DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT AND RESEARCH, AKURDI, PUNE-44

Department of Electronics & Telecommunication Engineering

2022-2023

LAB MANUAL

Subject – VLSI DESIGN & TECHNOLOGY

Subject code: 404182

Class – BE

INDEX

Introduction to VHDL

A digital system can be described at different levels of abstractions and from different points of view. As the design process progresses, the level and view are changed, either by human designers or by software tools, It is desirable to have a common framework to exchange information among the designers and various software tools, Hardware description languages (HDLs) serve this purpose. In this chapter we provide an overview of the design, use and capability of HDLs.

Limitations of traditional programming languages:

A programming language is characterized by its syntax and semantics. The syntax comprises the grammatical rules used to write a program, and the semantics is the "meaning" associated with language constructs. When a new computer language is developed, the designers first study the characteristics of the underlying processes and then develop syntactic constructs and their associated semantics to model and express these characteristics.

Most traditional general-purpose programming languages, such as C, are modeled after a sequential process. In this process, operations are performed in sequential order, one operation at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be altered at will. The sequential process model has two major benefits.

• At the abstract level, it helps the human thinking process to develop an algorithm step by step.

• At the implementation level, the sequential process resembles the operation of a basic computer model and thus allows efficient translation from an algorithm to machine instructions. The characteristics of digital hardware, on the other hand, are very different from those of the sequential model. A typical digital system is normally built by smaller parts, with customized wiring that connects the input and output ports of these parts. When signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly. These operations are performed concurrently, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete. After completion, each part updates the value of the corresponding output port. If the value is changed, the output signal will in turn activate all the connected parts and initiate another round of operations. This description shows several unique characteristics of digital systems, including the connections of parts, concurrent operations, and the concept of propagation delay and timing. The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a need for special languages (i.e., HDLs) that are designed to model digital hardware.

VHDL:

VHDL and Verilog are the two most widely used HDLs. Although the syntax and "appearance" of the two

languages are very different, their capabilities and scopes are quite similar. Both are industrial standards and are supported by most software tools.

VHDL stands for VHSIC (Very High Speed Integrated Circuit) HDL. The development of VHDL was sponsored initially by the US Department of Defense as a hardware documentation standard in the early 1980s and then was transferred to the IEEE (Institute of Electrical and Electronics Engineers). IEEE ratified it as IEEE standard 1076 in 1987, which is referred to as VHDL-87. Each IEEE standard is reviewed every few years and is revised as needed, IEEE revised the VHDL standard in 1993, which is referred to as VHDL-93, and made minor modifications and bug fixes in 2001, which is referred to as VHDL-2001. Since no new language construct is added in the new version, there is no significant difference between VHDL-93 and VHDL-2001. A suffix is sometimes added to the IEEE standard to indicate the year the standard was released. For example, VHDL-87 and VHDL-2001 are known as IEEE standards 1076-1987 and IEEE 1076-2001 respectively.

STRUCTURE OF VHDL

```
entity entity_name is
generic ( generic body; );
port ( port declarations ( input, output, inout ); );
end entity entity_name;

architecture architecture_name of entity_name is

signal declaration part;
variable declaration part;
constant declaration part;

begin
process_name : process ( sensitivity_list )
                begin
                        process body;
                end process process_name;

end architecture architecture_name;
```

Description:

VHDL structure has two major bodies. One in **entity** body and the other is **architecture** body. VHDL code can have multiple architectures for a single entity body.

Entity structure supports **generic** and **port** body. In **generic** you can initialize any generic constants with its initial values. In the **port** section you need to declare in input, output or inout ports. VHDL code should always have input and either inout or output ports.

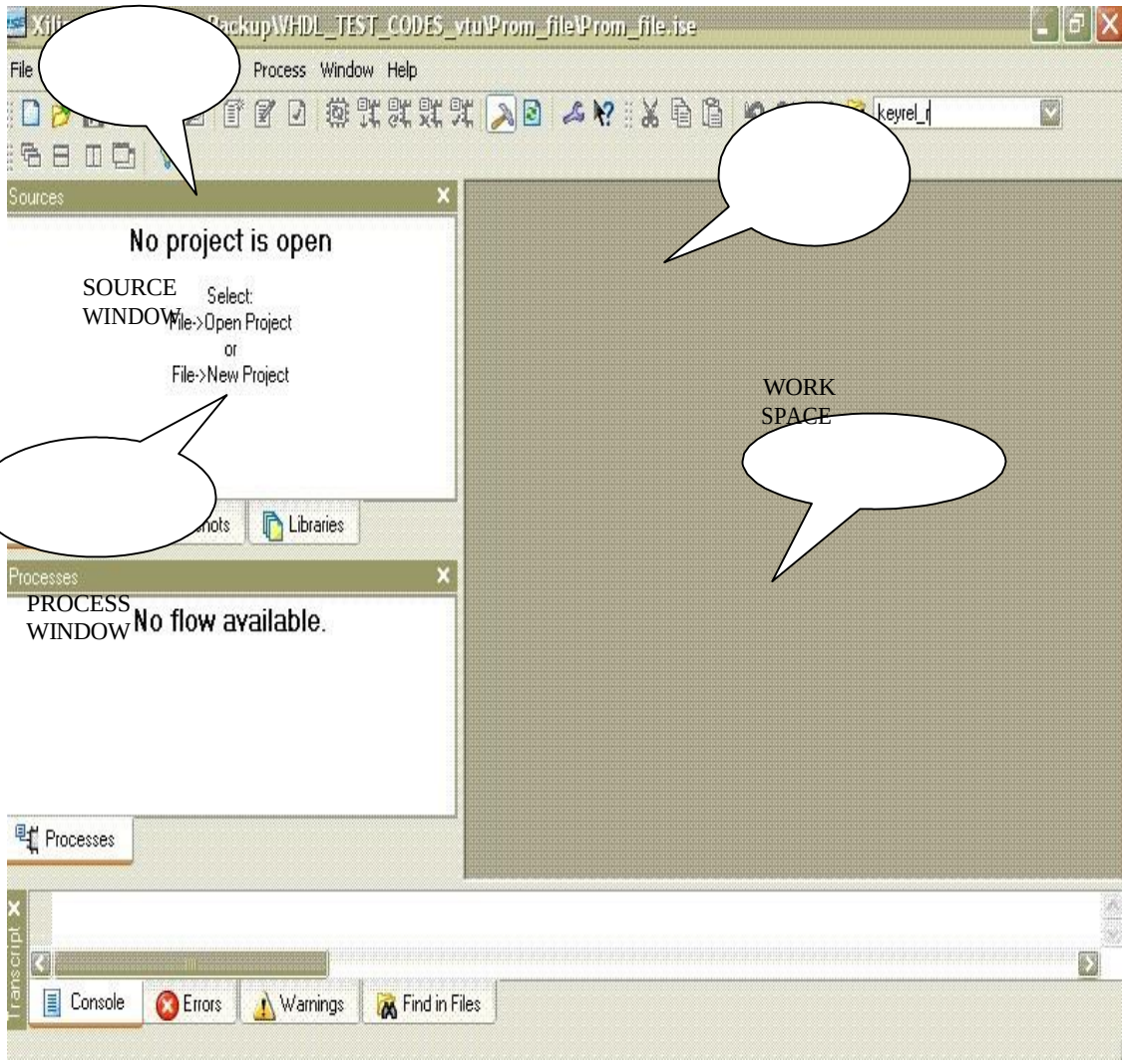
Architecture structure begins with declaration part followed by architecture body.
In the declaration part signals, variables and constants can be declared. These are limited to that architecture body. These cannot be used in another architecture body of the design

Architecture body starts with keyword **begin**, following it can contain **process** part. Process part contains sensitivity list. Whenever any of the values in sensitivity list changes, then only process body will get executed. Again process body can have declaration part (same as that of architecture) which are limited into that specific process. Architecture body also supports for concurrent assignments. The architecture body is closed by **end architecture** arch name.

A single entity can have multiple architecture bodies.

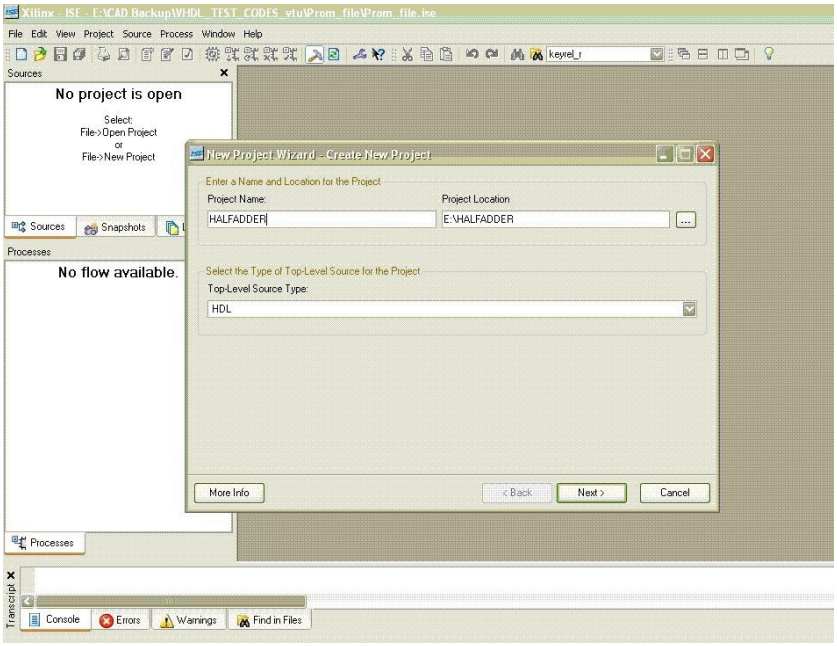
STEPS TO EXECUTE PROGRAM USING XILINX SOFTEWARE

Step 1 : Start the Xilinx Project Navigator by using the desktop shortcut or by using the *Start* ☐ *Programs* ☐ *Xilinx ISE* (14.7).

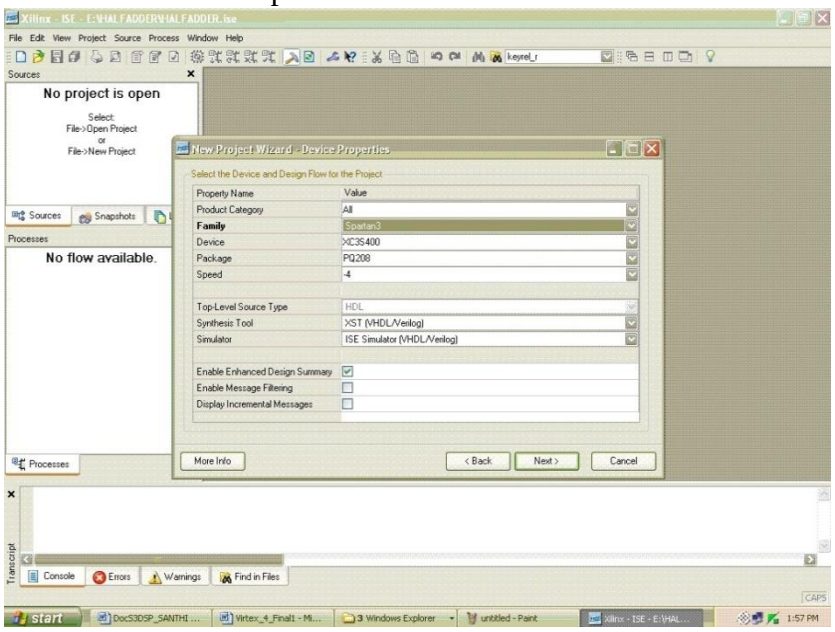


Step 2

Create a new project
In the window go to *FILE* ☐*New project*.
Specify the project name and location and say *NEXT*



Select Device. Use the pull-down arrow to select the Value for each Property Name. Click in the field to access the pull-down list.

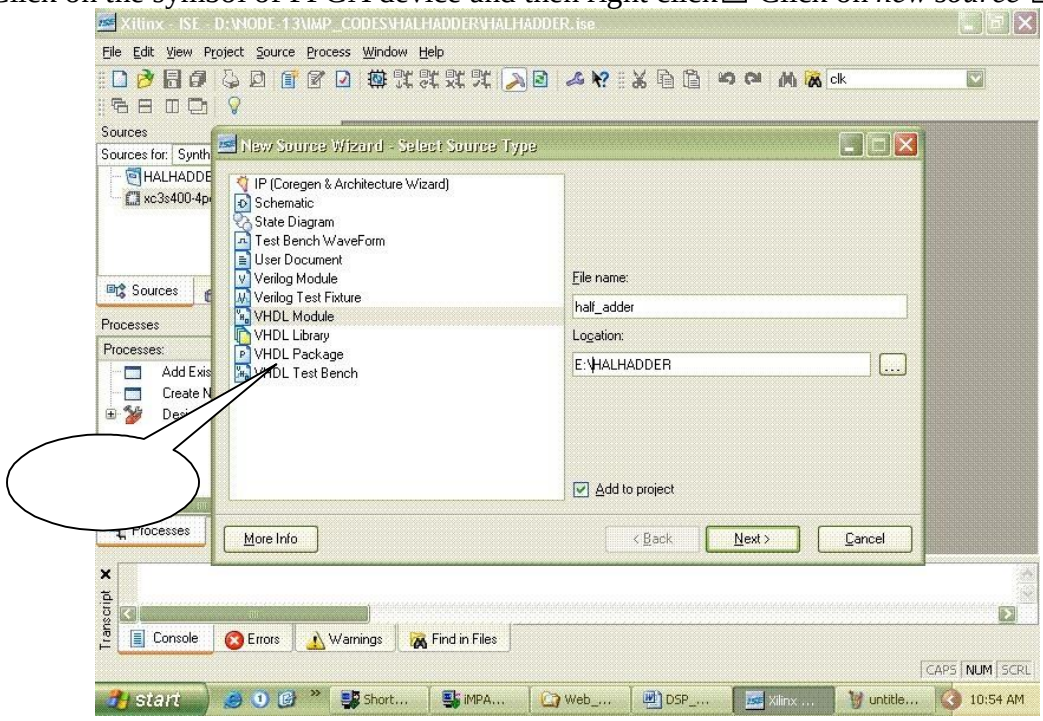


Say *FINISH*. Project summary is seen.

Step 3:

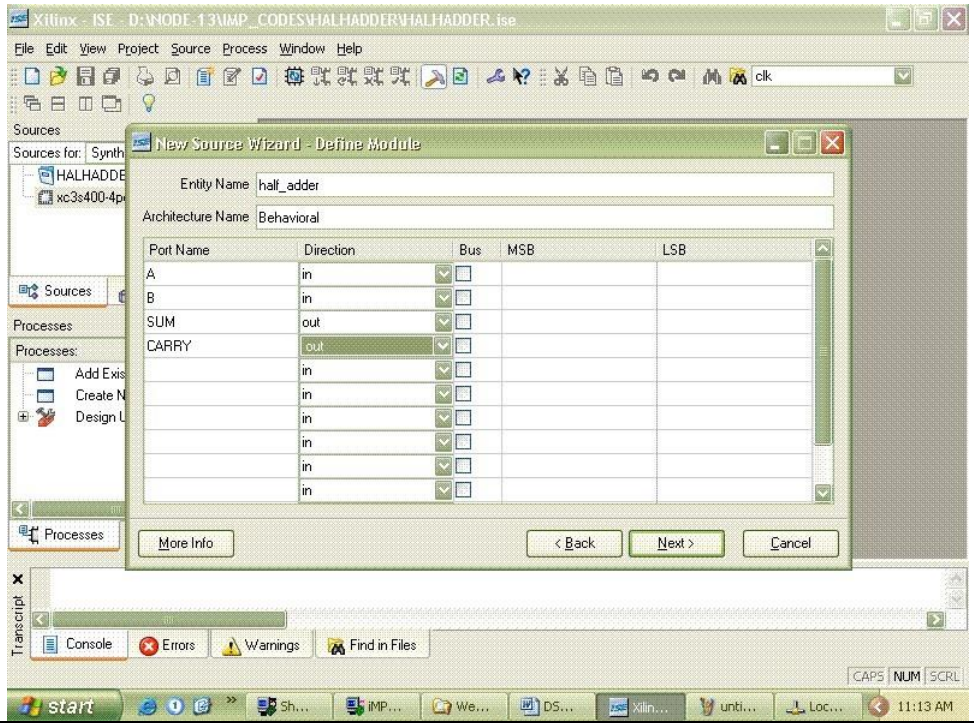
Creating a new VHD file

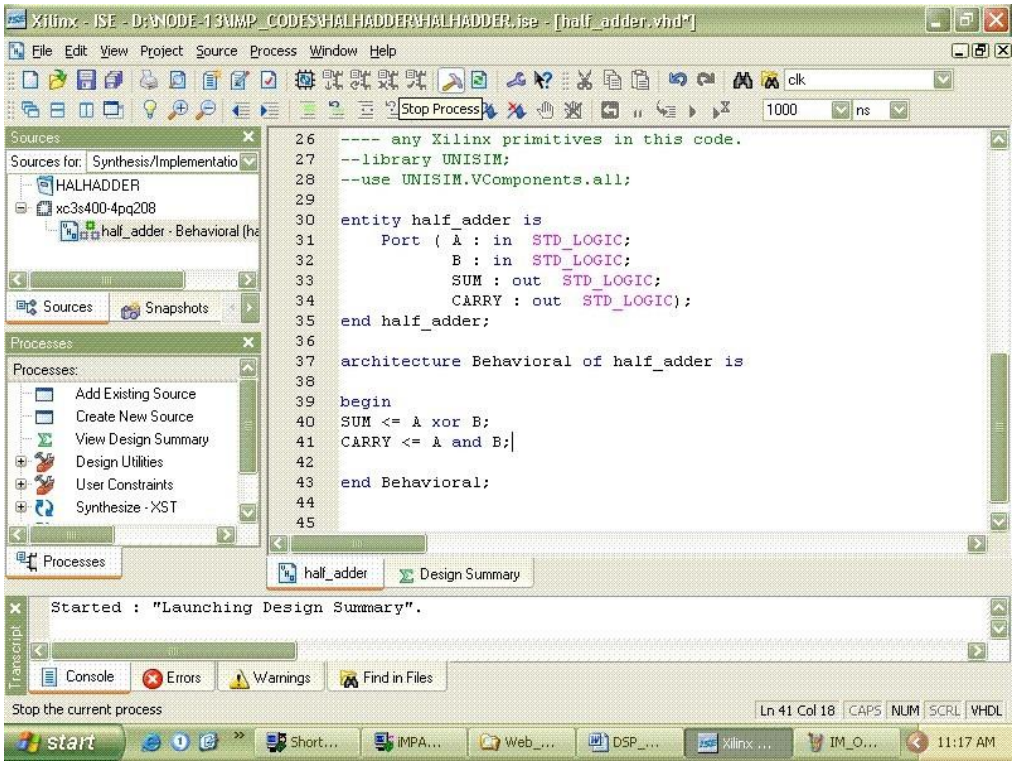
Click on the symbol of FPGA device and then right click☐ Click on *new source* ☐*VHDL*



