  "11011000";
82        when  46  =>  sin <=  "11011001";
83        when  47  =>  sin <=  "11011011";
84        when  48  =>  sin <=  "11011100";
85        when  49  =>  sin <=  "11011110";
86        when  50  =>  sin <=  "11011111";
87        when  51  =>  sin <=  "11100001";
88        when  52  =>  sin <=  "11100010";
89        when  53  =>  sin <=  "11100100";
90        when  54  =>  sin <=  "11100101";
91        when  55  =>  sin <=  "11100110";
92        when  56  =>  sin <=  "11101000";
93        when  57  =>  sin <=  "11101001";
94        when  58  =>  sin <=  "11101010";
95        when  59  =>  sin <=  "11101011";
96        when  60  =>  sin <=  "11101100";
97        when  61  =>  sin <=  "11101101";
98        when  62  =>  sin <=  "11101111";
99        when  63  =>  sin <=  "11110000";
100       when  64  =>  sin <=  "11110001";
101       when  65  =>  sin <=  "11110010";
102       when  66  =>  sin <=  "11110011";
103       when  67  =>  sin <=  "11110100";
104       when  68  =>  sin <=  "11110100";
105       when  69  =>  sin <=  "11110101";
106       when  70  =>  sin <=  "11110110";
107       when  71  =>  sin <=  "11110111";
108       when  72  =>  sin <=  "11111000";
109       when  73  =>  sin <=  "11111000";
110       when  74  =>  sin <=  "11111001";
111       when  75  =>  sin <=  "11111010";
112       when  76  =>  sin <=  "11111010";
113       when  77  =>  sin <=  "11111011";
114       when  78  =>  sin <=  "11111011";
115       when  79  =>  sin <=  "11111100";
116       when  80  =>  sin <=  "11111100";
117       when  81  =>  sin <=  "11111101";
118       when  82  =>  sin <=  "11111101";
119       when  83  =>  sin <=  "11111101";
120       when  84  =>  sin <=  "11111110";
121       when  85  =>  sin <=  "11111110";
122       when  86  =>  sin <=  "11111110";
123       when  87  =>  sin <=  "11111110";
124       when  88  =>  sin <=  "11111110";
125       when  89  =>  sin <=  "11111110";
126       when  90  =>  sin <=  "11111110";
127       when  91  =>  sin <=  "11111111";
128       when  92  =>  sin <=  "11111110";
129       when  93  =>  sin <=  "11111110";
130       when  94  =>  sin <=  "11111110";
131       when  95  =>  sin <=  "11111110";
132       when  96  =>  sin <=  "11111110";
133       when  97  =>  sin <=  "11111110";
134       when  98  =>  sin <=  "11111110";
135       when  99  =>  sin <=  "11111010";
136       when  100 =>  sin <=  "11111101";
```