Then say *Next*☐*Define ports*.In this case

- *a* and *b* are the input ports defined as *in*
- *sum* and *carry* are output ports defined as *out*

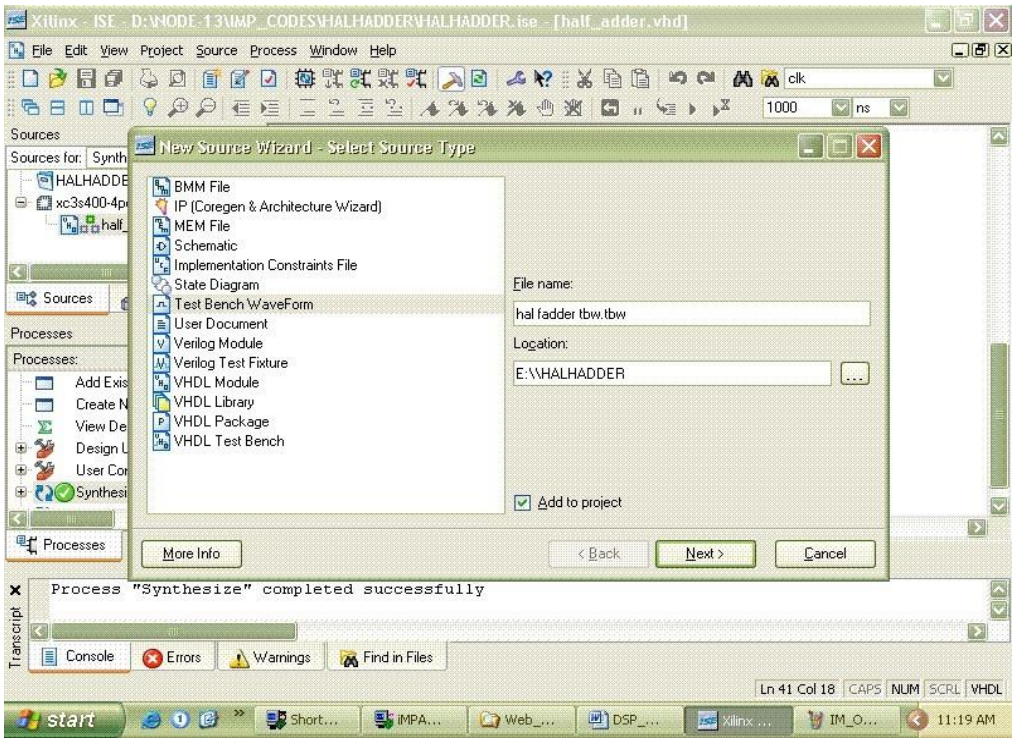


Step 4:	*Writing the Behavioural VHDL Code in VHDL Editor* Sample code is given below for this experiment. 
Step 5	*Check Syntax* Run the *Check syntax* ☐ *Process window*☐ *synthesize*☐*check syntax* >, and remove errors if present.

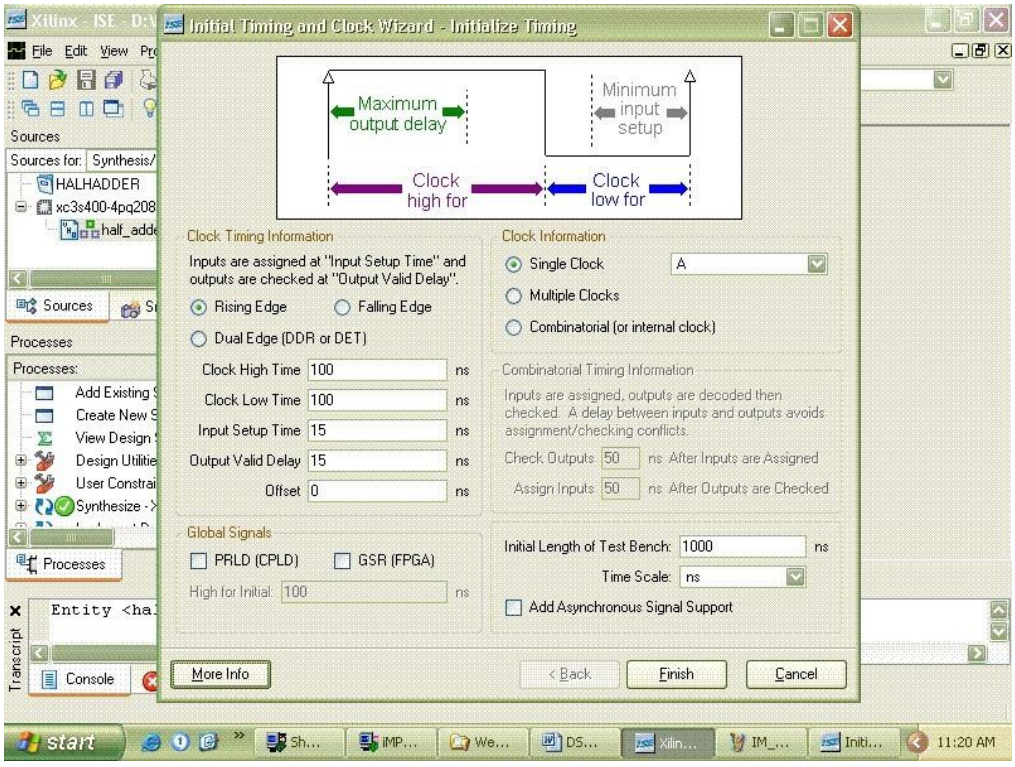
Step 6

Creating a test bench file

Verify the operation of your design before you implement it as hardware. Simulation can be done using ISE simulator. For this click on the symbol of FPGA device and then right click☐ *Click on new source* ☐*Test Bench Waveform and give the name* ☐ *Select entity*☐*Finish.*



Select the desired parameters for simulating your design. In this case *combinational circuit and Simulation time.*



Step 7:

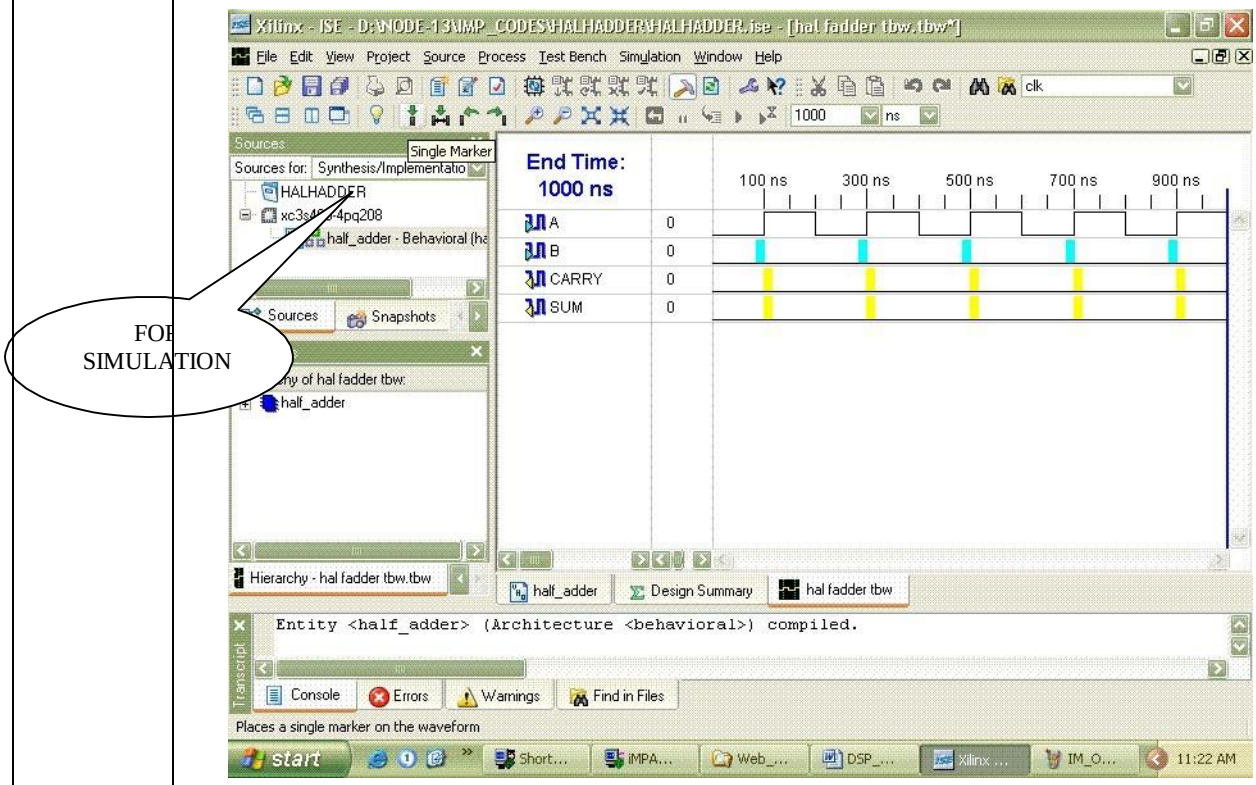
Simulate the code

Simulation Tools

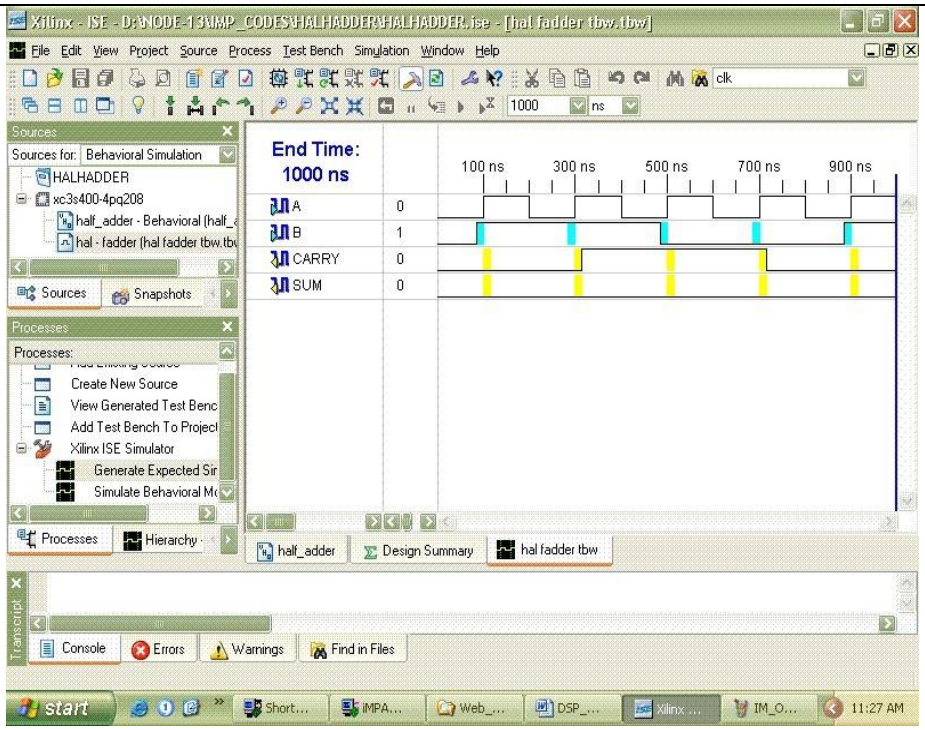
ISE tool supports the following simulation tools:

- *HDL Bencher is an automated test bench creation tool. It is fully integrated with Project Navigator.*
- *ModelSim from Model Technology, Inc., is integrated in Project Navigator to simulate the design at all steps (Functional and Timing). ModelSim XE, the Xilinx Edition of Model Technology, Inc.'s ModelSim application, can be installed from the MTI CD included in your ISE Tool*

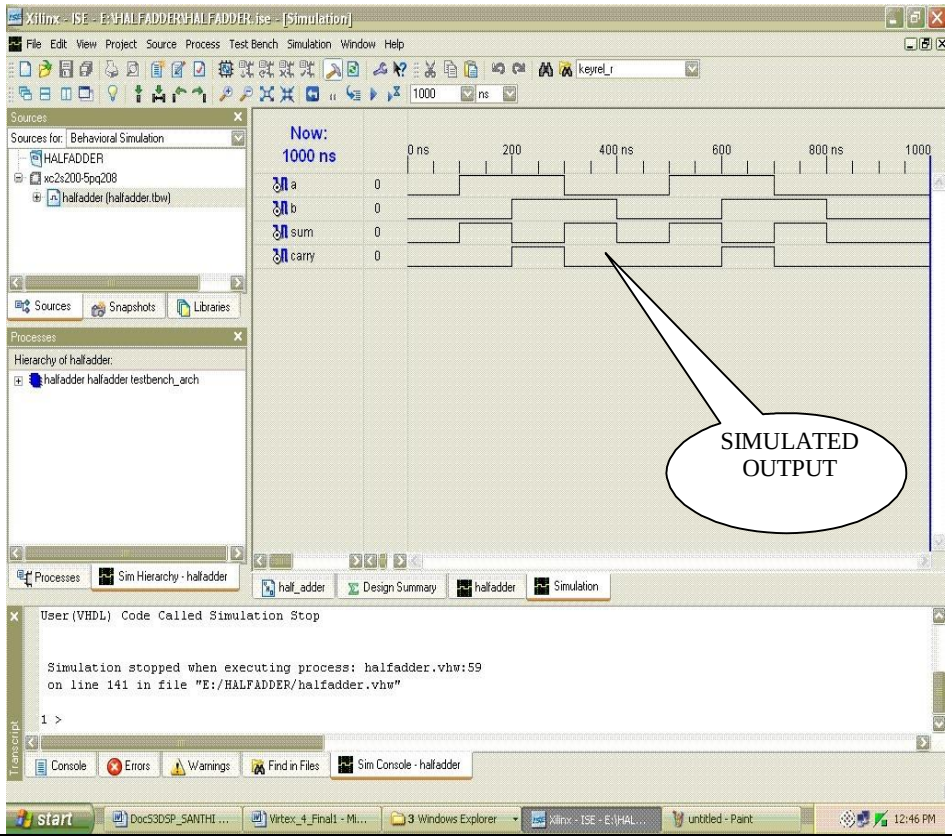
In source Window from the Drop-down menu select *Behavioural Simulation* to view the created test Bench file.



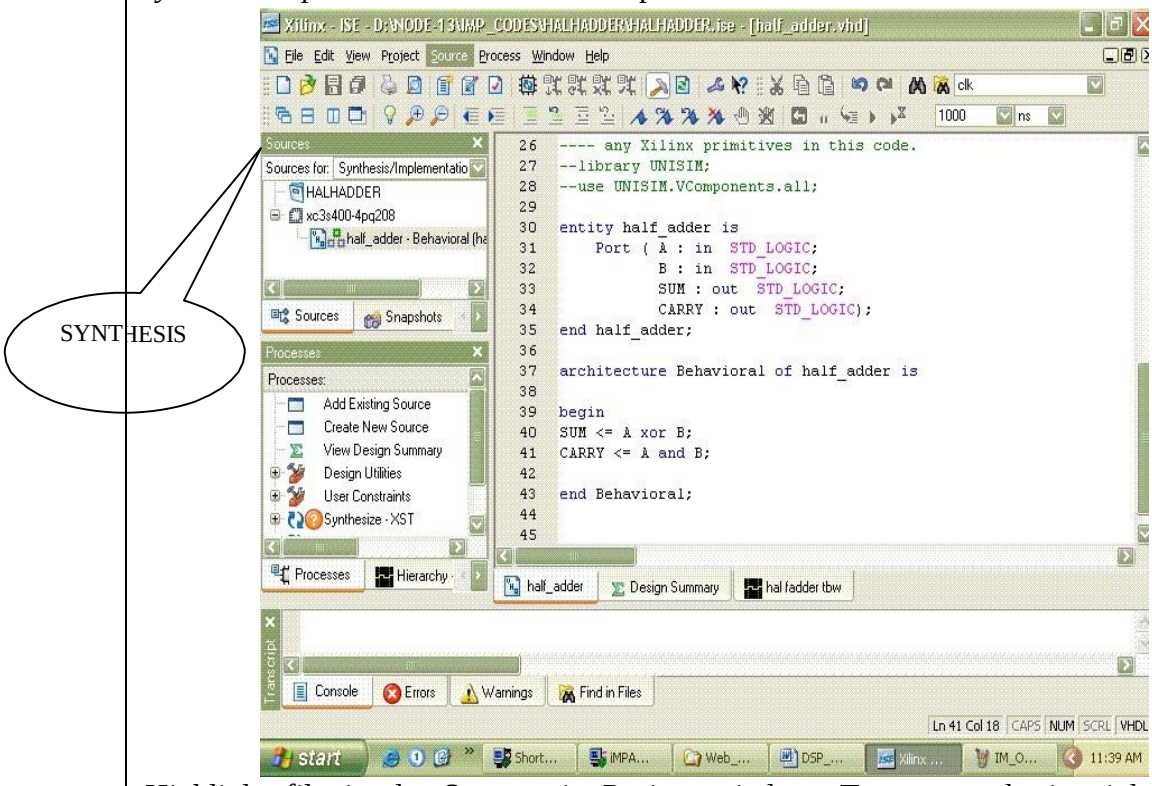
Click on test bench file. Test bench file will open in main window. Assign all the signals and save File. From the source of process window. Click on *Simulate Behavioral Model* in Process window.



Verify your design in wave window by seeing behaviour of output signal with respect to input signal. Close the ISE simulator window



Step 8:	*Synthesize the design using XST*. Translate your design into gates and optimize it for the target architecture. This is the synthesis phase. Again for synthesizing your design, from the source window select, *synthesis/Implementation* from the drop-down menu.

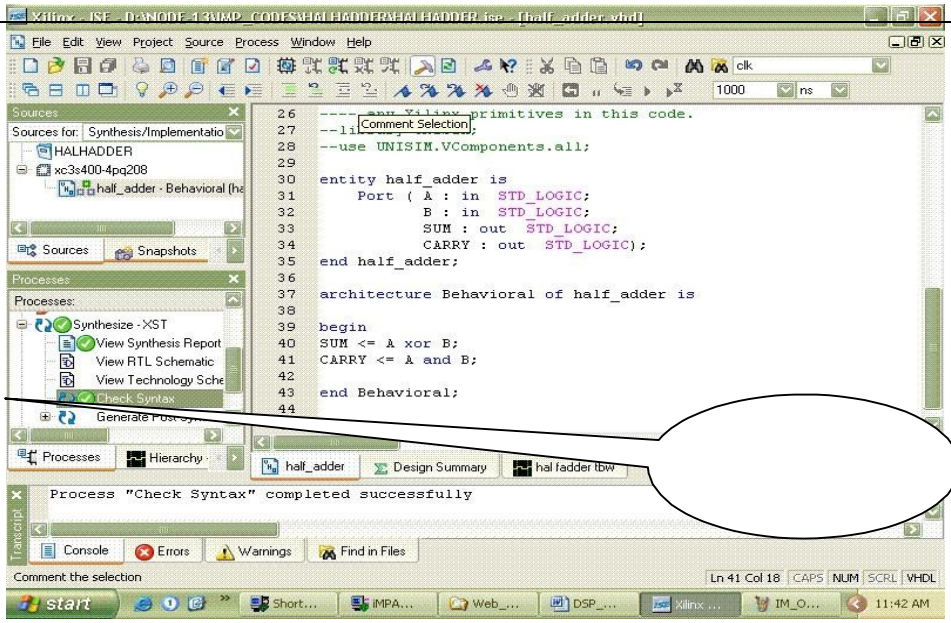


Highlight file in the Sources in Project window. To run synthesis, right-click on Synthesize, and the Run option, *or* double-click on Synthesize in the Processes for Current Source window. Synthesis will run, and

- a green check ☐will appear next to Synthesize when it is successfully completed.
- a red cross ☐indicates an error was generated and
- a yellow exclamation ! mark indicates that a warning was generated, (warnings are OK).

  Check the synthesis report.

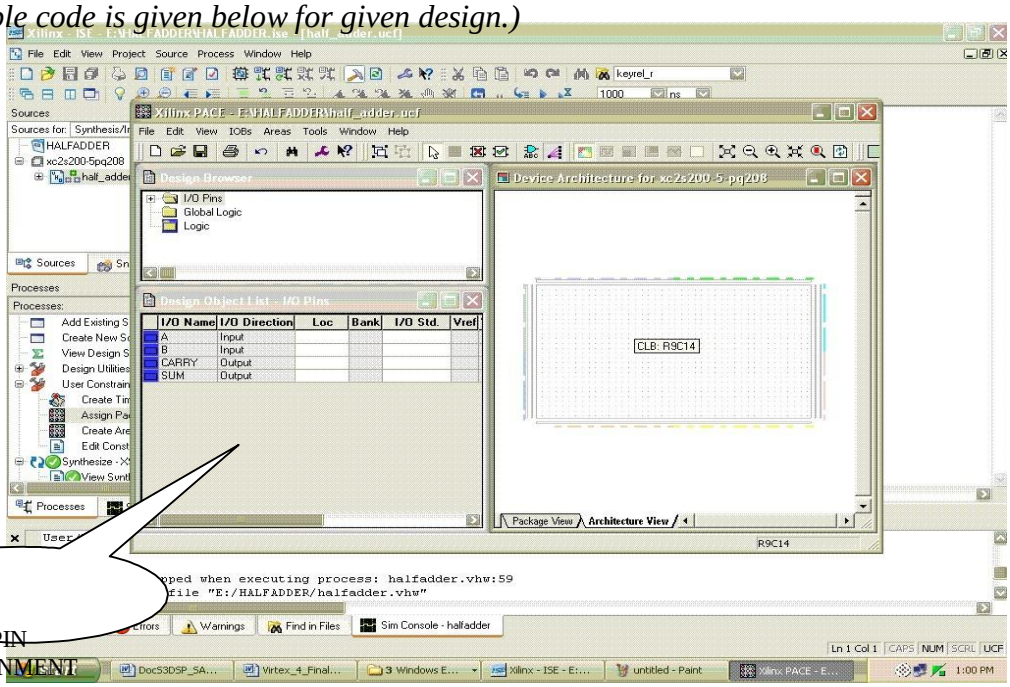
  If there are any errors correct it and rerun synthesis.



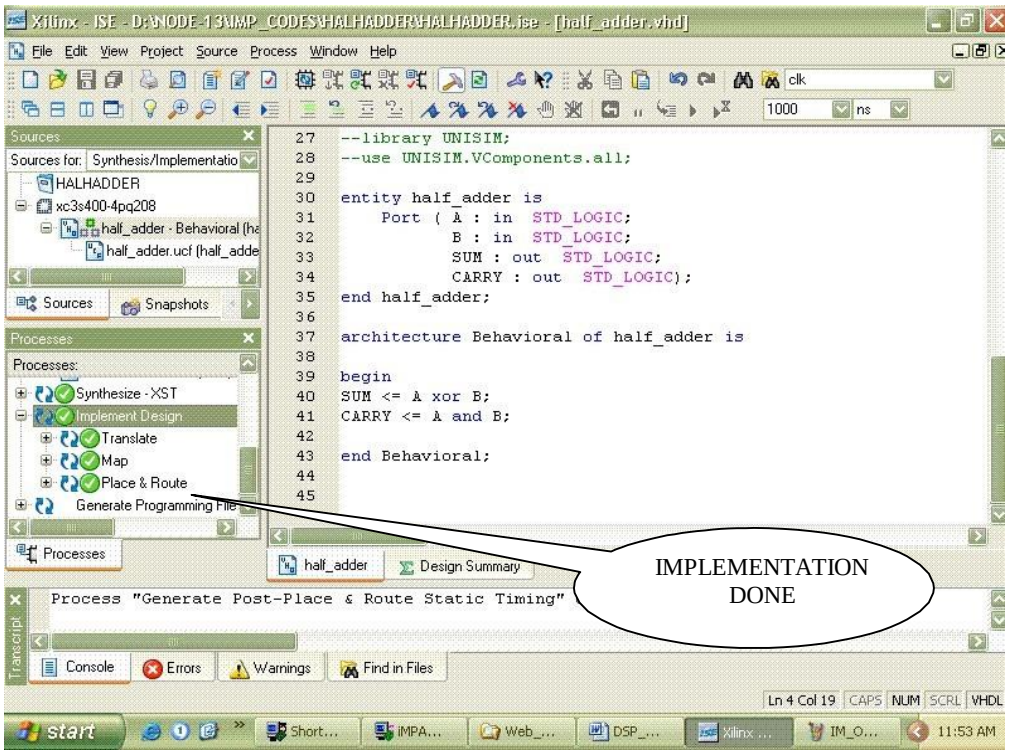
SYNTHESIS COMPLETED SUCCESSFULLY

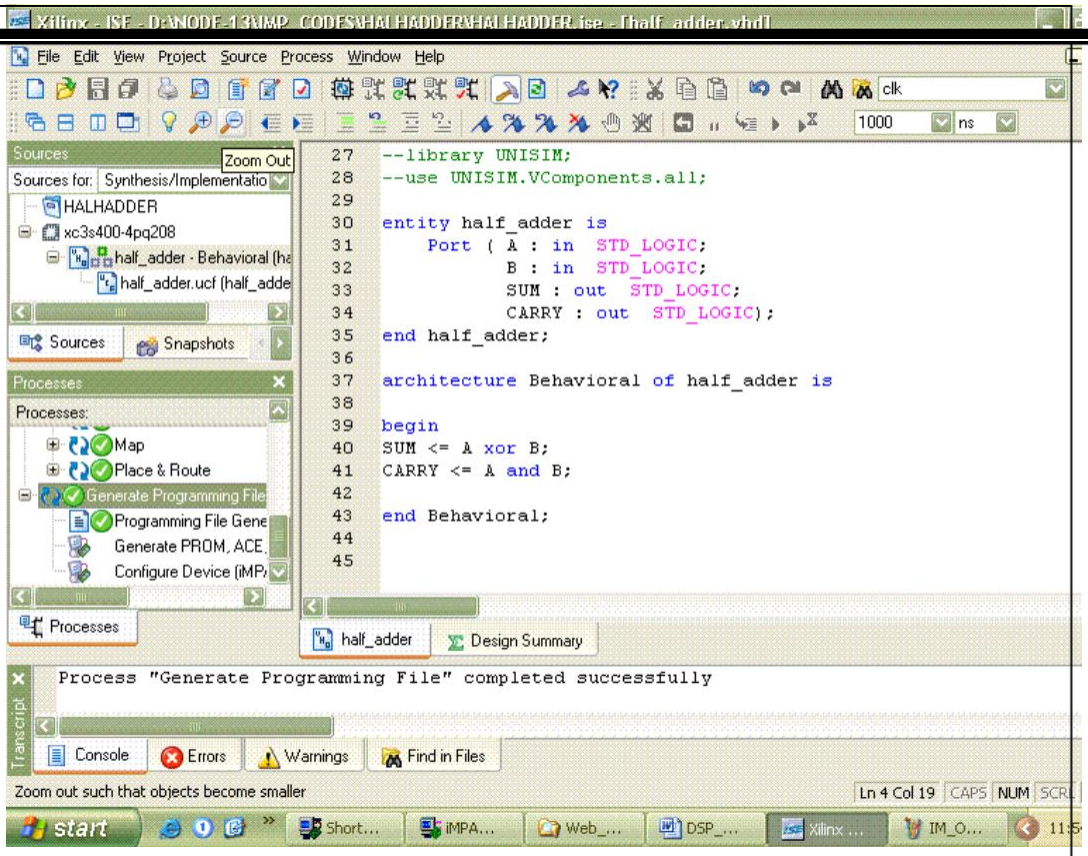
Step 9: *Create Constraints File(UCF)*

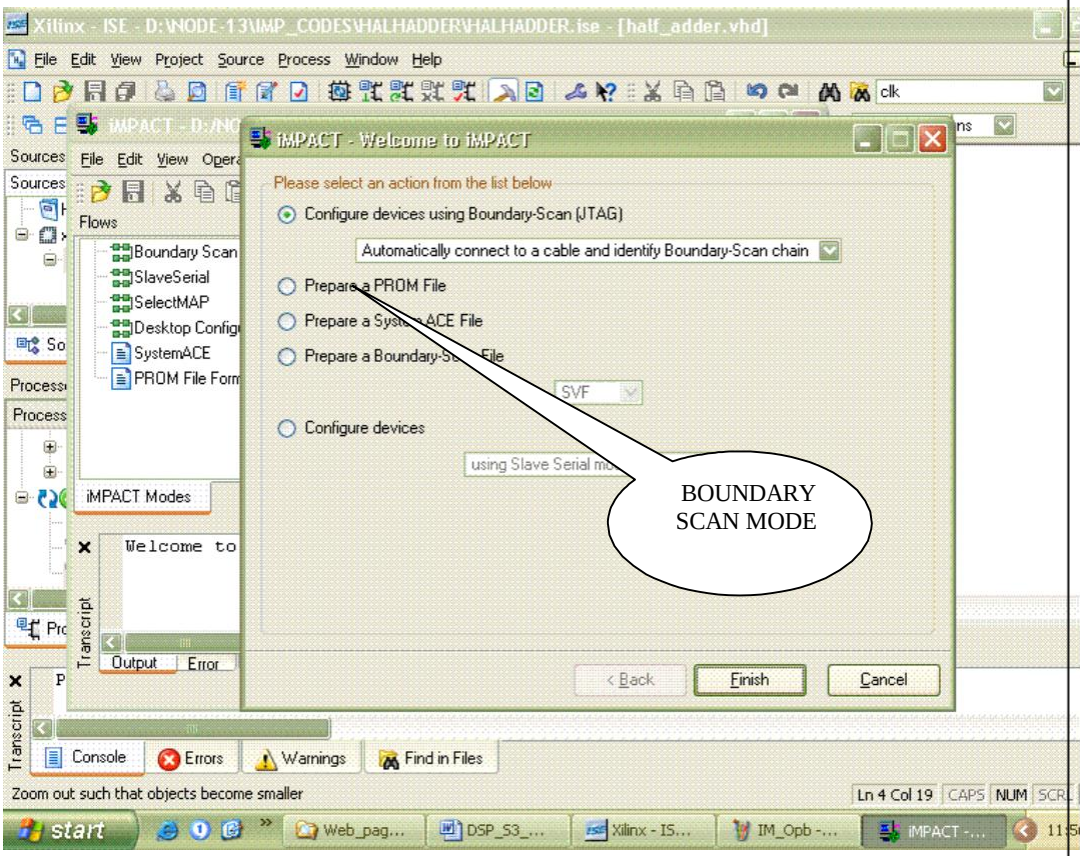
Click on the symbol of FPGA device and then right click☐ Click on new source ☐Implementation Constraints File and give the name ☐ Select entity☐Finish. Click on *User Constraint and* in that Double Click on *Assign Package Pins* option in Process window. Xilinx PACE window opens. Enter all the pin assignments in PACE., depending upon target device and number of input and outputs used in your design. *(sample code is given below for given design.)*

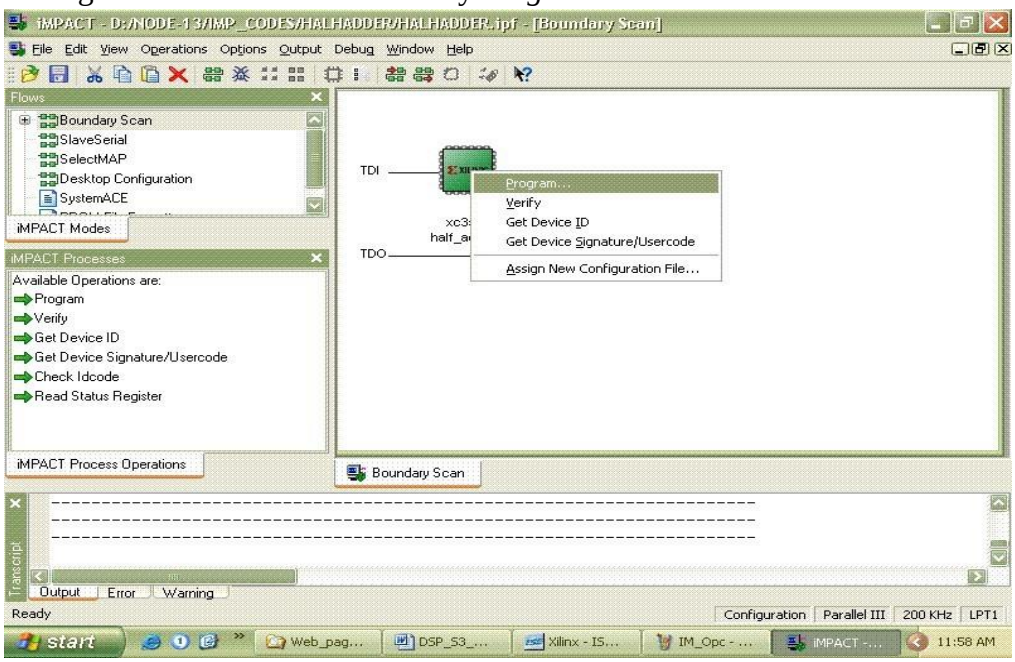


PIN ASSIGNMENT

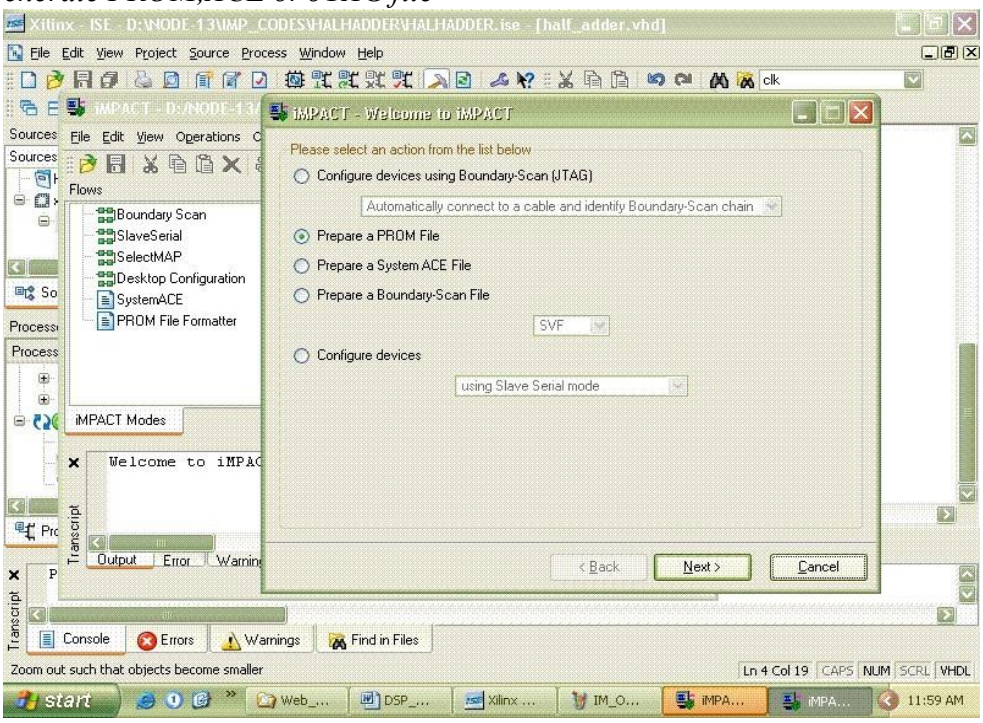
Step 10:	*Implementing a Design* Once synthesis is complete, you can place and route your design to fit into a Xilinx device, and you can also get some post place-and-route timing information about the design. The implementation stage consists of taking the synthesized netlist through translation, mapping, and place and route. To check your design as it is implemented, reports are available for each stage in the implementation process. Use the Xilinx Constraints Editor to add timing and location constraints for the implementation of your design. This procedure runs you through the basic flow for implementation. Right-click on *Implement Design*, and choose the Run option, *or* double left-click on *Implement Design.* 
Step 11:	*Generating Programming File* Right-click on *Generate Programming File*, choose the Run option, *or* double left-click on *Generate Programming File***.** This will generate the Bit stream
Step 12	*Downloading in* Boundary Scan Mode. *Note : Xilinx provides 2-tools for downloading purpose, viz.* • *iMPACT - is a command line and GUI based tool* • *PROM File Formatter*