```
137         when   101 =>   sin <=   "11111101";
138         when   102 =>   sin <=   "11111100";
139         when   103 =>   sin <=   "11111100";
140         when   104 =>   sin <=   "11111011";
141         when   105 =>   sin <=   "11111011";
142         when   106 =>   sin <=   "11111010";
143         when   107 =>   sin <=   "11111010";
144         when   108 =>   sin <=   "11111001";
145         when   109 =>   sin <=   "11111000";
146         when   110 =>   sin <=   "11111000";
147         when   111 =>   sin <=   "11110111";
148         when   112 =>   sin <=   "11110110";
149         when   113 =>   sin <=   "11110101";
150         when   114 =>   sin <=   "11110100";
151         when   115 =>   sin <=   "11110100";
152         when   116 =>   sin <=   "11110011";
153         when   117 =>   sin <=   "11110010";
154         when   118 =>   sin <=   "11110001";
155         when   119 =>   sin <=   "11110000";
156         when   120 =>   sin <=   "11101111";
157         when   121 =>   sin <=   "11101101";
158         when   122 =>   sin <=   "11101100";
159         when   123 =>   sin <=   "11101011";
160         when   124 =>   sin <=   "11101010";
161         when   125 =>   sin <=   "11101001";
162         when   126 =>   sin <=   "11101000";
163         when   127 =>   sin <=   "11100110";
164         when   128 =>   sin <=   "11100101";
165         when   129 =>   sin <=   "11100100";
166         when   130 =>   sin <=   "11100010";
167         when   131 =>   sin <=   "11100001";
168         when   132 =>   sin <=   "11011111";
169         when   133 =>   sin <=   "11011110";
170         when   134 =>   sin <=   "11011100";
171         when   135 =>   sin <=   "11011011";
172         when   136 =>   sin <=   "11011001";
173         when   137 =>   sin <=   "11011000";
174         when   138 =>   sin <=   "11010110";
175         when   139 =>   sin <=   "11010100";
176         when   140 =>   sin <=   "11010011";
177         when   141 =>   sin <=   "11010001";
178         when   142 =>   sin <=   "11001111";
179         when   143 =>   sin <=   "11001110";
180         when   144 =>   sin <=   "11001100";
181         when   145 =>   sin <=   "11001010";
182         when   146 =>   sin <=   "11001000";
183         when   147 =>   sin <=   "11000111";
184         when   148 =>   sin <=   "11000101";
185         when   149 =>   sin <=   "11000011";
186         when   150 =>   sin <=   "11000001";
187         when   151 =>   sin <=   "10111111";
188         when   152 =>   sin <=   "10111101";
189         when   153 =>   sin <=   "10111011";
190         when   154 =>   sin <=   "10111001";
191         when   155 =>   sin <=   "10110111";
192         when   156 =>   sin <=   "10110101";
193         when   157 =>   sin <=   "10110011";
194         when   158 =>   sin <=   "10110001";
195         when   159 =>   sin <=   "10101111";
196         when   160 =>   sin <=   "10101101";
197         when   161 =>   sin <=   "10101011";
198         when   162 =>   sin <=   "10101001";
199         when   163 =>   sin <=   "10100111";
200         when   164 =>   sin <=   "10100101";
201         when   165 =>   sin <=   "10100011";
202         when   166 =>   sin <=   "10100000";
203         when   167 =>   sin <=   "10011110";
204         when   168 =>   sin <=   "10011100";
205         when   169 =>   sin <=   "10011010";
206         when   170 =>   sin <=   "10011000";
```

```
      when  171 =>  sin <=  "10010110";
      when  172 =>  sin <=  "10010011";
      when  173 =>  sin <=  "10010001";
      when  174 =>  sin <=  "10001111";
      when  175 =>  sin <=  "10001101";
      when  176 =>  sin <=  "10001011";
      when  177 =>  sin <=  "10001000";
      when  178 =>  sin <=  "10000110";
      when  179 =>  sin <=  "10000100";
      when  180 =>  sin <=  "10000010";
      when  181 =>  sin <=  "10000000";
      when others =>
    end case;
  end process;

  process(clk,reset,val)
  begin
    if (reset= '1') then
      temp<= (others=> '0');
    elsif rising_edge(clk) then
      if  c_up = '1' then
        temp<= sin;
        val <= val + 1;
          if val=181 then
            c_up<= '0';
            val <= 1;
          end if;
      else
        temp<= "11111111"-sin;
        val <= val + 1;
          if val=181 then
            c_up<= '1';
            val <= 1;
          end if;
      end if;
    end if;
  end process;

led <= temp;
output<= temp;

end Behavioral;
```

UCF file

```
net "reset" loc = "A7";

net "clk1" loc = "B8";

net "output(0)" loc = "M5";
net "output(1)" loc = "M11";
net "output(2)" loc = "P7";
net "output(3)" loc = "P6";
net "output(4)" loc = "N5";
net "output(5)" loc = "N4";
net "output(6)" loc = "P4";
net "output(7)" loc = "G1";

net "led(0)" loc = "B7";
net "led(1)" loc = "C5";
net "led(2)" loc = "B6";
net "led(3)" loc = "C6";
net "led(4)" loc = "B5";
net "led(5)" loc = "J3";
net "led(6)" loc = "A3";
net "led(7)" loc = "B2";
```