Xilinx - ISE - D:\NODE-13\IMP_CODES\HALHADDER\HALHADDER.ise - [half_adder.vhd]
File Edit View Project Source Process Window Help
Sources
Sources for: Synthesis/Implementation
HALHADDER
xc3s400-4pq208
half_adder - Behavioral (ha
half_adder.ucf (half_adde
Sources
Snapshots
Processes
Processes:
Map
Place & Route
Generate Programming File
Programming File Gene
Generate PROM, ACE
Configure Device (iMP
Processes
27 --library UNISIM;
28 --use UNISIM.VComponents.all;
29
30 entity half_adder is
31 Port (A : in STD_LOGIC;
32 B : in STD_LOGIC;
33 SUM : out STD_LOGIC;
34 CARRY : out STD_LOGIC);
35 end half_adder;
36
37 architecture Behavioral of half_adder is
38
39 begin
40 SUM <= A xor B;
41 CARRY <= A and B;
42
43 end Behavioral;
44
45
half_adder
Design Summary
Process "Generate Programming File" completed successfully
Transcript
Console
Errors
Warnings
Find in Files
Zoom out such that objects become smaller
Ln 4 Col 19
CAPS
NUM
start

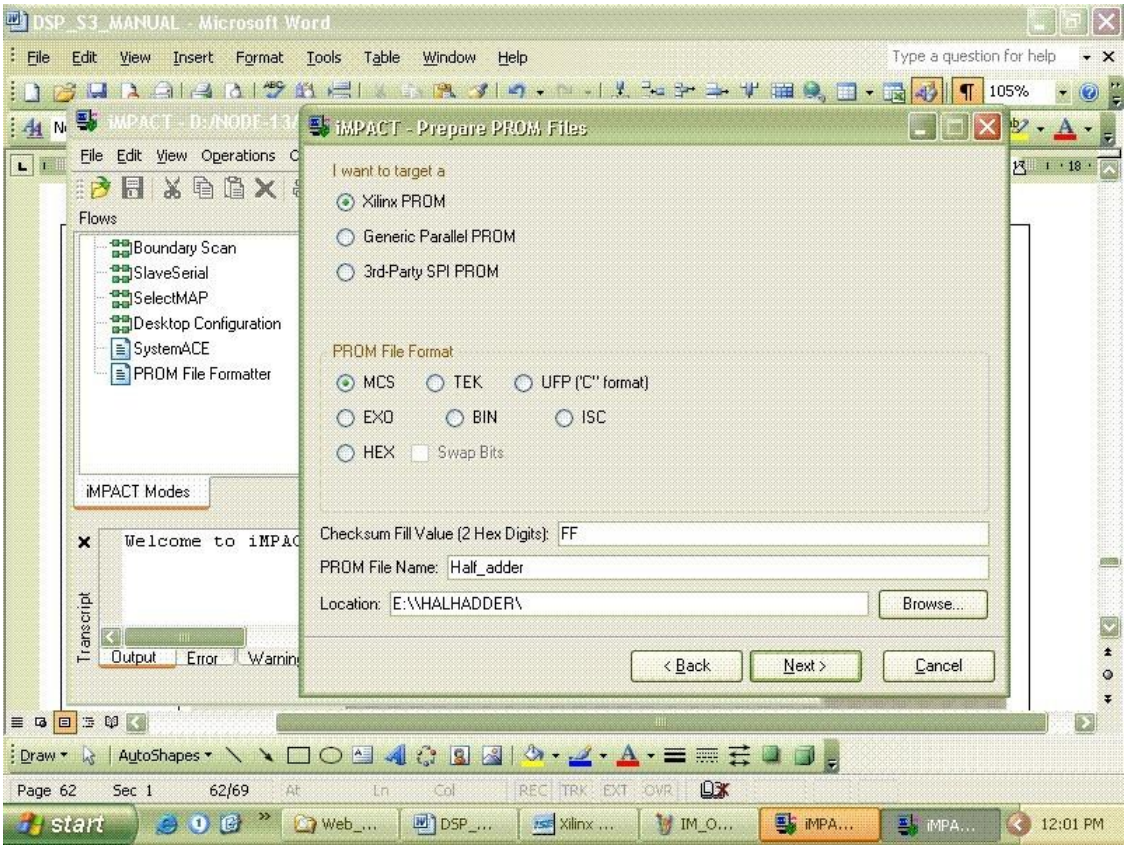

Xilinx - ISE - D:\NODE-13\IMP_CODES\HALHADDER\HALHADDER.ise - [half_adder.vhd]
File Edit View Project Source Process Window Help
iMPACT - Welcome to iMPACT
Please select an action from the list below
Configure devices using Boundary-Scan (JTAG)
Automatically connect to a cable and identify Boundary-Scan chain
Prepare a PROM File
Prepare a System ACE File
Prepare a Boundary-Scan File
SVF
Configure devices
using Slave Serial mo
Flows
Boundary Scan
SlaveSerial
SelectMAP
Desktop Config
SystemACE
PROM File Form
iMPACT Modes
Welcome to
Output
Error
< Back
Finish
Cancel
BOUNDARY SCAN MODE
Console
Errors
Warnings
Find in Files
Zoom out such that objects become smaller
Ln 4 Col 19
CAPS
NUM
start

	Procedure for downloading using iMPACT 1. Boundary Scan Mode *1.* Right click on "*Configure Device (iMPACT)" -> and Say RUN* or Double click on "*Configure Device (iMPACT)".* *2.* Right click in workspace and say *Initialize chain* .The device is seen. *3.* Right click on the device and say *Program.*  If the device is programmed properly, it says *Programming Succeeded* or else. *Programming Failed.* The *DONE* Led glows green if programming succeeds.
	Note: Before downloading make sure that Protoboard is connected to PC's parallel port with the cable provided and power to the Protoboard is ON.
Step 13:	Apply input through DIP Switches, output is displayed on LEDs
Step 14:	Configuration through PROM: *Generating PROM file*: FPGA can also be configured in Master Serial Mode through PROM. For this you need to program the PROM through a .mcs file.

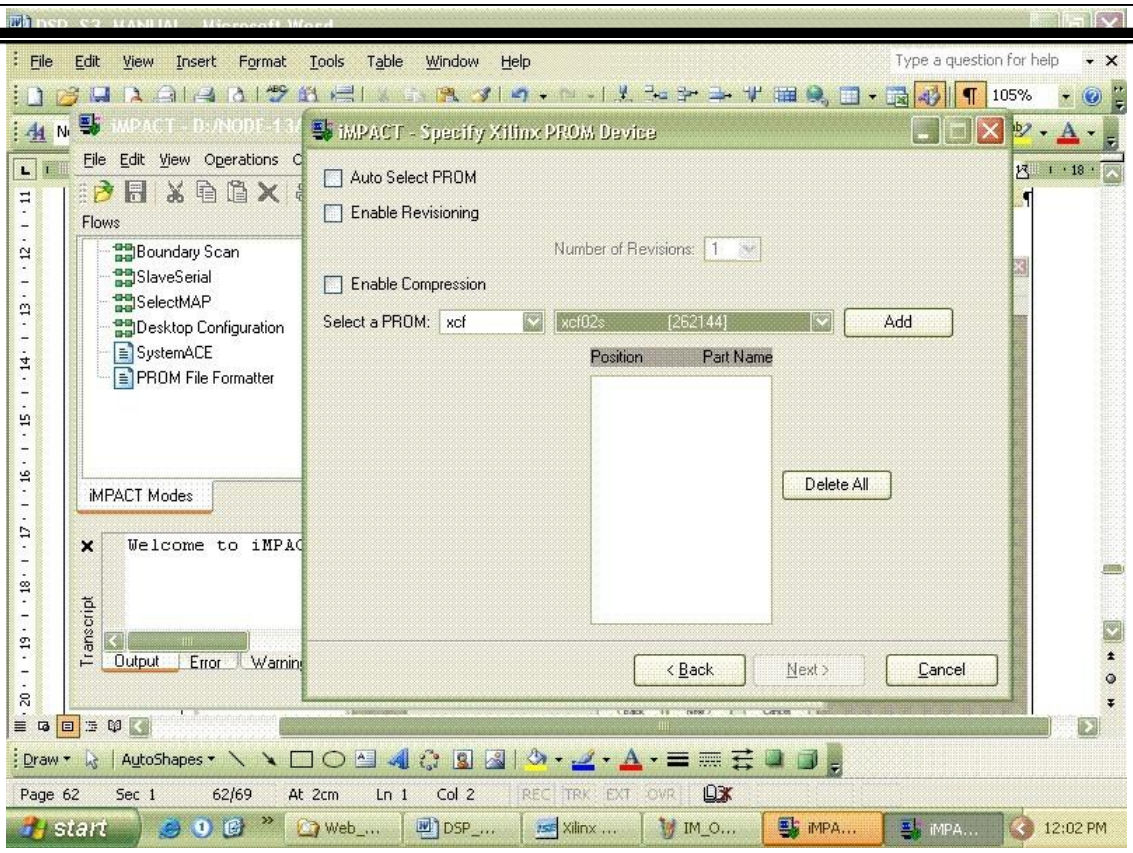
Right click on "*Generate PROM,ACE or JTAG file" -> and Say RUN* or Double click on "*Generate PROM,ACE or JTAG file"*



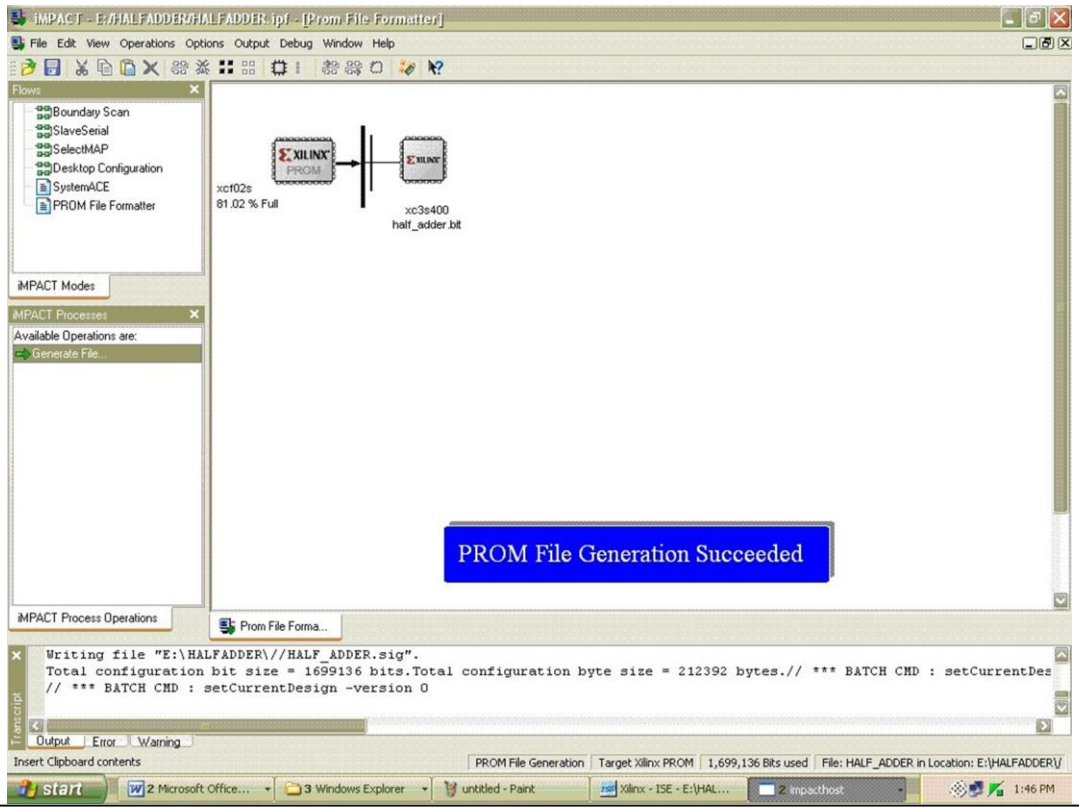
Specify the PROM file name and location where it is to be generated.



Specify the desired parameters of the PROM on board and say *ADD* then *FINISH*



Say *Generate File* from the Process Window.

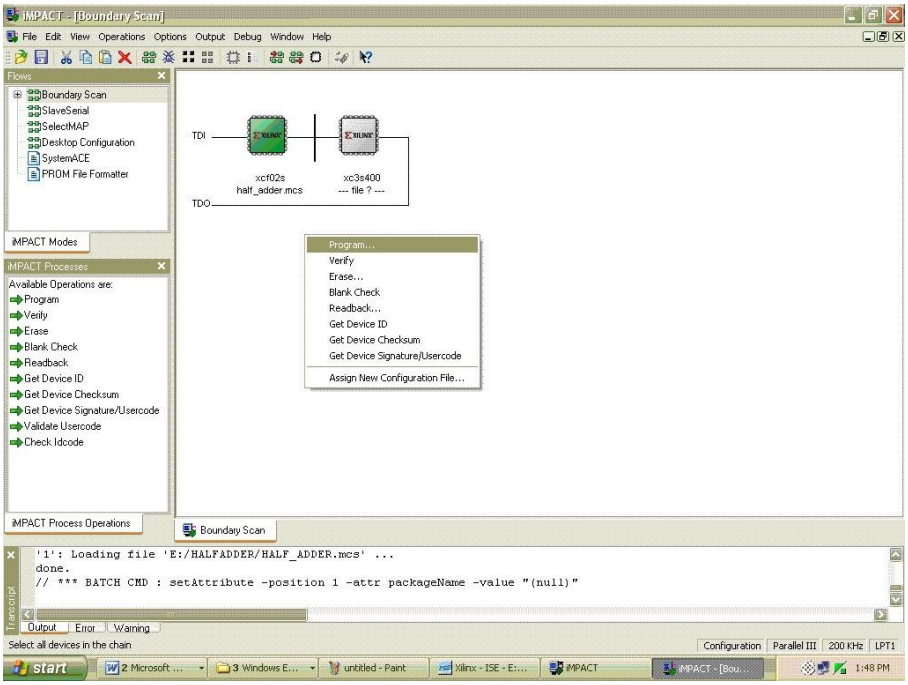


PROGRAMMING THE PROM

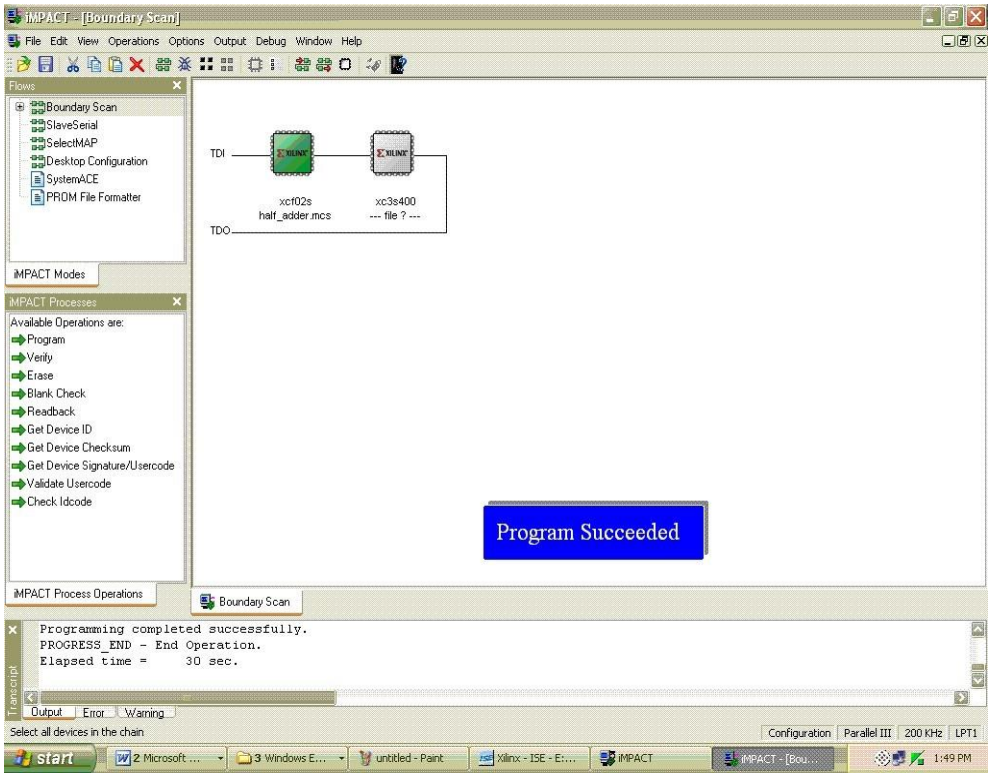
Note: Check the Jumper setting on the board. Refer the Chapter jumper Setting

Similar to Step 12.Initialize chain through iMPACT. PROM and FPGA devices on board are seen .Assign the generated mcs file and bit file as desired.

Right click the PROM symbol and say *PROGRAM.*



Now, whenever the board is powered on in master serial mode, FPGA is configured through PROM automatically.



Expt . No: 1 Date :	ALU:ARITHMATIC & LOGIC UNIT

Aim: A To write VHDL code, simulate with test bench, synthesis, implement on PLD.

Apparatus Required:

Synthesis tool: Xilinx ISE. Simulation tool: ModelSim Simulator FPGA STARTER Kit.

Theory:

An arithmetic logic unit (ALU) is a digital electronic circuit that performs arithmetic and bitwise logical operations oninteger binary numbers. It is a fundamental building block of the central processing unit (CPU) found in many computers. This is in contrast to a floating-point unit (FPU), which is a digital circuit that operates on floating point numbers with the aid of one or more internal ALUs. Powerful and complex ALUs are often used in modern, high performance CPUs, FPUs andgraphics processing units (GPUs). A single CPU, FPU or GPU may contain multiple ALUs.

The inputs to an ALU are the data to be operated on (called operands) and a code indicating the operation to be performed; the ALU's output is the result of the performed operation. In many designs, the ALU also exchanges additional information with a status register, which relates to the result of the current or previous operations.

Mathematician John von Neumann proposed the ALU concept in 1945 in a report on the foundations for a new computer called the EDVAC

Arithmetic operations

- *Add*: A and B are summed and the sum appears at Y and carry-out.
- *Add with carry*: A, B and carry-in are summed and the sum appears at Y and carry-out.
- *Subtract*: B is subtracted from A (or vice-versa) and the difference appears at Y and carry-out. For this function, carry-out is effectively a "borrow" indicator. This operation may also be used to compare the magnitudes of A and B; in such cases the Y output may be ignored by the processor, which is only interested in the status bits (particularly zero and negative) that result from the operation.
- *Subtract with borrow*: B is subtracted from A (or vice-versa) with borrow (carry-in) and the difference appears at Y and carry-out (borrow out).

- *Two's complement (negate)*: A (or B) is subtracted from zero and the difference appears at Y.
- *Increment*: A (or B) is increased by one and the resulting value appears at Y.
- *Decrement*: A (or B) is decreased by one and the resulting value appears at Y.
- *Pass through*: all bits of A (or B) appear unmodified at Y. This operation is typically used to determine the parity of the operand or whether it is zero or negative.

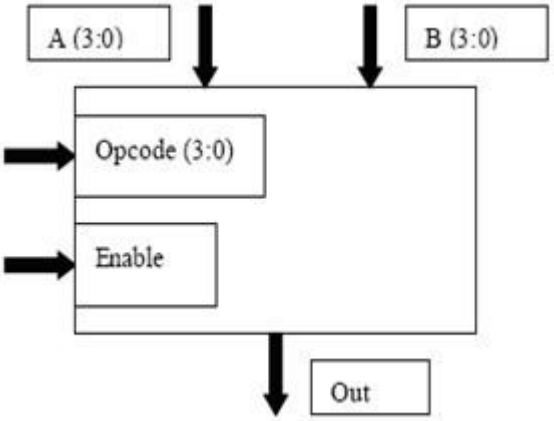
Bitwise logical operations

- *AND*: the bitwise AND of A and B appears at Y.
- *OR*: the bitwise OR of A and B appears at Y.
- *Exclusive-OR*: the bitwise XOR of A and B appears at Y.
- *One's complement*: all bits of A (or B) are inverted and appear at Y.

Bit shift operations

ALU shift operations cause operand A (or B) to shift left or right (depending on the opcode) and the shifted operand appears at Y. Simple ALUs typically can shift the operand by only one bit position, whereas more complex ALUs employ barrel shifters that allow them to shift the operand by an arbitrary number of bits in one operation. In all single-bit shift operations, the bit shifted out of the operand appears on carry-out; the value of the bit shifted into the operand depends on the type of shift.

- *Arithmetic shift*: the operand is treated as a two's complement integer, meaning that the most significant bit is a "sign" bit and is preserved.
- *Logical shift*: a logic zero is shifted into the operand. This is used to shift unsigned integers.
- *Rotate*: the operand is treated as a circular buffer of bits so that its least and most significant bits are effectively adjacent.
- *Rotate through carry*: the carry bit and operand are collectively treated as a circular buffer of bits.



Sel	Operation	Function	Unit
0000	Y<=A	Transfer A	Arithmetic
0001	Y<=A + 1	Increment A	
0010	Y<=A – 1	Decrement A	
0011	Y<=B	Transfer B	
0100	Y<=B + 1	Increment B	
0101	Y<=B – 1	Decrement B	
0110	Y<= A + B	Add A and B	
0111	Y<= A + B + CIN	Add A and B and Cin	
1000	Y<= NOT A	Complement A	Logic
1001	Y<= NOT B	Complement B	
1010	Y <= A AND B	AND	
1011	Y <= A OR B	OR	
1100	Y <= A NAND B	NAND	
1101	Y <= A NOR B	NOR	
1110	Y <= A XOR B	XOR	
1111	Y <= A XNOR B	XNOR	

VHDL CODE –

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity alu1 is
Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
b : in  STD_LOGIC_VECTOR (3 downto 0);
s : in  STD_LOGIC_VECTOR (2 downto 0);
c : out  STD_LOGIC_VECTOR (3 downto 0));
end alu1;
architecture Behavioral of alu1 is
begin
Process(a,b,s)
begin
case s is
when "000"=>c<=a+b;
when "001"=>c<=a-b;
when "010"=>c<=a and b;
when "011"=>c<=a or b;
when "100"=>c<=a xor b;
when "101"=>c<=not a;
when "110"=>c<=a xnor b;
when  others =>c<="0000";
end case;
end process;
end Behavioral;
```

TEST BENCH CODE

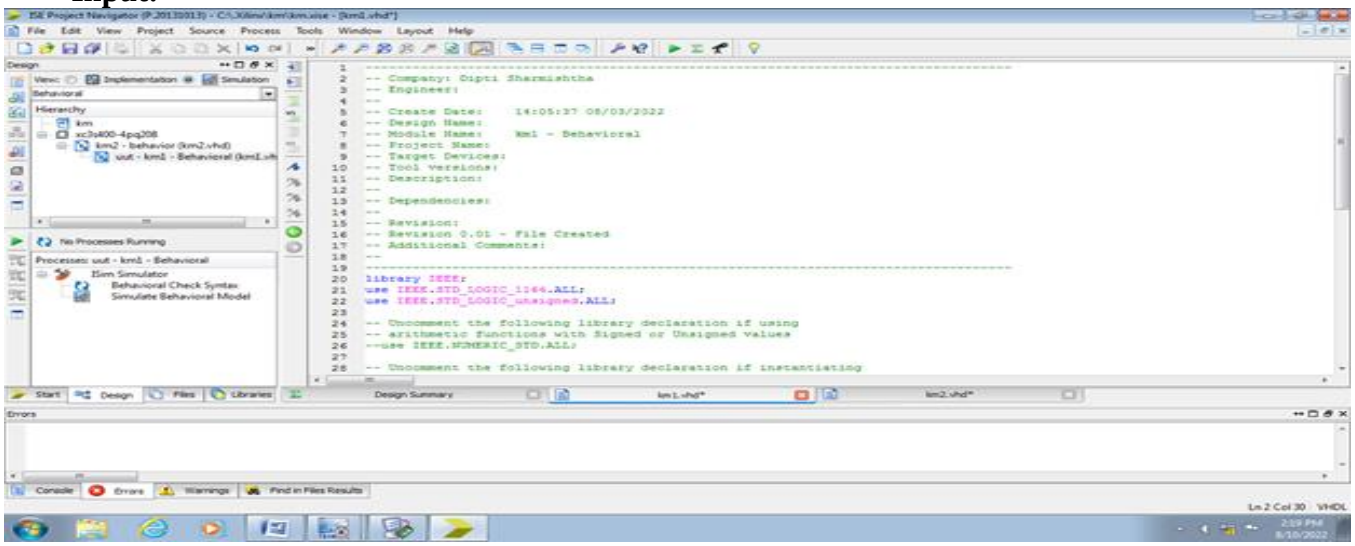
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;
ENTITY test1 IS
END test1;
ARCHITECTURE behavior OF test1 IS
-- Component Declaration for the Unit Under Test (UUT)
COMPONENT alu1
PORT(
a : IN  std_logic_vector(3 downto 0);
b : IN  std_logic_vector(3 downto 0);
s : IN  std_logic_vector(2 downto 0);
c : OUT  std_logic_vector(3 downto 0)
);
END COMPONENT;
--Inputs
signal a : std_logic_vector(3 downto 0) := (others => '0');
signal b : std_logic_vector(3 downto 0) := (others => '0');
signal s : std_logic_vector(2 downto 0) := (others => '0');
--Outputs
signal c : std_logic_vector(3 downto 0);
-- No clocks detected in port list. Replace <clock> below with
-- appropriate port name
--constant <clock>_ period : time := 100 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: alu1 PORT MAP (
a => a,
b => b,
s => s,
c => c
);
-- Clock process definitions
-- <clock>_process :process
--begin
--<clock> <= '0';
--wait for <clock>_period/2;
--<clock> <= '1';
--wait for <clock>_period/2;
-- end process;
-- Stimulus process
stim_proc: process
begin
a<="0000";
b<="0010";
s<="000";
wait for 100 ns;
a<="0000";
b<="0010";
s<="001";
wait for 100 ns;
```

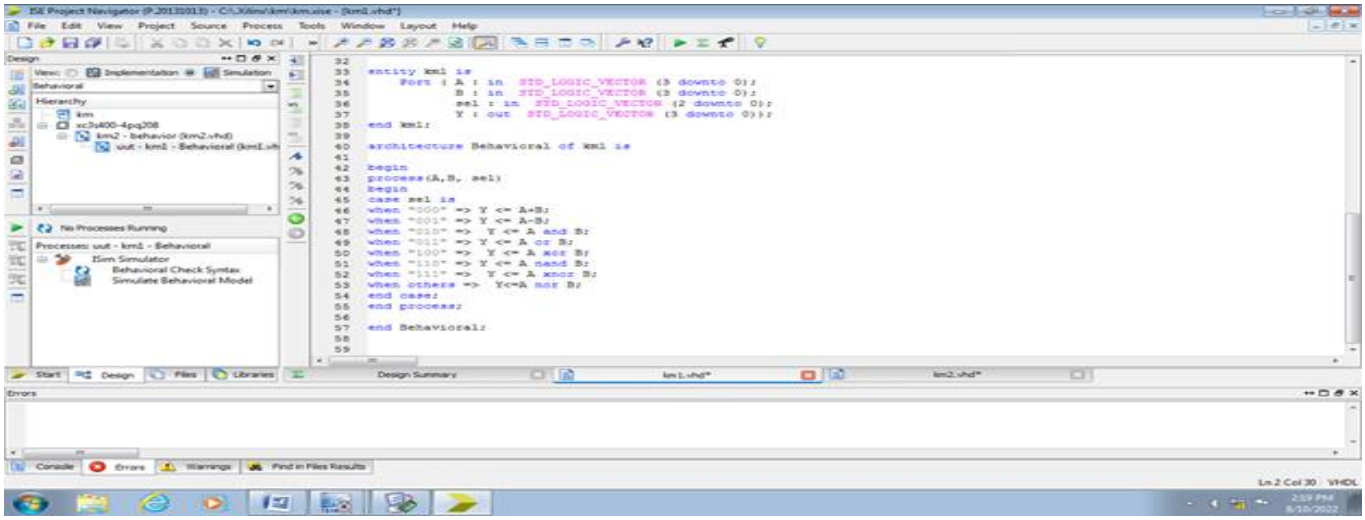
```
a<="0000";
b<="0010";
s<="010";
wait for 100 ns;
a<="0000";
b<="0010";
s<="011";
wait for 100 ns;
a<="0000";
b<="0010";
s<="100";
wait for 100 ns;
a<="0000";
b<="0010";
s<="101";
wait for 100 ns;
a<="0000";
b<="0010";
s<="110";
wait for 100 ns;
-- hold reset state for 100 ns.
wait for 100 ns;
wait for 1000 ns;
-- insert stimulus here
wait;
end process;
END;
```

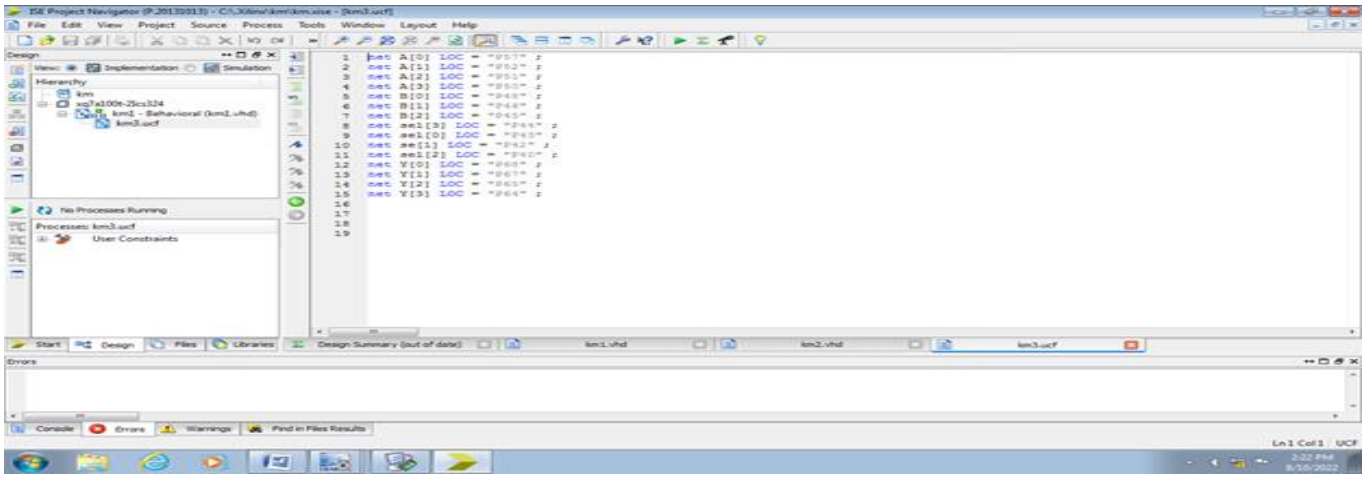
UCF CODE

```
NET a[0] LOC="P57";
NET a[1] LOC="P52";
NET a[2] LOC="P51";
NET a[3] LOC="P50";
NET b[0] LOC="P48";
NET b[1] LOC="P46";
NET b[2] LOC="P45";
NET b[3] LOC="P44";
NET s[0] LOC="P43";
NET s[1] LOC="P42";
NET s[2] LOC="P40";
NET c[0] LOC="P68";
NET c[1] LOC="P67";
NET c[2] LOC="P65";
NET c[3] LOC="P64";
```

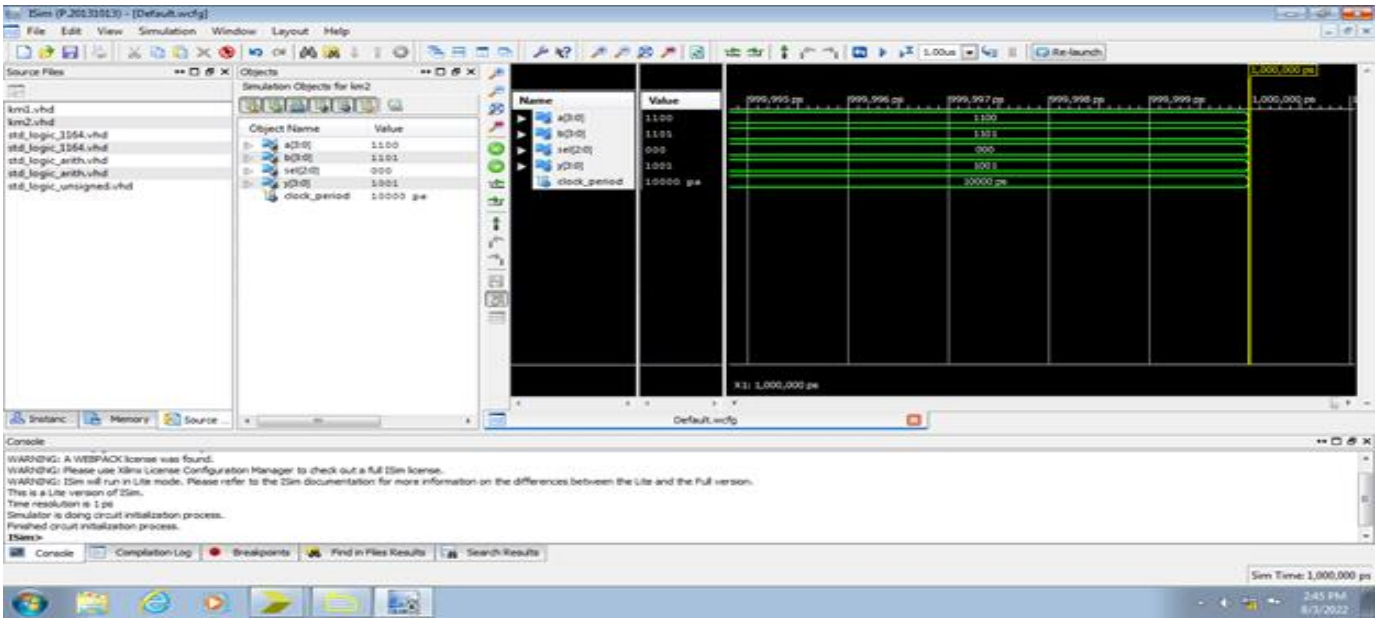
Input:

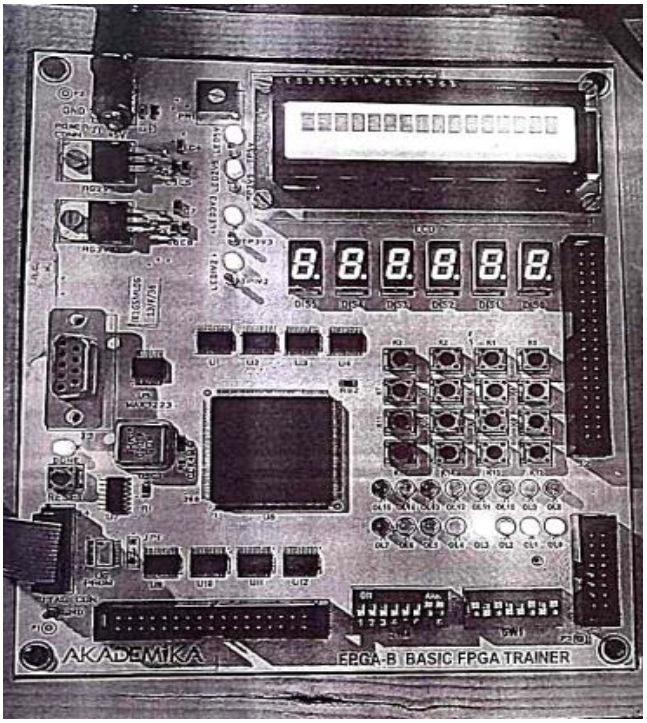


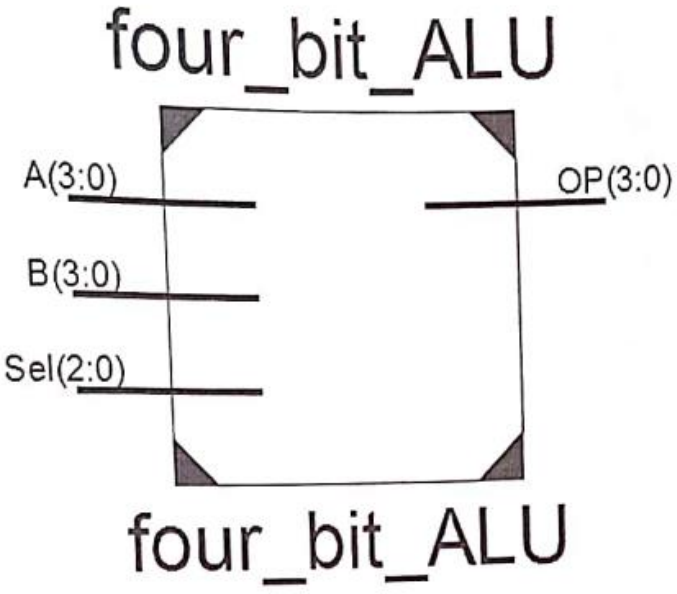




Output:







Conclusion:

Expt . No: 2 Date :	Universal shift register

Aim: Write a VHDL Code for Universal shift register with mode selection input for SISO, SIPO, PISO, & PIPO Modes.

Apparatus Required:

Synthesis tool: Xilinx ISE. Simulation tool: ModelSim Simulator FPGA STARTER Kit.

Theory:

The Shift Register :(SISO, SIPO, PISO, PIPO)

The Shift Register is another type of sequential logic circuit that can be used for the storage or the transfer of data in the form of binary numbers. This sequential device loads the data present on its inputs and then moves or "shifts" it to its output once every clock cycle, hence the name "shift register".

A shift register basically consists of several single bit "D-Type Data Latches", one for each data bit, either a logic "0" or a "1", connected together in a serial type daisy-chain arrangement so that the output from one data latch becomes the input of the next latch and so on.

Data bits may be fed in or out of a shift register serially, that is one after the other from either the left or the right direction, or all together at the same time in a parallel configuration.

The number of individual data latches required to make up a single Shift Register device is usually determined by the number of bits to be stored with the most common being 8-bits (one byte) wide constructed from eight individual data latches.

Shift Registers are used for data storage or for the movement of data and are therefore commonly used inside calculators or computers to store data such as two binary numbers before they are added together, or to convert the data from either a serial to parallel or parallel to serial format. The individual data latches that make up a single shift register are all driven by a common clock (Clk) signal making them synchronous devices.

Shift register IC's are generally provided with a clear or reset connection so that they can be "SET" or "RESET" as required.

Generally, shift registers operate in one of four different modes with the basic movement of data through a shift register being:

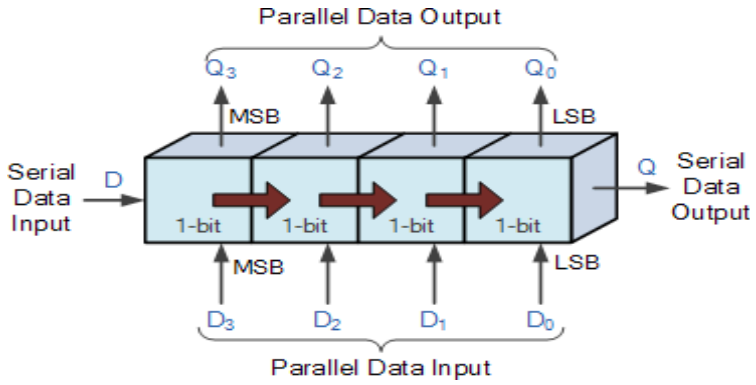
• Serial-in to Parallel-out (SIPO) - the register is loaded with serial data, one bit at a time, with the stored data being available at the output in parallel form.

• Serial-in to Serial-out (SISO) - the data is shifted serially "IN" and "OUT" of the register, one bit at a time in either a left or right direction under clock control.

• Parallel-in to Serial-out (PISO) - the parallel data is loaded into the register simultaneously and is shifted out of the register serially one bit at a time under clock control.

• Parallel-in to Parallel-out (PIPO) - the parallel data is loaded simultaneously into the register, and transferred together to their respective outputs by the same clock pulse.

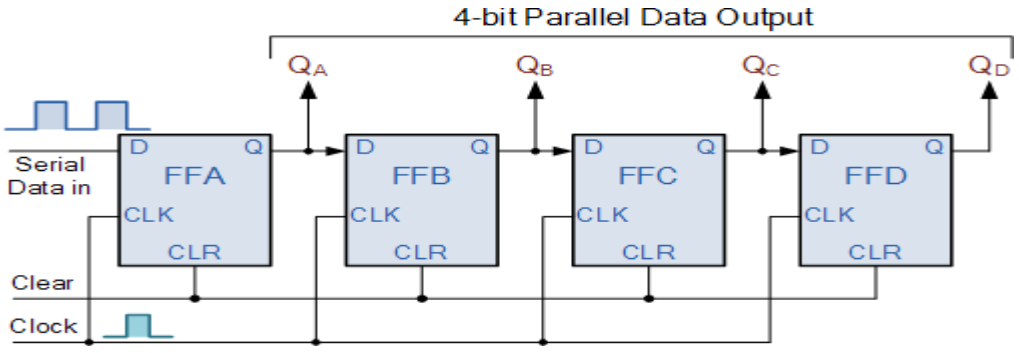
The effect of data movement from left to right through a shift register can be presented graphically as:



Also, the directional movement of the data through a shift register can be either to the left, (left shifting) to the right, (right shifting) left-in but right-out, (rotation) or both left and right shifting within the same register thereby making it bidirectional.

Serial-in to Parallel-out (SIPO) Shift Register:

4-bit Serial-in to Parallel-out Shift Register:



The operation is as follows. Lets assume that all the flip-flops (FFA to FFD) have just been RESET (CLEAR input) and that all the outputs Q_A to Q_D are at logic level "0" ie, no parallel data output.

If a logic "1" is connected to the DATA input pin of FFA then on the first clock pulse the output of FFA and therefore the resulting Q_A will be set HIGH to logic "1" with all the other outputs still remaining LOW at logic "0". Assume now that the DATA input pin of FFA has returned LOW again to logic "0" giving us one data pulse or 0-1-0.

The second clock pulse will change the output of FFA to logic "0" and the output of FFB and Q_B HIGH to logic "1" as its input D has the logic "1" level on it from Q_A. The logic "1" has now moved or been "shifted" one place along the register to the right as it is now at Q_A.

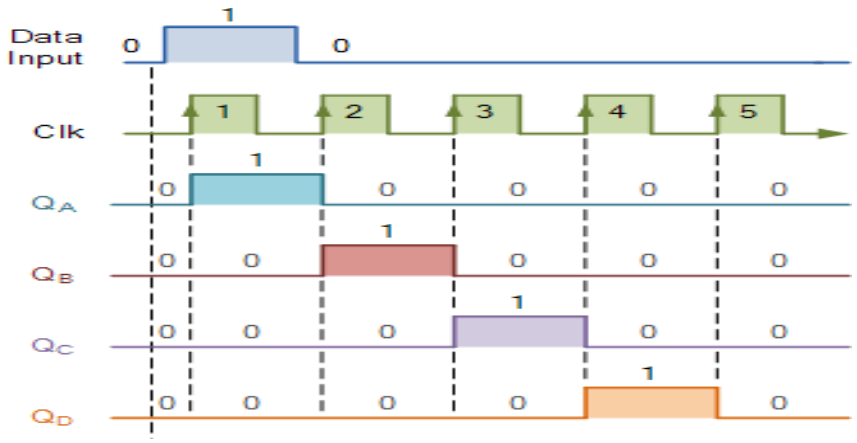
When the third clock pulse arrives this logic "1" value moves to the output of FFC (Q_C) and so on until the arrival of the fifth clock pulse which sets all the outputs Q_A to Q_D back again to logic level "0" because the input to FFA has remained constant at logic level "0".

The effect of each clock pulse is to shift the data contents of each stage one place to the right, and this is shown in the following table until the complete data value of 0-0-0-1 is stored in the register. This data value can now be read directly from the outputs of Q_A to Q_D.

Then the data has been converted from a serial data input signal to a parallel data output. The truth table and following waveforms show the propagation of the logic "1" through the register from left to right as follows.

Basic Data Movement Through A Shift Register

Clock Pulse No	QA	QB	QC	QD
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0



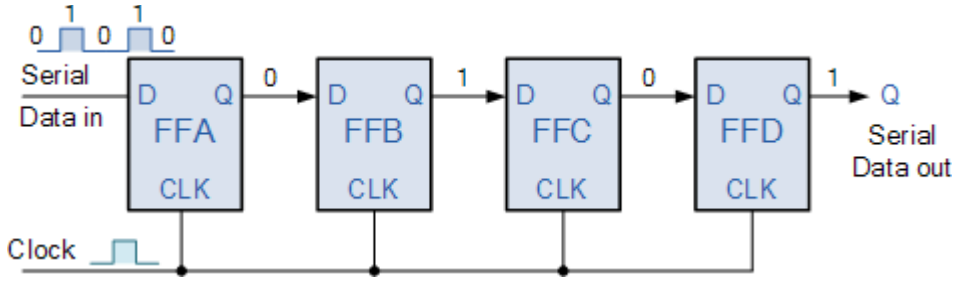
Note that after the fourth clock pulse has ended the 4-bits of data (0-0-0-1) are stored in the register and will remain there provided clocking of the register has stopped. In practice the input data to the register may consist of various combinations of logic "1" and "0". Commonly available SIPO IC's include the standard 8-bit 74LS164 or the 74LS594.

Serial-in to Serial-out (SISO) Shift Register:

This **shift register** is very similar to the SIPO above, except were before the data was read directly in a parallel form from the outputs Q_A to Q_D, this time the data is allowed to flow straight through the register and out of the other end. Since there is only one output, the DATA leaves the shift register one bit at a time in a serial pattern, hence the name **Serial-in to Serial-Out Shift Register** or **SISO**.

The SISO shift register is one of the simplest of the four configurations as it has only three connections, the serial input (SI) which determines what enters the left hand flip-flop, the serial output (SO) which is taken from the output of the right hand flip-flop and the sequencing clock signal (Clk). The logic circuit diagram below shows a generalized serial-in serial-out shift register. Commonly available IC's include the 74HC595 8-bit Serial-in to Serial-out Shift Register all with 3-state outputs.

4-bit Serial-in to Serial-out Shift Register:

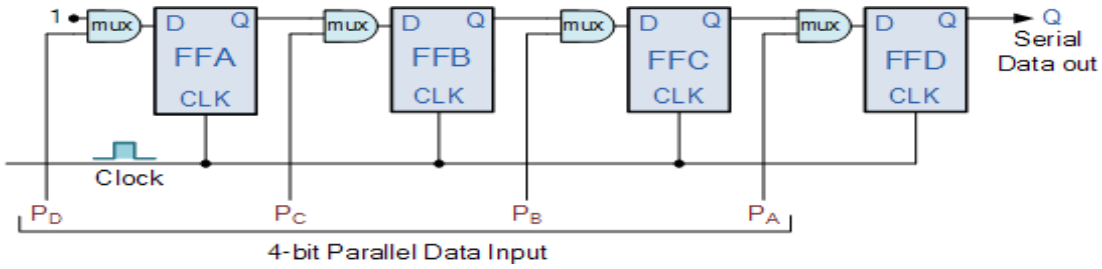


Parallel-in to Serial-out (PISO) Shift Register:

The Parallel-in to Serial-out shift register acts in the opposite way to the serial-in to parallel-out one above. The data is loaded into the register in a parallel format in which all the data bits enter their inputs simultaneously, to the parallel input pins P_A to P_D of the register. The data is then read out sequentially in the normal shift-right mode from the register at Q representing the data present atP_A to P_D.

This data is outputted one bit at a time on each clock cycle in a serial format. It is important to note that with this type of data register a clock pulse is not required to parallel load the register as it is already present, but four clock pulses are required to unload the data.

4-bit Parallel-in to Serial-out Shift Register:



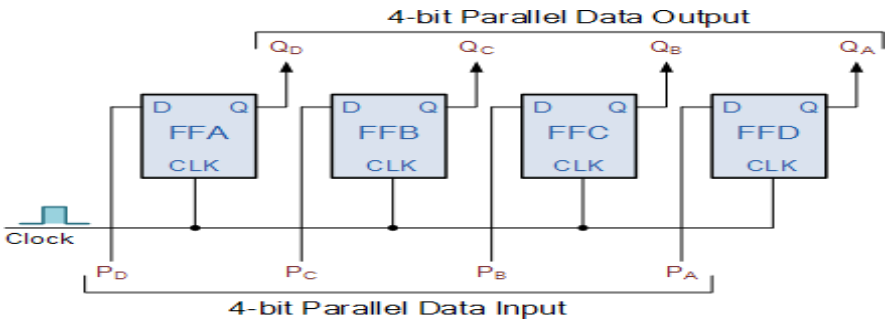
As this type of shift register converts parallel data, such as an 8-bit data word into serial format, it can be used to multiplex many different input lines into a single serial DATA stream which can be sent directly to a computer or transmitted over a communications line. Commonly available IC's include the 74HC166 8-bit Parallel-in/Serial-out Shift Registers.

Parallel-in to Parallel-out (PIPO) Shift Register:

The final mode of operation is the Parallel-in to Parallel-out Shift Register. This type of shift register also acts as a temporary storage device or as a time delay device similar to the SISO configuration above. The data is presented in a parallel format to the parallel input pins P_A to P_D and then transferred together directly to their respective output pins Q_A to Q_A by the

same clock pulse. Then one clock pulse loads and unloads the register. This arrangement for parallel loading and unloading is shown below.

4-bit Parallel-in to Parallel-out Shift Register:



The PIPO shift register is the simplest of the four configurations as it has only three connections, the parallel input (PI) which determines what enters the flip-flop, the parallel output (PO) and the sequencing clock signal (Clk).

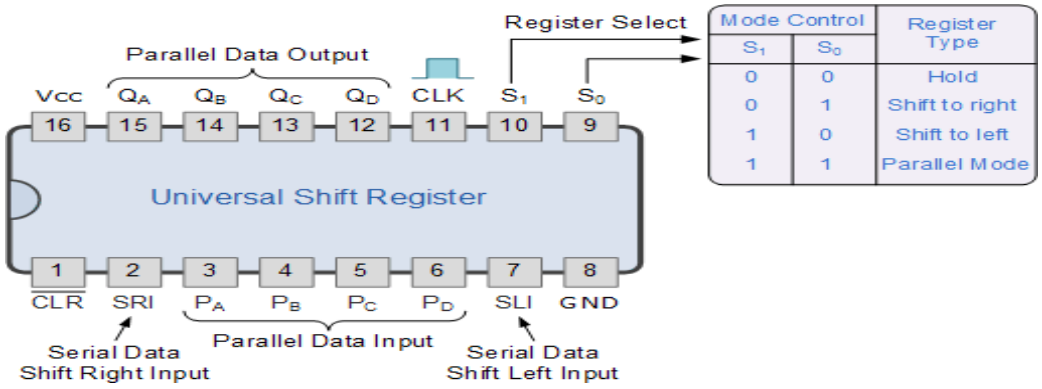
Similar to the Serial-in to Serial-out shift register, this type of register also acts as a temporary storage device or as a time delay device, with the amount of time delay being varied by the frequency of the clock pulses. Also, in this type of register there are no interconnections between the individual flip-flops since no serial shifting of the data is required.

Bi-directional Universal Shift Register:

Today, there are many high speed bi-directional "universal" type **Shift Registers** available such as the TTL 74LS194, 74LS195 or the CMOS 4035 which are available as 4-bit multi-function devices that can be used in either serial-to-serial, left shifting, right shifting, serial-to-parallel, parallel-to-serial, or as a parallel-to-parallel multifunction data register, hence the name "Universal".

These universal shift registers can perform any combination of parallel and serial input to output operations but require additional inputs to specify desired function and to pre-load and reset the device. A commonly used universal shift register is the TTL 74LS194 as shown below.

4-bit Universal Shift Register 74LS194:



Universal shift registers are very useful digital devices. They can be configured to respond to operations that require some form of temporary memory storage or for the delay of information such as the SISO or PIPO configuration modes or transfer data from one point to another in either a serial or parallel format. Universal shift registers are frequently used in arithmetic operations to shift data to the left or right for multiplication or division.

Shift Register Summary

- A simple Shift Register can be made using only D-type flip-Flops, one flip-Flop for each data bit.
- The output from each flip-Flop is connected to the D input of the flip-flop at its right.
- Shift registers hold the data in their memory which is moved or "shifted" to their required positions on each clock pulse.
- Each clock pulse shifts the contents of the register one bit position to either the left or the right.
- The data bits can be loaded one bit at a time in a series input (SI) configuration or be loaded simultaneously in a parallel configuration (PI).
- Data may be removed from the register one bit at a time for a series output (SO) or removed all at the same time from a parallel output (PO).
- One application of shift registers is in the conversion of data between serial and parallel, or parallel to serial.

Shift registers are identified individually as SIPO, SISO, PISO, PIPO, or as a Universal Shift Register with all the functions combined within a single device.

VHDLCODE-

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
entity shift18 is
Port ( si : in  STD_LOGIC;
pi : in  STD_LOGIC_VECTOR (3 downto 0);
so : out  STD_LOGIC;
po : out  STD_LOGIC_VECTOR (3 downto 0);
clk : in  STD_LOGIC;
s : in  STD_LOGIC_VECTOR (1 downto 0));
end shift18;
architecture Behavioral of shift18 is
signal temp:std_logic_vector (3 downto 0);
begin
process(clk,s)
begin
if(clk='1' and clk' event)then
if(s="00")then
temp(3 downto 1) <= temp(2 downto 0);
temp(0)<=si;
so<=temp(3);
elsif(s="01")then
temp(3 downto 1)<=temp(2 downto 0);
temp(0)<=si;
po(3 downto 0)<=temp(3 downto 0);
elsif(s="10")then
temp(3 downto 1)<=temp(2 downto 0);
temp(3 downto 0)<=pi(3 downto 0);
so<=temp(3);
elsif(s="11")then
temp(3 downto 1)<=temp(2 downto 0);
temp(3 downto 0)<=pi(3 downto 0);
po(3 downto 0)<=temp(3 downto 0);
end if;
end if;
end process;
end Behavioral;
```

TESTBENCHCODE-

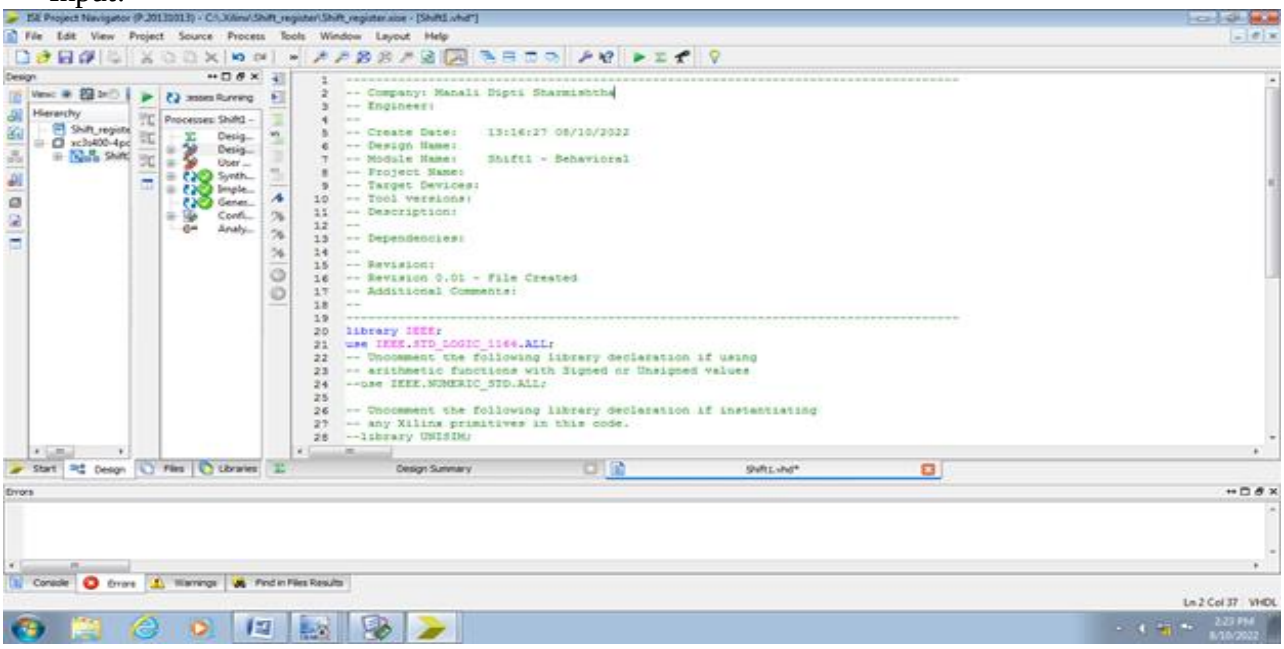
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;
ENTITY test1 IS
END test1;
ARCHITECTURE behavior OF test1 IS
-- Component Declaration for the Unit Under Test (UUT)
COMPONENT shift18
PORT(
si : IN  std_logic;
pi : IN  std_logic_vector(3 downto 0);
so : OUT  std_logic;
po : OUT  std_logic_vector(3 downto 0);
clk : IN  std_logic;
s : IN  std_logic_vector(1 downto 0)
);
END COMPONENT;
--Inputs
signal si : std_logic := '0';
signal pi : std_logic_vector(3 downto 0) := (others => '0');
signal clk : std_logic := '0';
signal s : std_logic_vector(1 downto 0) := (others => '0');
--Outputs
signal so : std_logic;
signal po : std_logic_vector(3 downto 0);
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: shift18 PORT MAP (
si => si,
pi => pi,
so => so,
po => po,
clk => clk,
s => s
);
-- Clock process definitions
clk_process :process
begin
clk <= '0';
wait for clk_period/2;
clk <= '1';
wait for clk_period/2;
end process;
-- Stimulus process
stim_proc: process
```

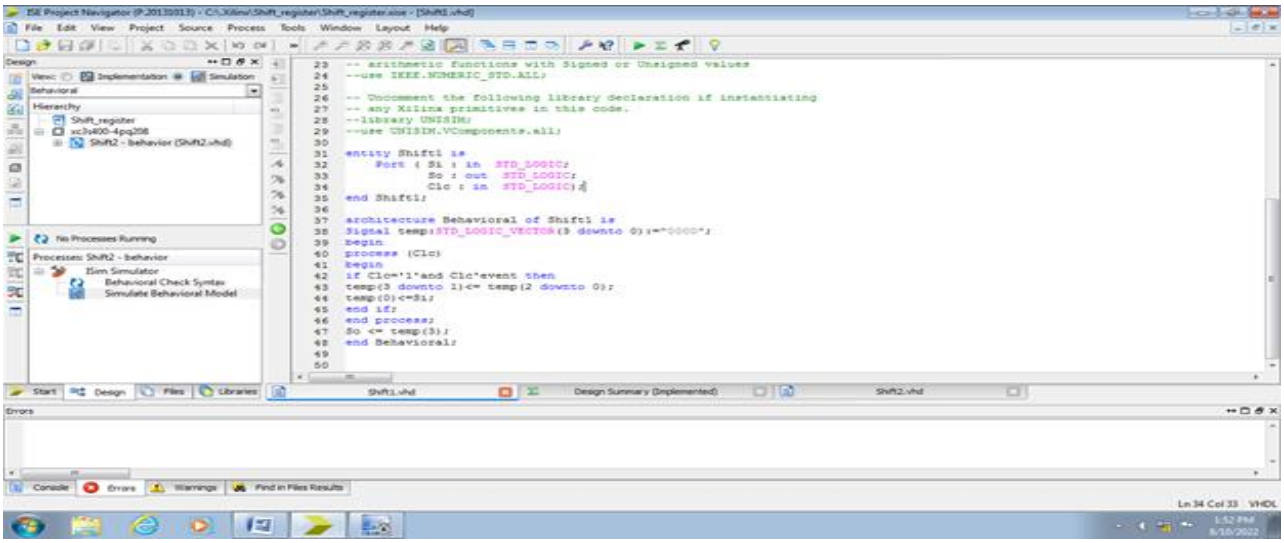
```
      begin
      s<="00";
      si<='1';
      pi<="0010";
-- hold reset state for 100 ns.
wait for 100 ns;
      s<="01";
      si<='1';
      pi<="0010";
                wait for 100 ns;
      s<="10";
      si<='1';
      pi<="0010";
      wait for 100 ns;
      s<="11";
      si<='1';
      pi<="0010";
      wait for 100 ns;
        wait for 1000 ns;
-- insert stimulus here
          wait;
         end process;
      END;
```

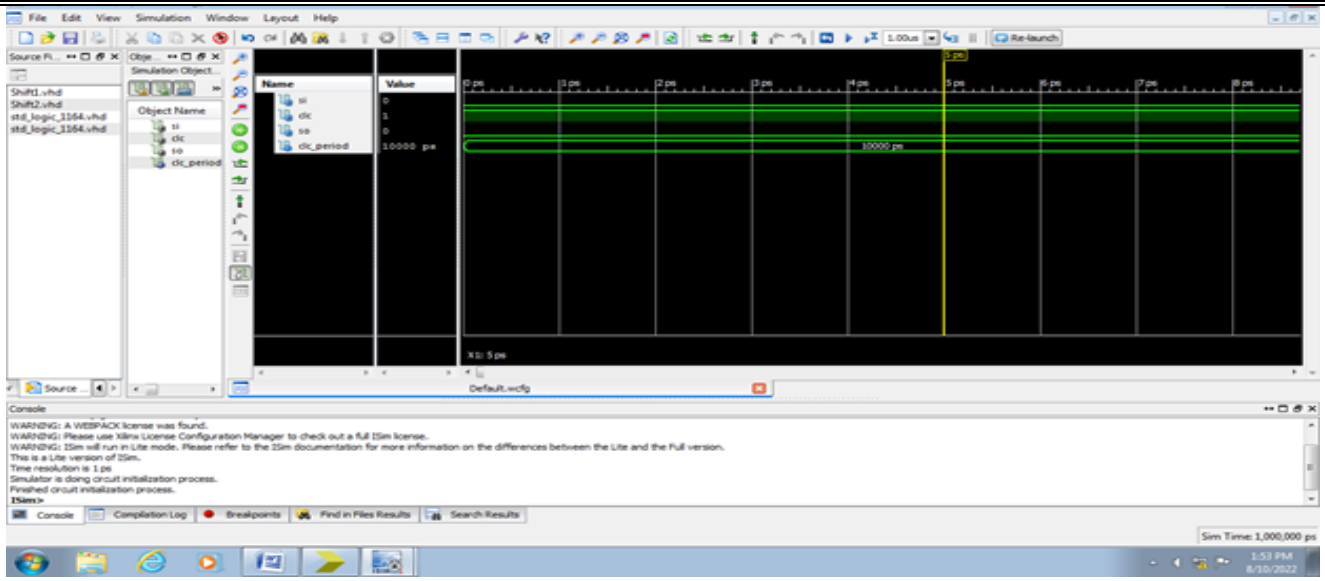
UCF PROGRAM-

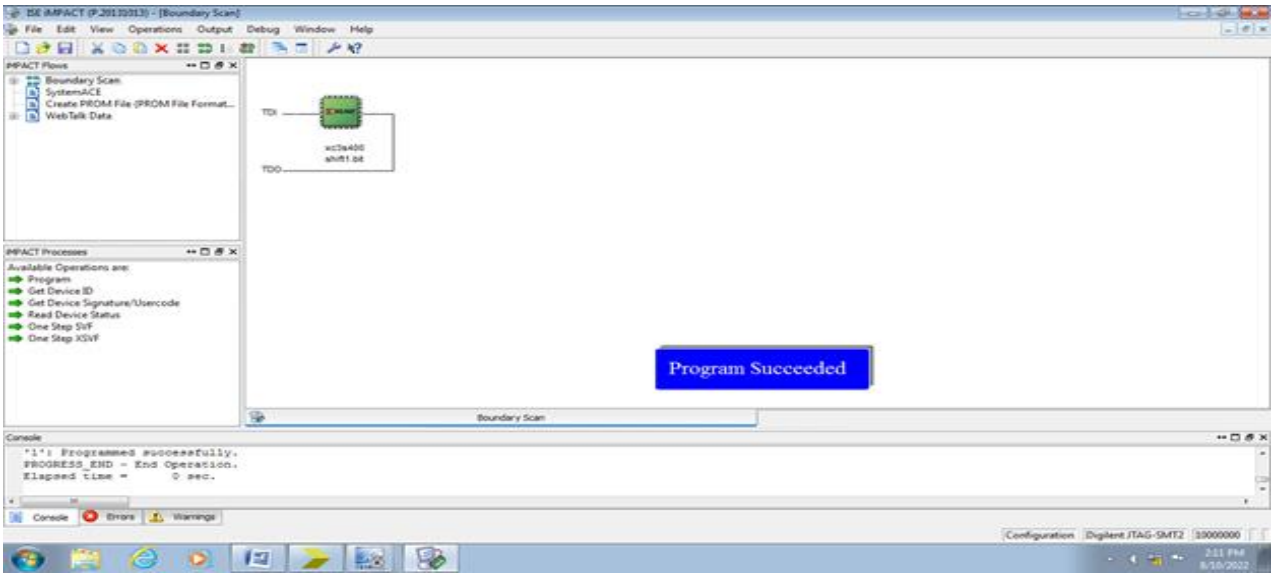
```
net "clk" loc ="p181";
net "si" loc ="p34";
net "pi[0]" loc="p52";
net "pi[1]" loc="p51";
net "pi[2]" loc="p50";
net "pi[3]" loc="p48";
net "so" loc="p68";
net "po[0]" loc="p67";
net "po[1]" loc="p65";
net "po[2]" loc="p64";
net "po[3]" loc="p63";
net "s[0]" loc="p46";
net "s[1]" loc="p45";
```

Input:


Company: Manali Dipti Sharmishtha
Create Date: 13:16:27 08/10/2022
Module Name: Shift1 - Behavioral
Revision 0.01 - File Created
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity Shift1 is
Port (Si : in STD_LOGIC;
So : out STD_LOGIC;
Clc : in STD_LOGIC);
end Shift1;
architecture Behavioral of Shift1 is
Signal temp:STD_LOGIC_VECTOR(3 downto 0):="0000";
begin
process (Clc)
begin
if Clc='1'and Clc'event then
temp(3 downto 1)<= temp(2 downto 0);
temp(0)<=Si;
end if;
end process;
So <= temp(3);
end Behavioral;





Conclusion:

Expt . No: 3 Date :	LCD INTERFACE

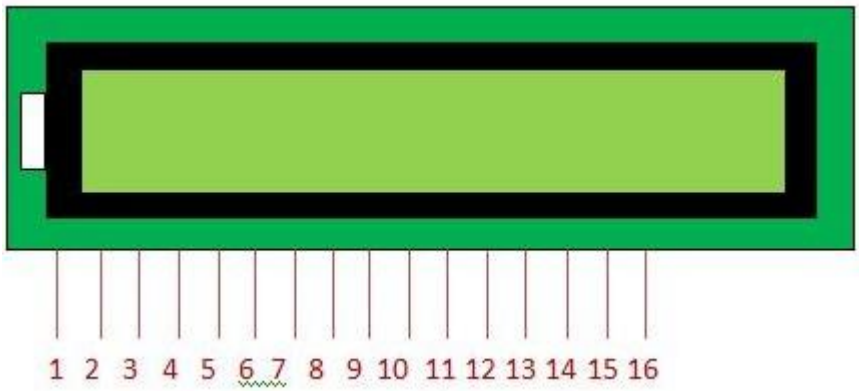
Aim: A To write VHDL code, simulate with test bench, synthesis, implement on PLD.

Apparatus Required:

Synthesis tool: Xilinx ISE.
Simulation tool: ISim Simulator
FPGA STARTER Kit(Spartan 3)

Description:

The expansion of LCD is Liquid Crystal Display which is used to display the character. The character is represented as the ASCII value (American Standard Code for Information Interchange). To display the character only the ASCII values are sent to LCD. The ASCII is 8bit. Here we have used 16x2 LCD with 5x8 pixel matrix (per character). The definition of 16x2 is the LCD contains 2 rows and 16 characters can be displayed per line. The single character displayed in 5x8 pixel matrix.

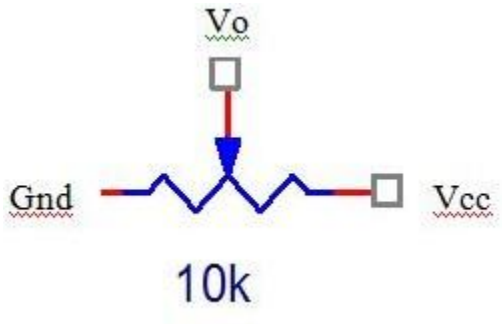


Pin Description:
Power supply

The first two pins of LCD must be connected to +5v and 0v.

Vo

The Vo pin is a contrast pin which is used to control the contrast of LCD. That is done by variable resistor. In FPGA kit the 10K variable resistor is used to control the contrast. The simple connection is given. There are three pins the two pins are connected to power supply and middle one is connected to Vo.



RS (Register Select)

RS is a command pin for LCD. The LCD command and RW operations are determined by RS pin. The LCD contains two register which is the data and command register. When command writes on the LCD, the data register is used. When the data either read or write on the LCD, the command register is used. The selections of register are determined by the logic status of RS. If the logic state of RS is '1', the data register is selected. If the logic state of RS is '0', the command register is selected.

Enable

When the data send to data pins of LCD, the high to low pulse will be given.

Data pins (D0-D1)

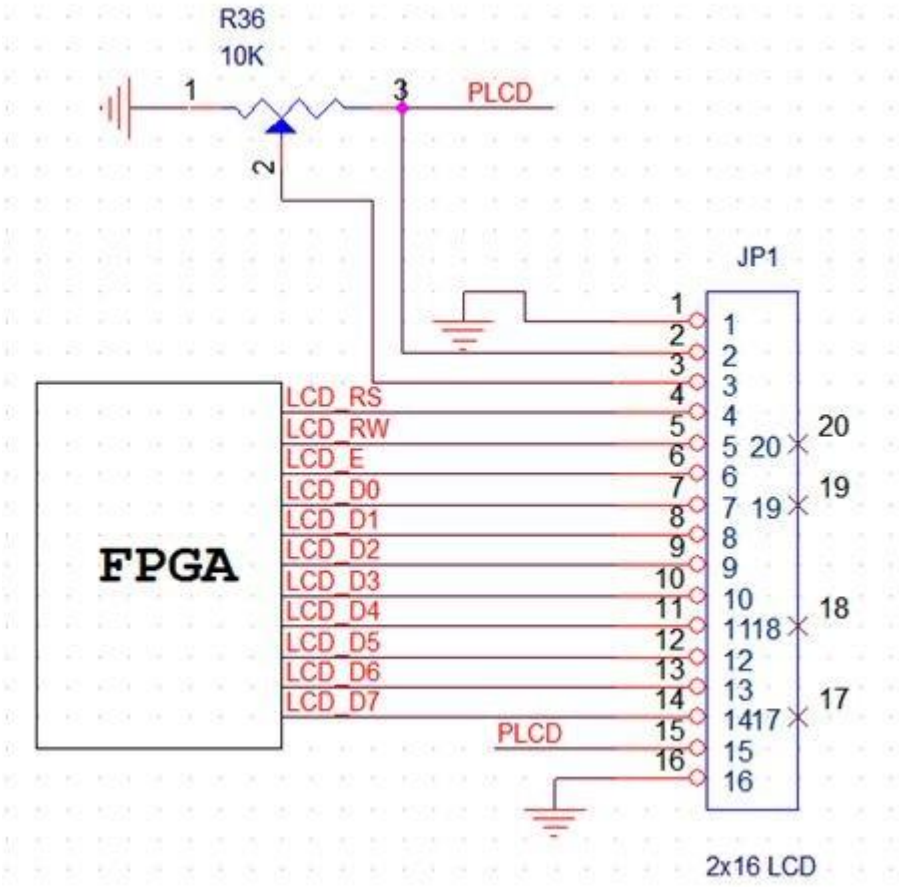
The D0-D1 is data I/O pins. The LCD is accepted the 8 bit data as a parallel form. The format of data stream is first bit must be LSB bit continue it the other bits are sent.The LCD support only ASCII value of whatever the data is.

LED backlight

The pin no's 15 & 16 are allocated for LCD backlight. The supply of LED backlight is +5v & 0v. It makes brightness of LCD display. The LED backlight pins denoted as (Anode (A), Cathode (K)) on LCD.

R/W: Read/write select control line

- High on R/W # selects the read operation
- Low on R/W # select the write operation.



LCD consist of 8- Data lines D0-D7, RS- Register Select line, RW-Read Write line, En- Enable line.

First we need to send commands to initialize the display, Curser Position, Clear Display, increment curser etc. All this command are send to instruction Register. Instruction Register can be enabled by RS = '0', RW = '1', En= '1'.

ASCI Values for Commands used in the code

38 = Function Set: 8-bit, 2 Line, 5x7 Dots

0c = Display on Cursor off

06 = Entry Mode

01= Clear Display

C0 = Place Curser to 2nd line

After sending commands, Data can be transferred to Display in the LCD. For sending Data enable Data Register by sending RS= ’1’, RW= ‘1’, En= ’1’.

VL-FPGA-B includes a LCD module, which is a dot matrix liquid crystal display that alphanumeric, kana (Japanese) characters and symbols. Built in controller provides connectivity between LCD and FPGA.

This LCD has a built in Dot Matrix Controller, with font 5×7 or 5×10 dots, display data RAM for 80 character (80×8 bit) and a character generator ROM which provides 160 character with 5×7 font and 32 character with font of 5×10.

All the functions required for LCD are provided internally. Internal refresh is provided by the Controller.

The interface details of the details of the LCD display are as shown in figure1.

DATA LINES CONNECTION

LCD has 8 bit bidirectional data bus interface to FPGA. when Enable signal is at low level, this data bus remains in high impedance state. Interface details of the data lines with SPARTAN-3 FPGA are as in below Table

Data Line Interface to SPARTAN-3 FPGA

DATA BIT	FPGA PIN
“LCD_D<0>”	P167
“LCD_D<1>”	P166
“LCD_D<2>”	P165
“LCD_D<3>”	P162
“LCD_D<4>”	P161
“LCD_D<5>”	P156
“LCD_D<6>”	P155
“LCD_D<7>”	P154

Control Line Interface to SPARTAN-3FPGA

Control bit	**FPGA Pin**
“LCD_E”	**P168**
“LCD_RS”	**P171**
“LCD_RW_BAR”	**P169**

VHDL CODE

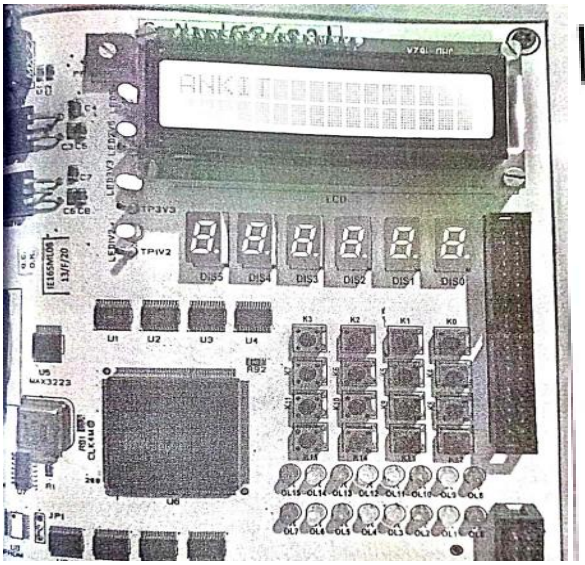
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_BIT.ALL;
--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity LCD_controller1 is
Port ( clk : in std_logic;
                              rst : in std_logic;
   lcd_data : out std_logic_vector(7 downto 0);
   lcd_en : out std_logic;
                       RW:out std_logic;
   RS : out std_logic);
  end LCD_controller1;

  architecture Behavioral of LCD_controller1 is
    signal temp:std_logic_vector(12 downto 0);
    signal slow_clk:std_logic;
    signal LUT_data:std_logic_vector(9 downto 0);
    signal LUT_index:std_logic_vector(6 downto 0):="0000000";
   begin
  PR1: process(clk,rst)
 begin

              if (rst='1') then
              temp<=(others=>'0');
              elsif (clk' event and clk='1') then
              temp<=temp+1;
              end if;
    end process;
              slow_clk<=temp(12);
     lcd_en<='1' when slow_clk='1' else '0';

    PR2:process(slow_clk,rst)
begin
     if rst='1' then
       LUT_index<="0000000";
     elsif(slow_clk'event and slow_clk='1') then
         if LUT_index<="1000000" then
                LUT_index<=LUT_index +1;
            else
                   LUT_index<="0000000";
            end if;
 end if;
```

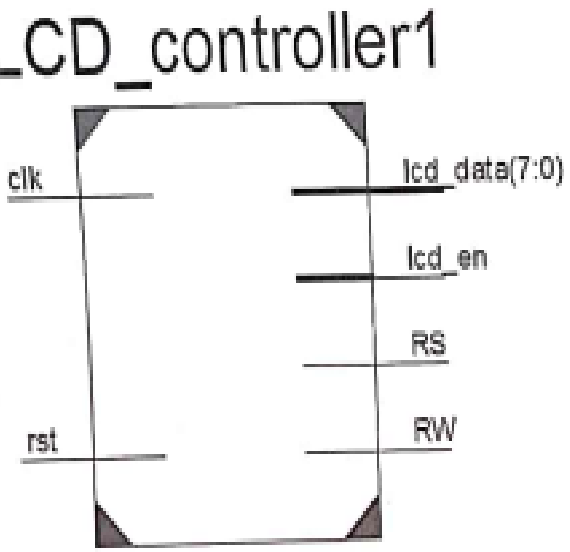
```
        end process;

      process(LUT_index)
begin
        case LUT_index is
        when "0000001"=>LUT_data<="0000111000"; --command 038(0 0011 1000)
        when "0000010"=>LUT_data<="0000001100"; --command 00C(0 0000 1100)
        when "0000011"=>LUT_data<="0000000001"; --command 001(0 0000 0001)
        when "0000100"=>LUT_data<="0000000110";         --command 006(0 0000 0110)--line 1
        when "0000101"=>LUT_data<="0010000000";         --command 080(0 1000 0000)--cursor to 1st line
        when "0000110"=>LUT_data<="0101000101";         --command 148(1 0100 1000)--E
        when "0000111"=>LUT_data<="0101011010";         --command 145(1 0100 0101)--Z
        when "0001000"=>LUT_data<="0101001001";         --command 14C(1 0100 1100)--I
        when "0001001"=>LUT_data<="0101001111";         --command 14C(1 0100 1100)--O
        when "0001011"=>LUT_data<="0011000000";         --command 0C0(0 1100 0000)--cursor to 2nd line
        when "0001100"=>LUT_data<="0101000001";         --command 150(1 0100 0100)--A
        when "0001101"=>LUT_data<="0101010101";         --command 59(1 1001 0101)--U
        when "0001110"=>LUT_data<="0101000100";         --command 50(1 0000 0111)--D
        when "0001111"=>LUT_data<="0101001001";         --command 49(1 1001 0100)--I
        when "0010000"=>LUT_data<="0101010100";         --command 45(1 0101 0100)--T
                when "0010001"=>LUT_data<="0101001111"; --command 4d(1 0010 0101)--O
                when "0010010"=>LUT_data<="0101010010"; --command 52(1 0010 0101)--R
                when others=>LUT_data<="0100000000";
        end case;
        end process;
        RW<=LUT_data(9);
        RS<=LUT_data(8);
                lcd_data<=LUT_data(7 downto 0);
      end Behavioral;
```

UCF PROGRAM-

```
net clk loc = "p181";
net rst loc = "p182";
####--------lcd----------
net lcd_data<0> loc = "p167";
net lcd_data<1> loc = "p166";
net lcd_data<2> loc = "p165";
net lcd_data<3> loc = "p162";
net lcd_data<4> loc = "p161";
net lcd_data<5> loc = "p156";
net lcd_data<6> loc = "p155";
net lcd_data<7> loc = "p154";
net RW loc = "p169";
net RS loc = "p171";
net lcd_en loc = "p168"
```

Observation:





Conclusion:

EXP - 4 KEYPAD INTERFACE

Aim: A To write VHDL code, simulate with test bench, synthesis, implement on PLD.

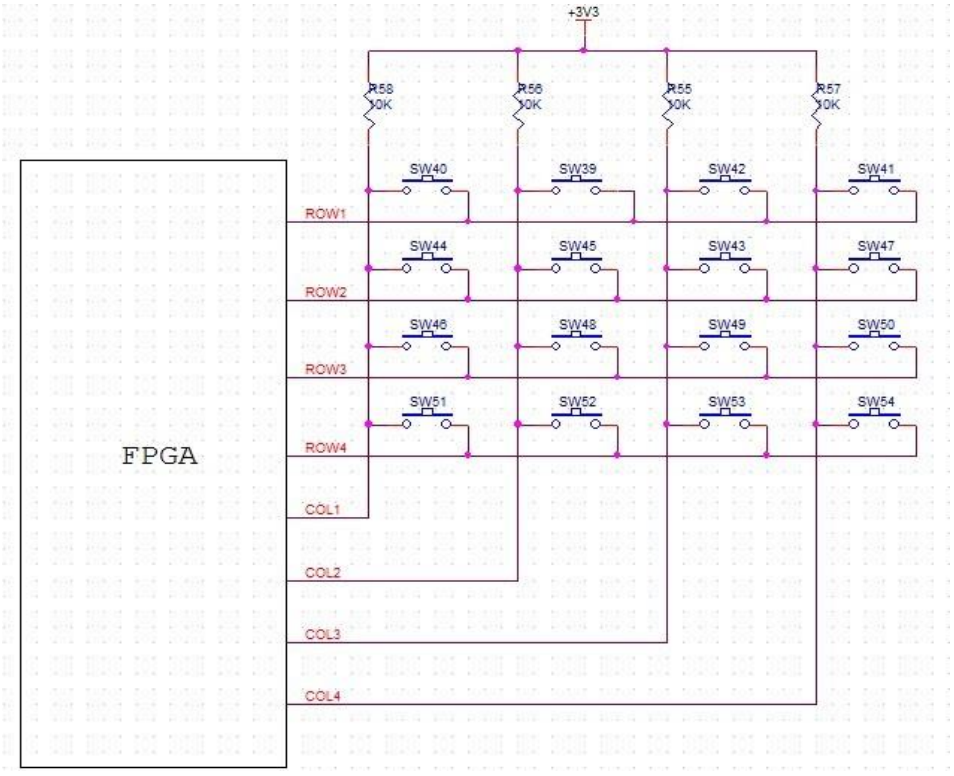
Apparatus Required:

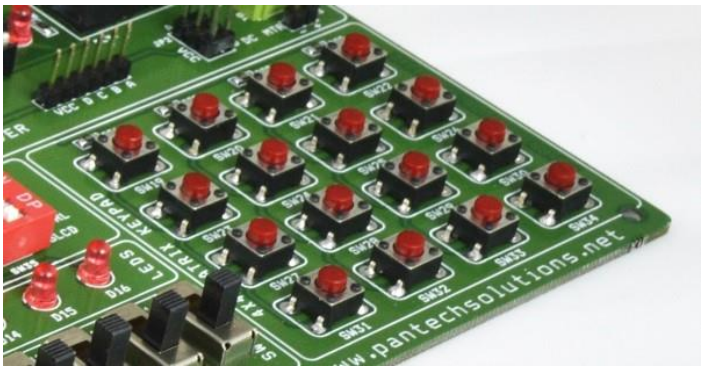
Synthesis tool: Xilinx ISE.
Simulation tool: ISim Simulator
FPGA STARTER Kit(Spartan 3)

Description:

Matrix Keypad

Keypad is most commonly used input device in electronics. To Simply number of input keypad lines to controller device leads to formation of matrix keypad. Matrix keyboard consist of NxN number of push button arranged in rows and column. 2N is the number of lines required to get the input of NxN Push Button. It consists of a set of buttons similar to an alphanumeric keyboard provided with keys usually marked with letters or numbers and various extra keys. Embedded systems which require user interaction must be interfaced with devices that accept user input such as a keypad.





VL-FPGA-B has a six multiplexed seven segment displays .Each individual character has a separate cathode input.

To light an individual signal, drive the individual segment control signal high along with that associated cathode signal for the individual character. The control signal is high, enabling the control inputs for the left-most character. The segment control inputs, A through G and DP drive the individual segment that comprise the character. A High value lights the individual segment, a Low turns off the segment.

The two types of the seven segment displays :

- **Common Cathode Displays:** In this type of display the cathode of all the LEDs are tied together and the anode terminal decides the status of the LED either ON or OFF.
- To turn ON the LED i.e segment value of driven segment should be 1 and 0 for turn OFF.
- **Common Anode Display:** In this type of display all the anode terminals of LEDs are tied together and the cathode terminals decide the status of the LED either ON or OFF.
- To turn ON the LED i.e segment value of driven segment should be 0 and 1 for turn OFF.

CLOCK & RESET

SR NO	SIGNAL NAME	PIN LOCATION
1	RESET	182
	CLK_4M	181

4×4 MATRIX KEYBOARD:

SR NO	**SIGNAL NAME**	**PIN LOCATION**
1	**RL0**	**85**
2	**RL1**	**94**
3	**RL2**	**95**
4	**RL3**	**96**
5	**SL0**	**93**
6	**SL1**	**90**
7	**SL2**	**87**
8	**SL3**	**86**

PROGRAM-

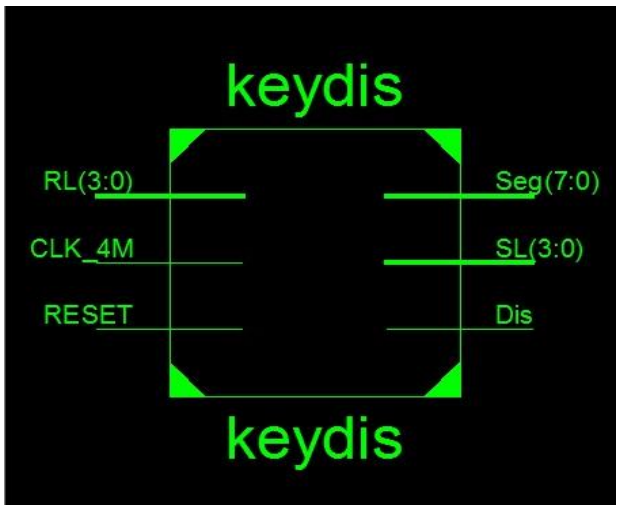
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity keydis is
Port ( RESET       : in std_logic;
       CLK_4M      : in std_logic;
-- keyboard return and scan lines
       RL          : in std_logic_vector(3 downto 0);
       SL          : out std_logic_vector(3 downto 0);
-- display i/os segments and display selects
       Seg   : out std_logic_vector(7 downto 0);
             Dis    : out std_logic);
end keydis;
----------------------------------------------------------------
architecture Behavioral of keydis is
----------------------------------------------------------------
signal Divider     : std_logic_vector(20 downto 0);
signal SClock, DClock, ORed, Interrupt    : std_logic;
signal Scan, Decoded    : std_logic_vector(1 downto 0);
signal SL_S,D,Display_s : std_logic_vector(3 downto 0);
signal segments_s   : std_logic_vector(7 downto 0);

---------------------------------------
begin
---------Clock Divider----------
process(RESET, CLK_4M)
begin
 if(RESET = '1')then
    Divider <= (others=>'0');
    elsif(CLK_4M' event and CLK_4M = '1')then
          Divider <= Divider + 1;
 end if;
end process;
-----------------Scan & Debounce clock--------------
SClock <= Divider(14);
DClock <= Divider(11);
-----------------Scan Counter----------------------
process(SClock, RESET)
begin
  if(RESET = '1')then
     Scan <= (others=>'0');
     elsif(SClock'event and SClock='1')then
           Scan <= Scan + 1;
  end if;
end process;

SL_S <= "0111" when Scan = "00" else
        "1011" when Scan = "01" else
        "1101" when Scan = "10" else
        "1110" ;

SL <= SL_S;
```

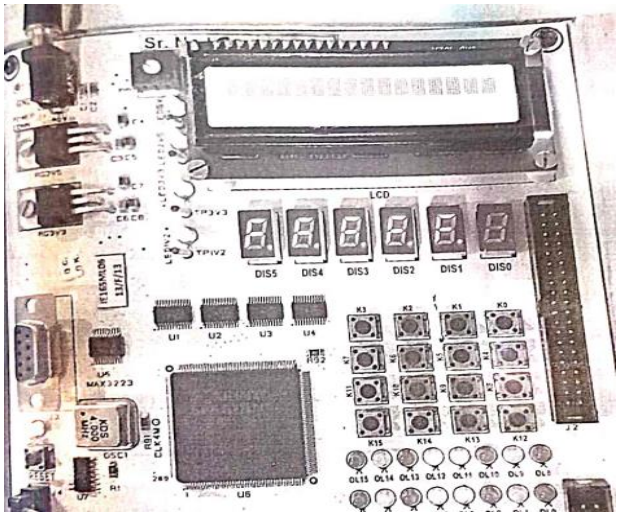
```
ORed <= (not RL(0)) or (not RL(1)) or (not RL(2)) or (not RL(3));
process(RESET,DClock)
begin
  if(RESET = '1')then
     D <=(others => '0');
     elsif(DClock' event and DClock = '1')then
          D <= D(2 downto 0) & ORed;
  end if;
end process;
Interrupt <= '1' when(D = "1111")else
           '0';
Decoded <= "00" when (RL = "0111") else
          "01" when (RL = "1011") else
          "10" when (RL = "1101") else
          "11";
process(Interrupt, RESET)
begin
  if(RESET = '1')then
     Display_s <= (others=>'0');
     elsif(Interrupt'event and Interrupt = '1')then
          Display_s <= Scan & Decoded;
  end if;
end process;
Segments_s     <= "01000000" when Display_s ="0000" else --0 -- k0
                              "01111001" when Display_s ="0001" else --1 -- k4
                              "00100100" when Display_s ="0010" else --2 -- k8
                              "00110000" when Display_s ="0011" else --3 -- k12
                              "00011001" when Display_s ="0100" else --4 -- k1
                              "00010010" when Display_s ="0101" else --5 -- k5
                              "00000010" when Display_s ="0110" else --6 -- k9
                              "01111000" when Display_s ="0111" else --7 -- k13
                              "00000000" when Display_s ="1000" else --8 -- k2
                              "00010000" when Display_s ="1001" else --9 -- k6
                              "00001000" when Display_s ="1010" else --A -- k10
                              "00000011" when Display_s ="1011" else --B -- k14
                              "01000110" when Display_s ="1100" else --C -- k3
                              "00100001" when Display_s ="1101" else --D -- k7
                              "00000110" when Display_s ="1110" else --E -- k11
                              "00001110" when Display_s ="1111" else --F -- k15
                              "01000000";
Seg <= segments_s;
Dis <= '1';
-- test input switches and output leds
--OL <= IL;
end Behavioral;
```

UCF

```
net CLK_4M loc = "p181";
net RESET loc = "p182";
####--------keypad----------
net RL<0> loc = "p85";
net RL<1> loc = "p94";
net RL<2> loc = "p95";
net RL<3> loc = "p96";
net SL<0> loc = "p93";
net SL<1> loc = "p90";
net SL<2> loc = "p87";
net SL<3> loc = "p86";
net  Seg<0> loc = "p144";
net  Seg<1> loc = "p143";
net  Seg<2> loc = "p141";
net  Seg<3> loc = "p140";
net  Seg<4> loc = "p139";
net  Seg<5> loc = "p138";
net  Seg<6> loc = "p137";
net  Seg<7> loc = "p135";
net  Dis loc = "p133";
```

Observation:





Conclusion: