COMPUTER ARCHITECTURE

LAB MANUAL

PREPARED BY

AVIJIT BOSE

SOMA BANDYOPADHYAY

Contents

INTRODUCTION

VHDL is a hardware description language. It describes the behavior of an electronic circuit or system from which the physical circuit or system can be attained or implemented.

What does VHDL stands for?

VHDL stands for VHSIC hardware description language. VHSIC is an abbreviation for very high speed integrated circuit.

VHDL is intended for circuit synthesis as well as circuit simulation, however all constructs are not synthesizable. VHDL is a standard, technology/ vendor independent language and is therefore portable and reusable.

Application of VHDL lies in the following area

- CPLD :- Complex Programmable logic device
- ASIC :- Application Specific Integrated Circuit
- FPGA :- Field Programmable GATE Array

It should be taken into consideration that once the VHDL code has been written it can be used to implement the circuit in a programmable device (e.g. Altera, Xilinx etc.).

Why VHDL is referred as code?

VHDL is referred to as code because it is not sequential but rather it is concurrent (parallel). However it may be considered that statements placed inside a process, function or procedure are executed sequentially.

One of the major utilities of VHDL is that it allows the synthesis of circuit or system in a programmable device (PLD/FPGA).

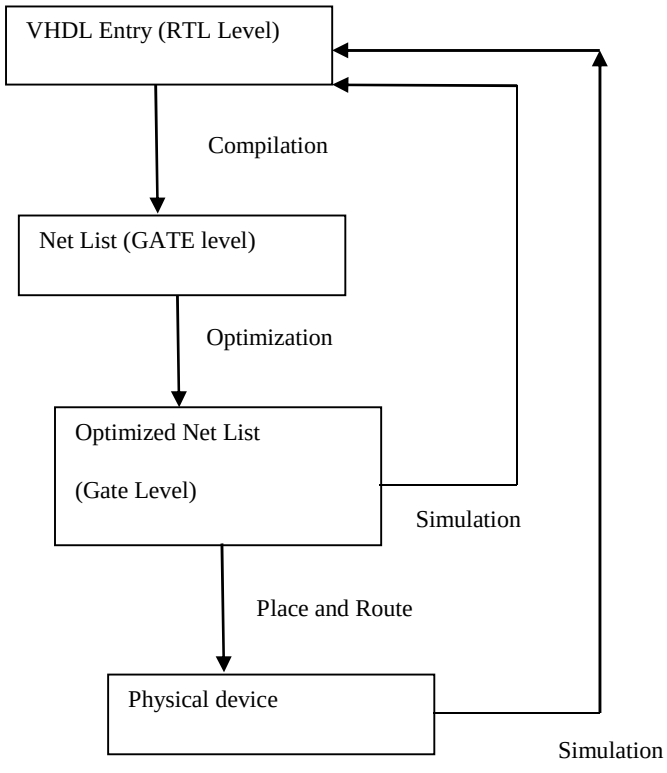


Figure 1-. Figure depicting the execution sequence

Writing the first code:-

Let us consider the following circuit for Full Adder circuit

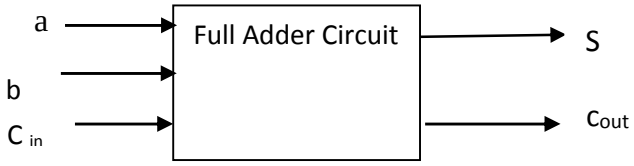


Figure-2 Full Adder Circuit

and the truth table will look as follows:-

A	B	cin	S	cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

The code now will look as follows

```
Entity full_adder is
      Port( a,b,cin : IN bit ;
            S, cout : out bit);
End full_adder;
Architecture dataflow of full_adder is
begin
  s<= a XOR b XOR cin;
cout<= ( a AND b) OR (a AND cin) OR(b AND cin);
end dataflow;
```

CHAPTER-1 STRUCTURAL CODE

1.1 VHDL Basics

VHDL is a hardware description language that can be used to model a digital system. The digital system can be as simple as a logic gate or as complex as a complete electronic system. A hardware abstraction of this digital system is called an entity in this text. An entity X, when used in another entity Y, becomes a component for the entity Y. Therefore a component is also an entity, depending on the level at which you are trying to model.

To describe an entity, VHDL provides five different types of primary constructs, called design units. They are:

1. Entity declaration
2. Architecture body
3. Configuration declaration
4. Package declaration
5. Package body

An entity is modeled using an entity declaration and at least one architecture body. The entity declaration describes the external view of the entity; for example, the input and output signal names. The architecture body contains the internal description of the entity; for example, as a set of concurrent or sequential statements that represents the behavior of the entity.

A configuration declaration is used to create a configuration for an entity. It specifies the binding of one architecture body from many architecture bodies that may be associated with the entity. It may also specify the bindings of components used in the selected architecture body to other entities.

A package declaration encapsulates a set of related declarations, such as type declarations, subtype declarations, and subprogram declarations, which can be shared across two or more design units. A package body contains the definitions of subprograms declared in a package declaration.

1.2 Fundamental VHDL units

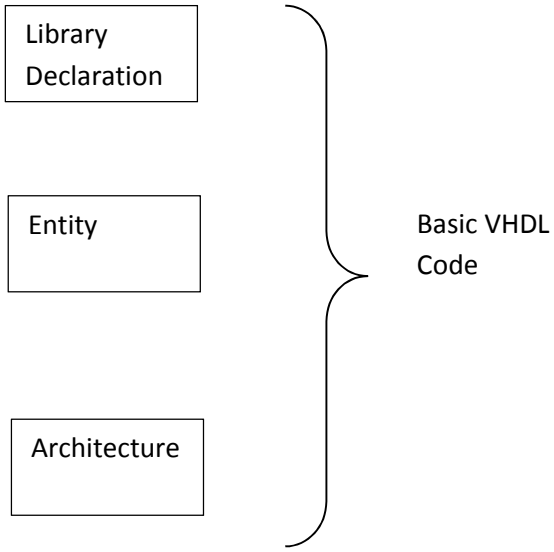


Figure-3 Basic VHDL code

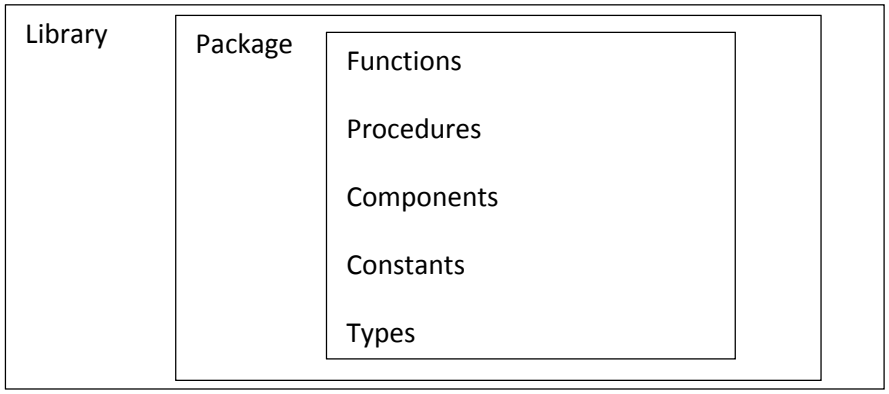


Figure-4 Fundamental parts of a Library

VHDL Code basically comprises of three parts

1) Library
2) Entity
3) Architecture

1.2.1 Library

Contains a list of libraries to be used in the design e.g.

a) ieee
b) std
c) work etc.

1.2.2 Entity

Specifies the I/O pins of the circuit.

1.2.3 Architecture

Contains the VHDL code which describes how the circuit should behave.

1.3 How to declare Library:

A Library is a commonly used piece of code. Placing such pieces inside a library allows them to be reused or shared by other designs. To declare a Library (that is to make it visible to the design) 2 lines of code are needed, one containing the name of the library, and the other use a clause as shown below:

Library library_name;
Use library_name.package_name.package_parts;

At least 3 packages from 3 different libraries are needed

a) ieee.std_logic_1164 (from ieee)
b) standard (from the std library)
c) work (work library)

The declaration will look as follows

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

Note:- The libraries std and work are made visible by default so no need to declare them. Only the IEEE library must be explicitly written.

However IEEE library should be necessary when std_logic or standard_ulogic data type is employed in the design.

Std_logic_1164 of IEEE library specifies multilevel logic system.

std is a resource library (data types, text i/o) work library is where we save our design.

ieee library consists of :

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

1.3.1 Purpose of IEEE library packages

(i) Std_logic_1164:- specifies the std_logic (8 levels) and std_ulogic (9 levels) for multivalued logic systems.

(ii) std_logic_arith:- specifies the signed and unsigned data types and related arithmetic and comparison operation. It also contains several data conversion function which allows one type to be converted to another for example conv_integer(p) which converts a parameter p to a type integer. Conv_unsigned(p,b) converts a parameter p to unsigned value of b bits.

(iii) std _logic_unsigned:- contains functions that allow operations with std_logic_vector to be performed as if data were of type unsigned.

1.4 Entity

An entity is a list with specification of all input and output pins (PORTS) of the circuit. Its syntax is shown as below

```
ENTITY entity_name IS
    PORT (
        port_name : signal_mode signal_type;
        port_name : signal_mode signal_type;
    ……..)
    END entity name;
```

the mode of the signal can be IN,OUT,INOUT, BUFFER. IN and OUT are truly unidirectional pins while INOUT is bidirectional. BUFFER is employed when the output signal must be used (read) internally. The type of the signal can be BIT, STD_LOGIC, and INTEGER etc. Name of the entity can be basically any name except VHDL reserved words.

ENTITY nand_gate IS

```
PORT (a, b: IN BIT;
        x: OUT BIT);
END nand_gate;
```

All three signals data type is of data type BIT.

1.5 Architecture

Architecture is a description of how the circuit should behave and its syntax is as follows:-

```
ARCHITECTURE architecture_name of entity_name IS
    [ declaration]
BEGIN
    (code)
END architecture_name;
```

Note that declarative part is optional where signals and constants are declared and the code part is written down from BEGIN. For example

```
ARCHITECTURE XYZ OF nand_gate IS
BEGIN
    X <= a NAND b;
END XYZ;
```

CHAPTER-2 DATA TYPES

In order to write VHDL code efficiently its essential to know what data types are allowed.

2.1 Pre-defined data types

IEEE 1076 and IEEE 1164 specify pre defined data types. Let's see what are the libraries and the corresponding packages and data types associated with it.

Library	Package	Data Types
Std	Standard	Bit, Boolean, integer and real
ieee	Std_logic_1164	STD_LOGIC, STD_ULOGIC
ieee	Std_logic_arith	SIGNED and UNSIGNED data type plus several data conversion function like conv_integer(p), conv_unsigned (p,b) Conv_signed(p,b) and conv_std_logic_vector(p,b)
ieee	Std_logic_signed Std_logic_unsigned	STD_LOGIC_VECTOR as if the data is of type signed or unsigned.

Pre-defined data types are described below

BIT (BIT_VECTOR): 2 level logic ('0','1')

Examples:

SIGNAL x: BIT

- X is declared as a one digit signal of type BIT

SIGNAL y: BIT_VECTOR (3 DOWNTO 0)

- Y is a 4 bit vector with the leftmost bit being the MSB.

SIGNAL w: BIT_VECTOR (0 to 7)

w is a 8 bit vector with the rightmost bit being the MSB.

x<='1';

x is a single bit signal whose value is '1' note that single quotes are used for a single bit.

y<="0111"

y is a 4 bit signal whose value is "0111" (MSB='0'). Note that double quotes are used for vectors.

w<="01110001";

w is a 8 bit signal whose value is "01110001" (MSB='1')

Some of the examples are given below:-

SIGNAL x: STD_LOGIC;

- x is declared as a one digit (scalar) signal of type STD_LOGIC

SIGNAL y: STD_LOGIC_VECTOR (3 down to 0):=”0001”;

- y is declared as a 4 bit vector with the leftmost bit being the MSB. The leftmost bit being the MSB. The initial value of y is “0001” Note that “:=” operator is used to establish the initial value.

Some of the data types and their range

(i) Boolean :- True/false
(ii) Integer :- 32 bit integer (ranges from -2,147,483,647 to +2,147,483,647)
(iii) Natural: - Non negative integers (0 - +2,147,483,647).
(iv) Real :- Real numbers ranging from -1.0E38 to +1.0E38
(v) Physical literals: Used to inform physical quantities like time, voltage etc.
(vi) Character literals:- Single ASCII character or a string of such characters.
(vii) SIGNED and UNSIGNED: - data types defined in the std_logic_arith package of the IEEE library. They have the appearance of STD_LOGIC_VECTOR, but accept arithmetic operations which are typical of INTEGER data types.

Some of the examples are given below:-

X0<=’0’; // std _logic, std_ulogic, bit value ‘0’

X1<=”00011111”

//bit_vector, std_logic_vector, std_ulogic_vector, signed or unsigned

X2 <=”0001_1111”;

// underscore allowed to ease visualization

X3<=”101111”

// binary representation of decimal 47

X4<=B”101111”

// binary representation of decimal 47

X5<=o”57”

// octal representation of decimal 47

X6<=X”2F”

// hexadecimal representation of decimal 47

N<=1200

//integer

M<=1_200

// integer underscore allowed

If ready then…..

// Boolean and executed if ready=true

Y<=1.2E-5

//real, not synthesizable

Q<=d after 10ns

// physical not synthesizable

Next let us take an example and see whether operation between different data types is legal or illegal

SIGNAL a : BIT;

SIGNAL b: BIT_VECTOR (7 DOWNTO 0)

SIGNAL c : STD_LOGIC;

SIGNAL d : STD_LOGIC_VECTOR (7 downto 0)

SIGNAL e : INTEGER range 0 to 255;

……

a<=b(5)

allowed because same data type bit.

b(0) <=a;

legal same scalar type BIT

c<=d(5)

legal same scalar type STD_LOGIC

d(0) <=c;

legal same scalar type STD_LOGIC

a<=c

illegal type mismatch between BIT and STD_LOGIC

b<=d

illegal type mismatch between bit_vector and std_logic vector

e<=b

illegal type mismatch between integer and bit vector

e<=d

illegal type mismatch between integer and std_logic_vector.

2.2 User-Defined Data Types

VHDL allows the user to define user his own data type. Two categories of user defined data types are shown as integer and enumerated.

- User defined integer type

TYPE integer IS RANGE -2147483647 TO +2147483647

// pre-defined type INTEGER

TYPE natural IS RANGE 0 TO +2147483647

// predefined type natural

TYPE my_ integer IS RANGE -32 to 32

// a user defined subset of integers

TYPE student_ grade IS RANGE 0 TO 100

// a user defined subset of integers or naturals

TYPE bit IS (‘0’ , ‘1’);

// this is indeed the pre-defined type Bit

TYPE my_logic IS (‘0’,’1’,’Z’)

// A user –defined subset of std_logic

2.3 Subtypes

A SUBTYPE is a TYPE with a constraint. The main reason for using a subtype rather than specifying a new type is that , though operations between data of different types are not allowed , between a subtype and its corresponding base type.

SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER’HIGH

// as expected NATURAL is a subtype (subset) of INTEGER

Let us see an example between legal and illegal operation between types and subtypes

SUBTYPE my_logic IS STD_LOGIC RANGE ‘0’ TO’1’

SIGNAL a: BIT

SIGNAL b: STD_LOGIC

SIGNAL c: my_logic

.....

b<=a;

// illegal BIT versus STD_LOGIC

b<=c;

// legal (same “base” type: STD_LOGIC

2.4 Arrays

Arrays are collection of objects of the same type. They can be one-dimensional (1D) , two-dimensional (2D) or one-dimensional-by-one dimensional (1DX1D). They can also be higher dimension, but then they are generally not synthesizable. The pre-defined synthesizable types in each of these categories are the following:-

Scalars: - BIT, STD_LOGIC,STD_ULOGIC and BOOLEAN.

Vectors:BIT_VECTOR,STD_LOGIC_VECTOR,STD_ULOGIC_VECTOR,INTEGER,SIGNED AND UNSIGNED.

The declaration is as follows:-

TYPE type_name IS ARRAY (specification) OF data_type;

SIGNAL signal_name: type_name := initial_value;

TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;

// 1D Array

TYPE matrix IS ARRAY (0 TO 3) OF row;

//1DX1D array

SIGNAL x : matrix;

// 1Dx1D array

TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR (7 DOWNTO 0)

From a data –compatibility point of view the latter might be advantageous over the previous example.

Example: - 2D Array

The array below is truly 2 dimensional. Note that its construction is not based on vectors, but rather entirely on scalars.

TYPE MATRIX2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;

- 2D Array

Example: Array initialization

Signal x: =”0001”

// this is the assignment for 1D array

Signal x :=(‘0’,’0’,’0’,’1’);

// this is the assignment for 1D array

Signal y:=((‘0’,’1’,’1’,’1’),(‘1’,’1’,’1’,’0’))

// 2 dimensional array initialization.

Note:- Here it must be remembered that assignment is possible only when data types are same as well as dimensional are also same.

2.5 Signed and Unsigned Data Types

As presented signed and unsigned data types are defined in the std_logic_arith package of ieee library. Their syntax is described as follows:-

SIGNAL x: SIGNED (7 DOWNTO 0)

SIGNAL y : UNSIGNED (0 TO 3)

An unsigned value is a number lower than 0. For example “0101” represents the decimal 5 while”1101” signifies 13. However if type signed is used the value can be positive or negative. Therefore “0101” would represent the decimal 5 while “1101” would mean -3. To use signed and unsigned data types are mainly intended for arithmetic operations, that is, contrary to STD_LOGIC_VECTOR, they accept arithmetic operations. On the other hand logical operations are not allowed. However comparison operation there is no restriction.

Example: - Legal and illegal operation with signed/unsigned data types

LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

SIGNAL a : IN SIGNED (7 DOWNTO 0);

SIGNAL b: In SIGNED (7 DOWNTO 0)

SIGNAL x : OUT SIGNED(7 DOWNTO 0);

…….

v<= a+b; // legal arithmetic operation ok

w <= a AND b;//illegal

Example:- Legal and illegal operation with std_logic_vector

LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

SIGNAL a : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0)

SIGNAL x : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);

…….

v<= a+b; // illegal arithmetic operation ok

w <= a AND b;//legal

But how can we make the second example to work perfectly its simple just add two extra packages.

LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.std_logic_unsigned.all;

2.6 Data Conversion

VHDL does not allow direct operations (arithmetic, logical etc) between data of different types. Therefore it is often necessary to convert data from one type to another. This can be done in two ways by writing a piece of VHDL code or if we invoke a function from a pre-defined package which is capable of doing this.

Let us look at the following subset example and see how it happens

TYPE long IS INTEGER RANGE -100 TO 100;

TYPE short IS INTEGER RANGE -10 TO 10;

SIGNAL x : short

SIGNAL y : long

……

y <= 2*x + 5;// error

In order to avoid the above error we should write

y<= long (2* x + 5)

Several data conversion functions can be found in the std_logic_arith package of the ieee library. They are as follows:-

(i) conv_integer(p):- converts a parameter p of type INTEGER,UNSIGNED,SIGNED, STD_ULOGIC to an INTEGER value.

(ii) conv_unsigned(p,b):- Converts a parameter p of type INTEGER, UNSIGNED, SIGNED or STD_ULOGIC to an unsigned value with size b bits.

(iii) conv_signed(p,b):- Converts a parameter p of type INTEGER, UNSIGNED,SIGNED or STD_ULOGIC to a SIGNED value with size b bits.

(iv) conv_ std_logic_vector(p,b):- Converts a parameter p of type INTEGER, UNSIGNED,SIGNED or STD_LOGIC to a STD_LOGIC_VECTOR value with size b bits.

Let us see an example of this data conversion

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

……..

SIGNAL a: IN UNSIGNED ( 7 DOWNTO 0)

SIGNAL b: IN UNSIGNED (7 DOWNTO 0)

SIGNAL y : OUT STD_LOGIC_VECTOR ( 7 DOWNTO 0);

y <= conv_STD_LOGIC_VECTOR((a+b),8);
```

// Legal operation a+b is converted from unsigned to a 8 bit STD_LOGIC_VECTOR value and then assigned to y. Alternatively we can add std_ logic_signed or std_logic_unsigned packages from the ieee library and such packages allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED respectively.

CHAPTER-3 OPERATORS AND ATTRIBUTES

VHDL provides several kinds of pre-defined operators:-

(a) Assignment operator
(b) Logical operator
(c) Arithmetic operator
(d) Relational operator
(e) Shift operator
(f) Concatenation operator

3.1 Assignment operator

These are used to assign values to signals, variables or constants. They are

<= used to assign values to a signal.

:= used to assign value to a variable, constant or generic. Used also for establishing initial values.

= > used to assign values to individual vector elements or with others.

Let us see the following example

SIGNAL x : STD_LOGIC

VARIABLE y : STD_LOGIC_VECTOR (3 DOWNTO 0)

// leftmost bit is MSB

SIGNAL w: STD_LOGIC_VECTOR (0 TO 7)

Then the following assignments are legal:

X<='1';

Y:="0000"

W<="10000000"

W<=(0=>'1' ,OTHERS=>'0');

3.2 Logical operator

Used to perform logical operation. The data must be of type BIT, STD_LOGIC, STD_ULOGIC. The logical operators are:-

(i) NOT
(ii) AND
(iii) OR
(iv) NAND
(v) NOR
(vi) XOR
(vii) XNOR

Note that operators have been put together as per precedence. However students are advised to try with XNOR and check whether that is working.

Examples:-

Y<= NOT a AND b; (a'.b)

Y <= NOT (a and b); (a.b)'

Y<= a NAND b;

3.3 Arithmetic operator

As mentioned earlier that data can be of type integer, signed, unsigned or real. However real data types cannot be synthesized. If the std_logic_signed or std_logic_unsigned package of the ieee library is used then STD_LOGIC_VECTOR can be employed directly in addition and subtraction operation.

+ Addition

--Subtraction

* Multiplication

/ divison

** Exponentiation

MOD modulus

REM remainder

ABS absolute value

However there are no synthesis restriction regarding addition and subtraction and the same is true for multiplication. For divison only power of 2 dividers (shift operation) are allowed. For exponentiation only static values of base and exponent are accepted. y mod x returns the remainder of y/x with the signal of x, while y rem x returns the remainder of y/x with the signal of y. abs returns the absolute value. However mod, rem, abs there is no synthesis support.

3.3 Comparison operator

The relational operators are
= equal to
/= not equal to

< less than

- Greater than

<= less than or equal to

>= greater than or equal to

3.4 Shift operator

The general syntax is as follows

< left operand><shift operation> < right operand>

The left operand must be of type vector and the right operand must be of type integer.

Sll → shift left logic; positions on the right are filled with 0

Srl → shift right logic; positions on the left are filled with 0

Sla → shift left arithmetic ; rightmost bit is replicated on the right.

Sra → shift right arithmetic; leftmost bit is replicated on the left.

Rol → rotate left logic

Ror→ rotate right logic

3.5 Concatenation operator

.&

.(,,,)

Example

Z<= x & “1000000”;

If x=1 then z = 11000000;

Z<=(‘1’,’1’,’0’,’0’,’0’,’0’,’0’,’0’)

Z= “ 11000000”

3.5 Attributes

Attributes are divided into two parts

(a) Data attribute:- returns information (a value) regarding a data vector.
(b) Signal attribute:- Serve to monitor a signal.(return true or false).

CHAPTER-4 CODE CONCURRENCY

Code concurrency is achieved when we use the following

(a) Operators
(b) WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN)
(c) GENERATE statement
(d) Block statement.

4.1 With Operators

Just when we use the logical, arithmetic, comparison, shift or

Concatenation operator to generate the concurrent code. Example

Y<= (a AND NOT s1 and NOT s0) OR…etc

4.2 WHEN (simple and selected)

4.2.1 WHEN/ELSE syntax

Assignment WHEN condition ELSE

Assignment WHEN condition ELSE

…..;

4.2.2 WITH/SELECT/WHEN

WITH identifier SELECT

Assignment WHEN value,

Assignment WHEN value,

……;

Example with WHEN/ELSE

Outp<=”000” WHEN (inp=’0’ OR reset=’1’)ELSE

…….

Example with SELECT/WHEN

WITH control SELECT

Output<=”000” WHEN reset;

…….

4.3 Generate

It follows the same feature as sequential code loop i.e it allows the section of code to be repeated number of times thus creating several instances of the same assignment.

Syntax :-

FOR/GENERATE

Label: FOR identifier IN range GENERATE

(concurrent assignments)

END GENERATE;

Syntax:-

IF/GENERATE

Label1: For identifier in range GENERATE

…….

Label2: IF CONDITION GENERATE

(concurrent assignments)

END GENERATE;

……

END GENERATE;

Example:-

SIGNAL x: BIT_VECTOR(7 DOWNTO 0)

SIGNAL x: BIT_VECTOR(15 DOWNTO 0)

SIGNAL x: BIT_VECTOR(7 DOWNTO 0)

G1: FOR I IN x' GENERATE

Z(i) <=x(i) AND y(i+8);

END GENERATE;

4.4 BLOCK

4.4.1

The block statement in its simplest form represents only a way of locally partitioning the code. It allows a set of concurrent statements to be clustered into a BLOCK.

General Syntax is:-

label : BLOCK

[declarative part]

BEGIN

(concurrent statements)

END BLOCK label;

An example of the BLOCK is

b1 : BLOCK

SIGNAL a : STD_LOGIC;

BEGIN

A<= input_sig when ena =’1’ else ‘z’;

End block b1;

4.4.2 Guarded Block

A guarded block is a special kind of block which includes an additional expression called guard expression. A guarded statement in a guarded block is executed only when the guard expression is true.

Guarded Block:-

Label : BLOCK (guard expression)

(declarative part)

BEGIN

(concurrent guarded and unguarded statements)

END BLOCK label;

Example:-

B1=BLOCK(clk’EVENT AND CLK=’1’)

BEGIN

Q<= GUARDED ‘0’ WHEN rst=’1’ ELSE d;

CHAPTER-5 SEQUENTIAL CODE

Till now we have seen that VHDL code is concurrent. However PROCESSES, FUNCTIONS and PROCEDURES are the only sections of code that are executed sequentially. Another important concept regarding sequential code is that it is not limited to sequential logic indeed we can build sequential or combinational circuit or both together. Note that IF, WAIT, CASE and LOOP are all sequential and are allowed inside PROCESSES, FUNCTIONS and PROCEDURES.

5.1 PROCESS

A process is sequential section and is characterized by the presence of IF, WAIT, CASE or LOOP by a sensitivity list. Sensitivity list is the same concept as arguments in functions that is used in high level language.

The syntax of PROCESS is as follows:-

[label:] PROCESS (sensitivity list)

[VARIABLE name type [range] [:= initial_value;]]

BEGIN

(sequential code)

END PROCESS [label];

Example:-

PROCESS (clk,rst)

BEGIN

IF (rst = ‘1’) THEN

Q<=’0’;

……

5.2 Difference between Signals and Variables

For passing non static values signals and variables are used. A signal can be declared in a package, entity or architecture. A variable can only be declared in a piece of sequential code i.e process for example.

5.3 IF syntax

The general syntax of IF is

IF condition then assignments;

ELSIF condition THEN assignments;

……

ELSE assignments;

END IF;

5.4 WAIT

The operation of WAIT is similar to that of IF. However more than one form of WAIT is available. There are 3 basic syntax what we follow:-

WAIT UNTIL signal_condition;

WAIT ON SIGNAL [,signal2,…];

WAIT for time;

Example:-

Wait until (clk'event and clk='1')…

Wait on clk,rst;

5.5 CASE

Case is another statement intended exclusively for sequential code. Its syntax is as follows:-

CASE identifier IS

WHEN value => assignments;

WHEN value=> assignments;

END CASE;

Example is:-

CASE control IS

WHEN "00" => x<=a;y<=b;

END CASE;

5.6 LOOP

As the name says LOOP is useful when a piece of code must be instantiated several times like IF,WAIT and CASE. There are several ways of using LOOP and the syntax looks as below:-

FOR loop:-

[label:] FOR identifier IN range LOOP

(sequential statements)

END LOOP [label];

WHILE loop:-

[label:] WHILE condition LOOP

(sequential statements)

END LOOP[label];

EXIT:- is used for ending the loop.

[label :] EXIT [label] [WHEN condition];

NEXT:- used for skipping loop steps.

[label:] NEXT[loop_label][WHEN condition]

Example of FOR/LOOP

For I IN 0 TO 5 LOOP

X(i)<= enable AND w(i+2);

Y(0,i)<=w(i);

END LOOP;

Example of WHILE LOOP

WHILE (i<10) LOOP

WAIT until clk'event and clk='1';

……..

END LOOP;

Example of NEXT

FOR I in 0 TO 15 LOOP

NEXT WHEN i=skip;

……

END LOOP;

5.7 DIFFERENCE BETWEEN CASE AND IF

The codes below show the difference in the implementation of the multiplexer circuit.

IF(sel="00") THEN x<=a;

ELSIF(sel="01") THEN x<=b;

ELSIF(sel="10") THEN x<=c;

ELSE x<=d;

The implementation of the same program with CASE is

CASE sel IS

WHEN "00"=> x<=a;

WHEN "01" => x<=b;

```
WHEN “10” => x<=c;

WHEN OTHERS => x<=d;

END CASE;
```

5.8 DIFFERENCE BETWEEN CASE AND WHEN

While WHEN is concurrent the CASE is sequential. Now let’s see the code with WHEN

```
WITH sel SELECT

        x <= a WHEN “000”

             b WHEN “001”

             c WHEN “010”

……..
```

The same program while being implemented with case will look as follows:-

```
                        CASE sel IS

                        WHEN “000”=> x<=a;

                        WHEN “001” => x <=b;

……..
```

CHAPTER-6 VARIABLES AND SIGNALS

6.1 Where can be used?

CONSTANT and SIGNAL can be global and can be used in either type of code concurrent or sequential. A VARABLE on the other hand is local for it can be used inside a piece of sequential code i.e. in a PROCESS, FUNCTION or PROCEDURE and its value can never be passed out directly.

6.2 CONSTANT

The syntax for CONSTANT is as follows:-

CONSTANT name: type: =value;

Example:

```
CONSTANT set_bit : BIT :='1';
CONSTANT datamemory : memeory :=(('0','0','0','0'),('0','0','0','1'),…);
```

A CONSTANT can be declared in a PACKAGE, ENTITY or ARCHITECTURE. When declared in an entity it is global to all architectures that follow that entity. Finally when declared in an architecture it is global to that architectures code only. Generally CONSTANT is declared in an ARCHITECTURE or in a PACKAGE.

6.3 SIGNAL

SIGNAL serves to pass values in and out the circuit as well as between internal circuits. For example all ports of an entity are signals by default. The syntax for signal are as follows:-

SIGNAL name: type (range) (:=initial value);

Example:-

```
SIGNAL control: BIT: ='0';
SIGNAL count: INTEGER RANGE 0 TO 100;
```

The declaration of a SIGNAL can be made in the same place as the declaration of a CONSTANT. However it must be noted that the value of a signal is not updated unless the PROCESS, FUNCTION or PROCEDURE gets completed. It is not synthesizable and will only be considered during simulation.

6.4 Variable

Compared to CONSTANT and SIGNAL a VARIABLE represents only local information. It can only be used inside a PROCESS, FUNCTION or PROCEDURE. Its update is immediate so the new value can be promptly used in the next line of code.

To declare a VARABLE the following syntax should be used:

VARIABLE name: type [range][:= init_value];

Example:-

VARABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) :="10001000";

Exercise Left for the Learner:-

1. What is the difference between SIGNAL and VARIABLE?

CHAPTER-7 FUNCTION AND PROCEDURE

7.1 What are Functions and Procedures?

Functions and Procedures are collectively called subprograms. From a construction point of view they are very similar to a PROCESS for they are the only pieces of sequential VHDL code , and thus employs the same sequential statements seen (IF,CASE,LOOP,WAIT is not allowed) . Where the PROCESS and FUNCTION are intended for immediate use in the main code, others are intended mainly for LIBRARY allocation.

7.2 FUNCTION

A FUNCTION is a section of sequential code. Its purpose is to create new functions to deal with commonly encountered problems, like data type conversions, logical operations, arithmetic computations and new operators and attributes. By writing such code as a FUNCTION , it can be shared and reused, also propitiating the main code to be shorter and easier to understand.

The syntax for the FUNCTION are shown as below

FUNCTION function_name [<parameter list >] return data_type IS

[declarations]

BEGIN

[sequential statements]

END function_name;

An example is shown as below

Function f1 (a, b: INTEGER; SIGNAL c : STD_LOGIC_VECTOR)

RETURN BOOLEAN IS

BEGIN

(sequential statements)

END f1;

7.2.1 Function call

A function is called as part of an expression. The expression can appear by itself or associated to a statement.

Example:-

X<= conv_integer (a);

7.3 PROCEDURE

A PROCEDURE is very similar to a FUNCTION and has the same basic purpose. However the procedure can return more than one value. Like a FUNCTION two parts are necessary to construct and use a PROCEDURE: the procedure itself (procedure body) and a procedure call.

The syntax for Procedure will look as follows:-

PROCEDURE procedure_name [<parameter list>] IS

[declarations]

BEGIN

(sequential statements)

END procedure_name;

Example :-

PROCEDURE my_procedure (a: IN BIT; SIGNAL b,c: IN BIT; SIGNAL x : OUT BIT_VECTOR (7 DOWNTO 0);

SIGNAL y : INOUT INTEGER RANGE 0 TO 99) IS

BEGIN

……

END my_procedure;

7.3.1 PROCEDURE CALL

A PROCEDURE call is a statement on its own. It can appear by itself or associated to a statement.

Example of PROCEDURE call

compute_min_max(in1,in2,in3,out1,out2);

// statement by itself

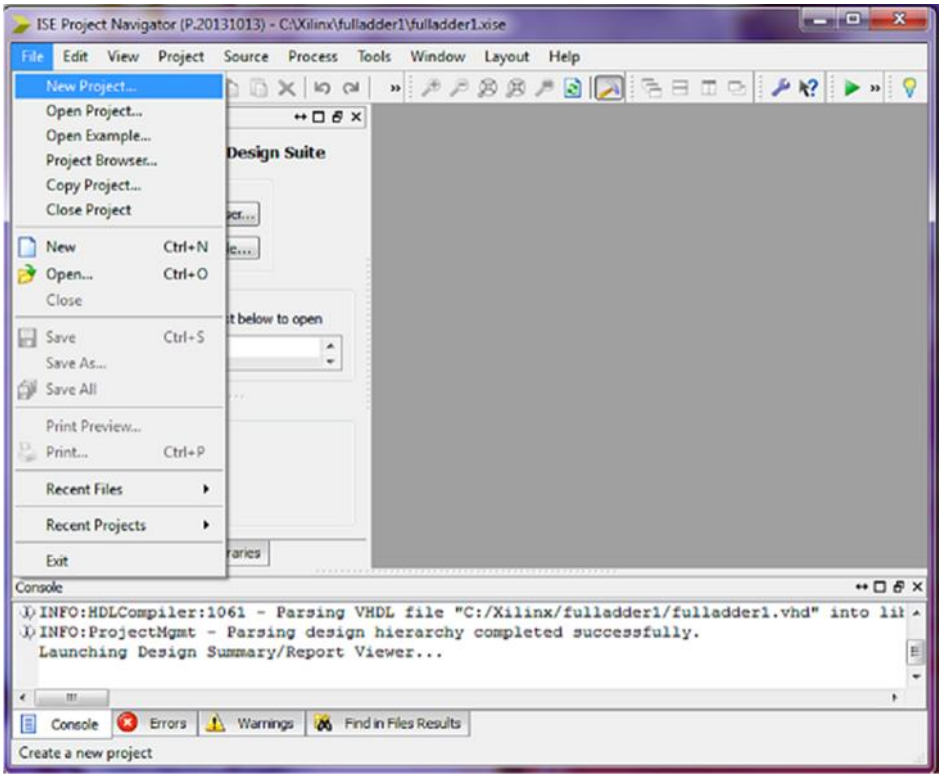
N.B. A PROCEDURE has the same location as those of a FUNCTION and can also be located in the main code.

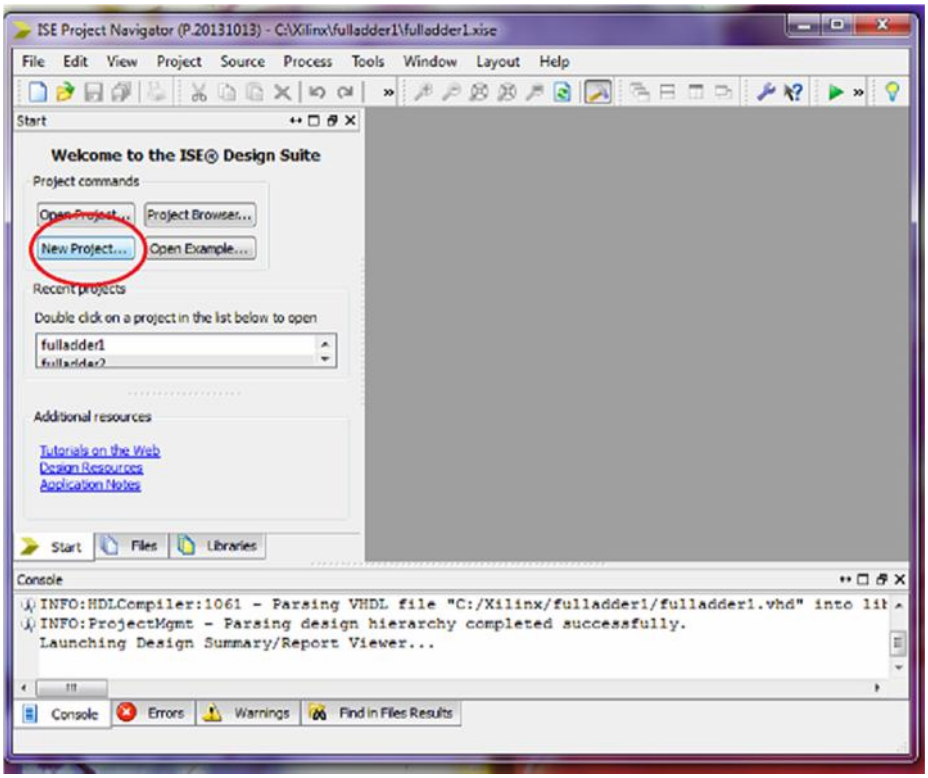
GETTING STARTED WITH VHDL

CREATION OF PROJECT

The steps to create a project in Xilinx ISE are as follows:

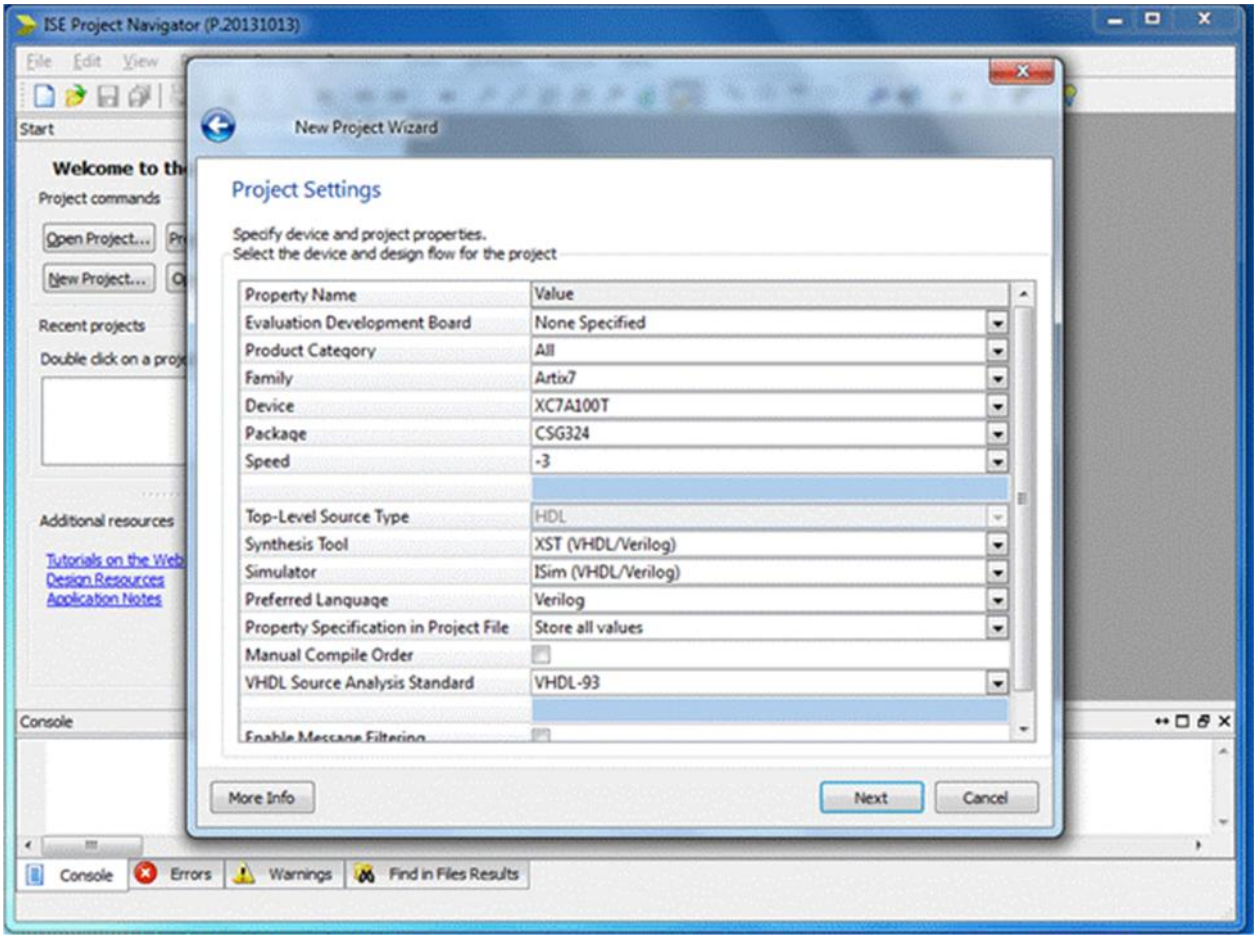
Create a new project by clicking File menu > New Project.





When the Create New Project window is displayed, choose your working directory and type your new project name in the project name field. Select “Top-level source type” as “HDL”. Add project description if you need.

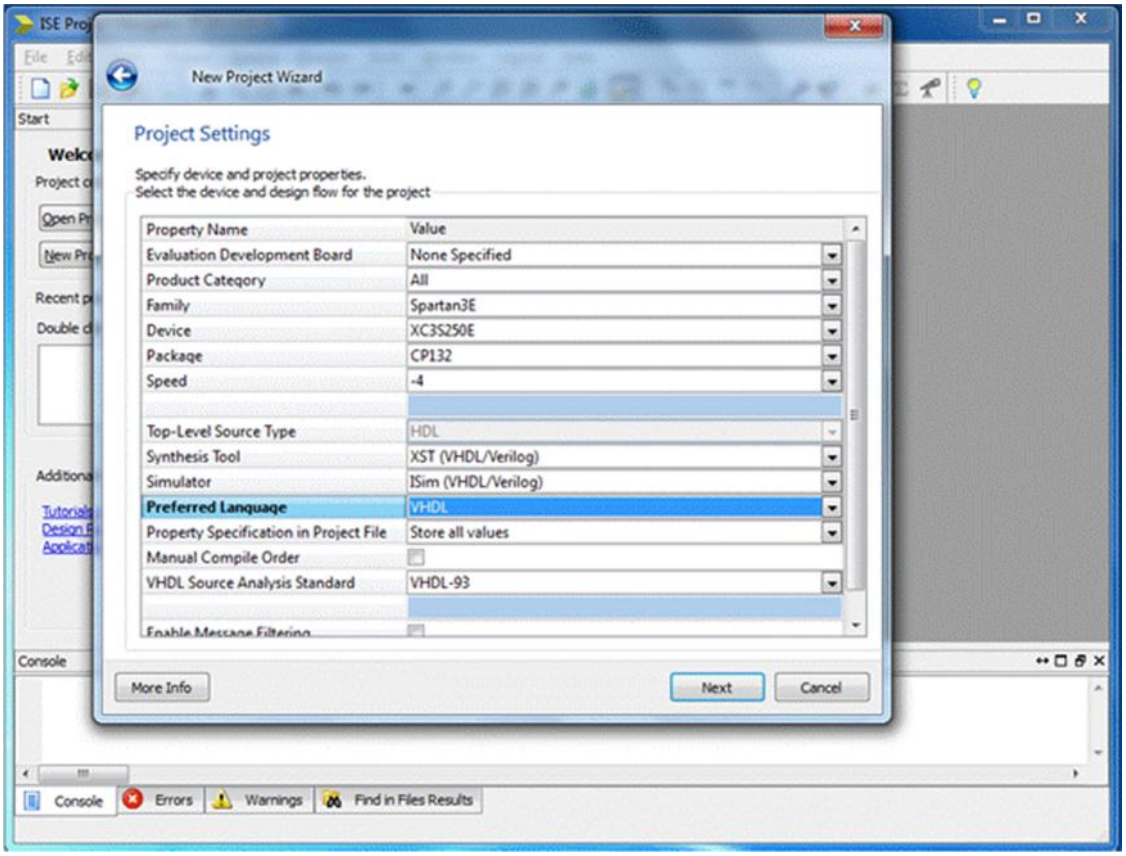
Click Next and you will go to the "Select the device and design flow for the project" interface.



Use the following settings to specify device and project properties.

1. Evaluation Development Board: - None Specified (default option)
2. Product Category: - All (default option)
3. Family: - Spartan3E
4. Device: - XC3S250E
5. Package: - CP132
6. Speed: - -4
7. Top-Level Source Type: - HDL (default option)
8. Synthesis Tool: - XST (VHDL/Verilog)
9. Simulator: - ISim (VHDL/Verilog)
10. Preferred Language: - VHDL

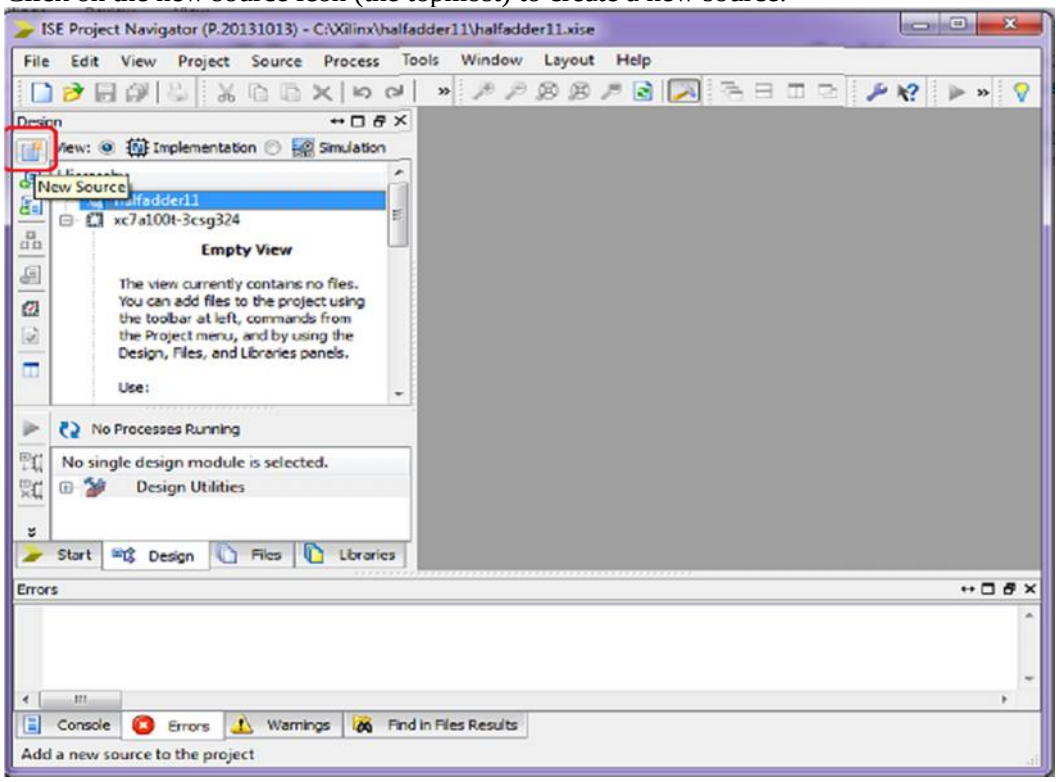
Note: - For the purpose of this course you can use the default settings for the remaining properties.



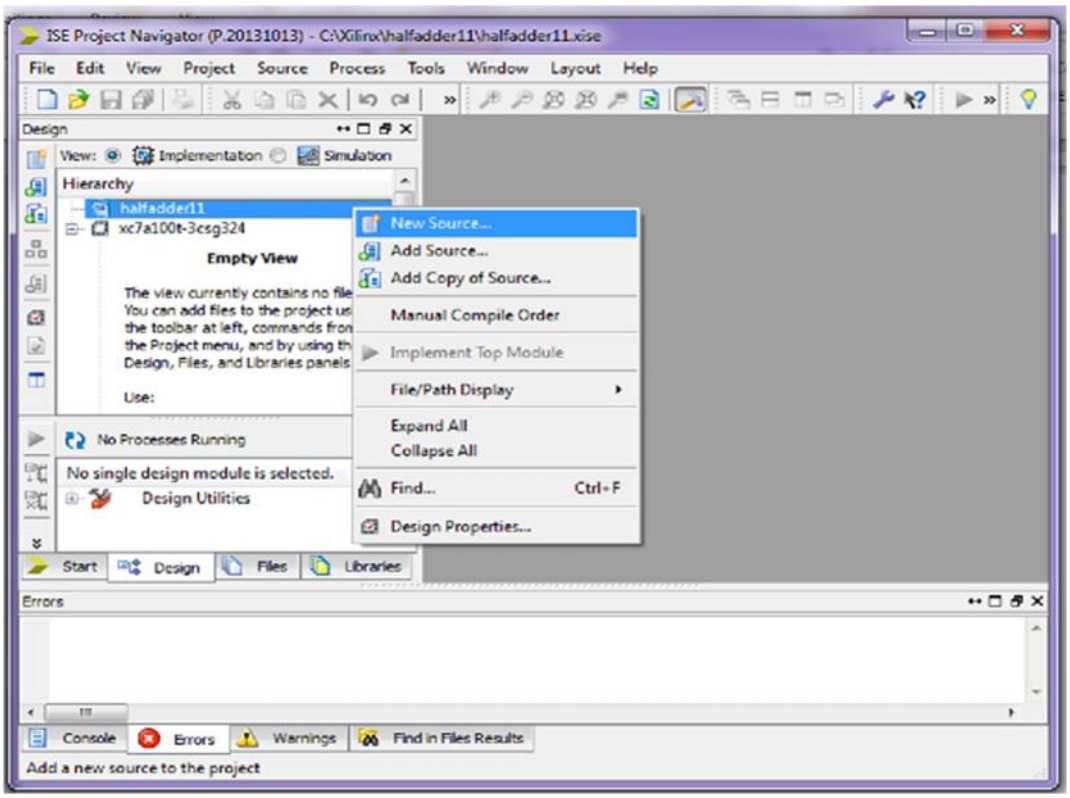
Click “Next” button and this will lead you to the “Project Summary” page of the “New Project Wizard”.

Click “Finish” to complete new project setup and go to the Project Navigator window.

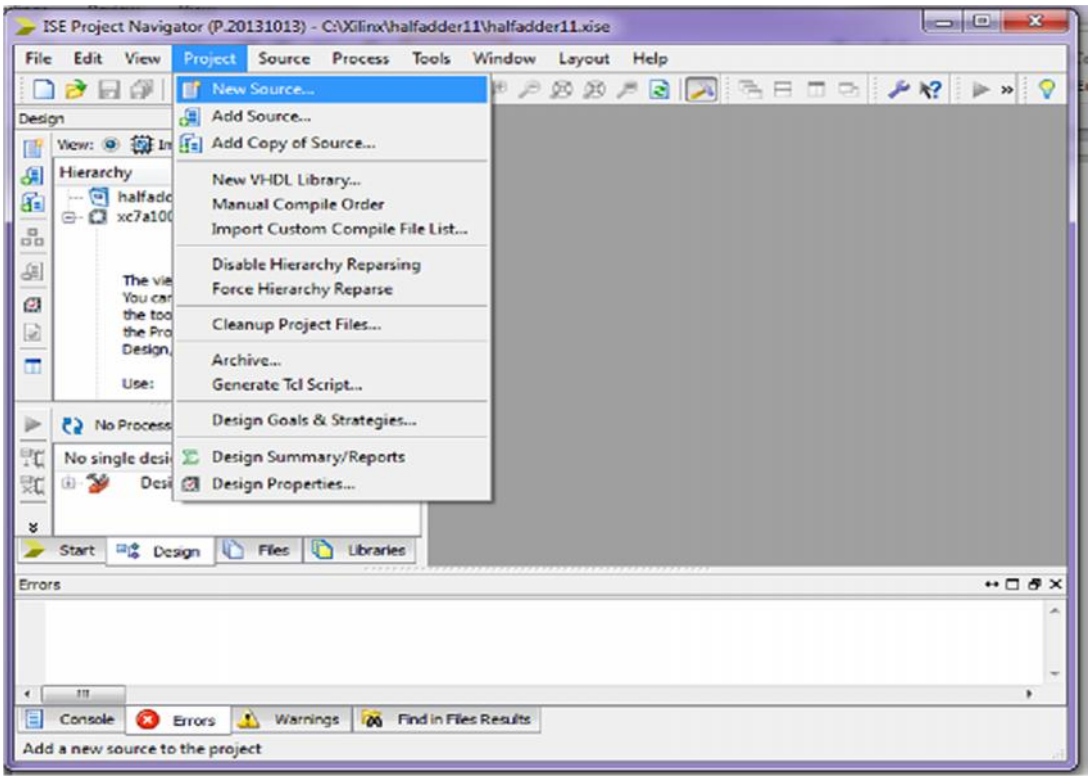
Create New Source You have three options in creating a new source file.
Option 1: Use the toolbar on the left-hand side of the project navigator window to create a new source. Click on the new source icon (the topmost) to create a new source.



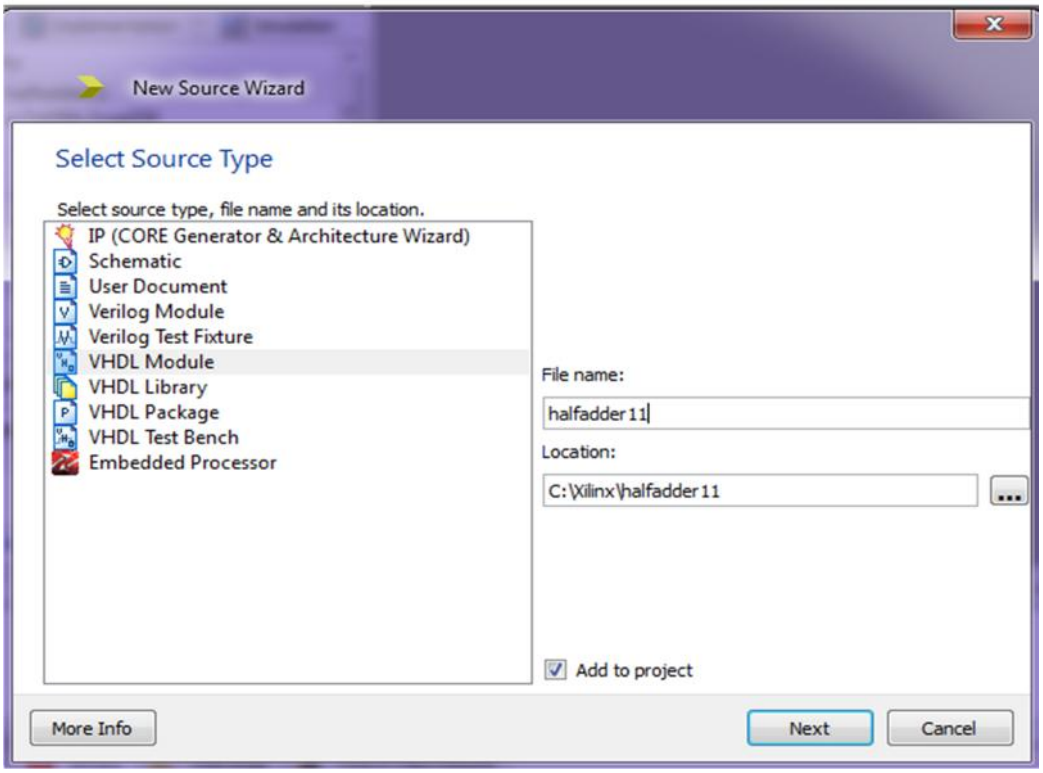
Option 2: Use the toolbar menu on the top. Project -> New Source to create a new source.



Option 3: Use the project in the “Hierarchy” section of the “Design” tab. Select the project (lab2 here), right-click and select “New Source” to create a new source.

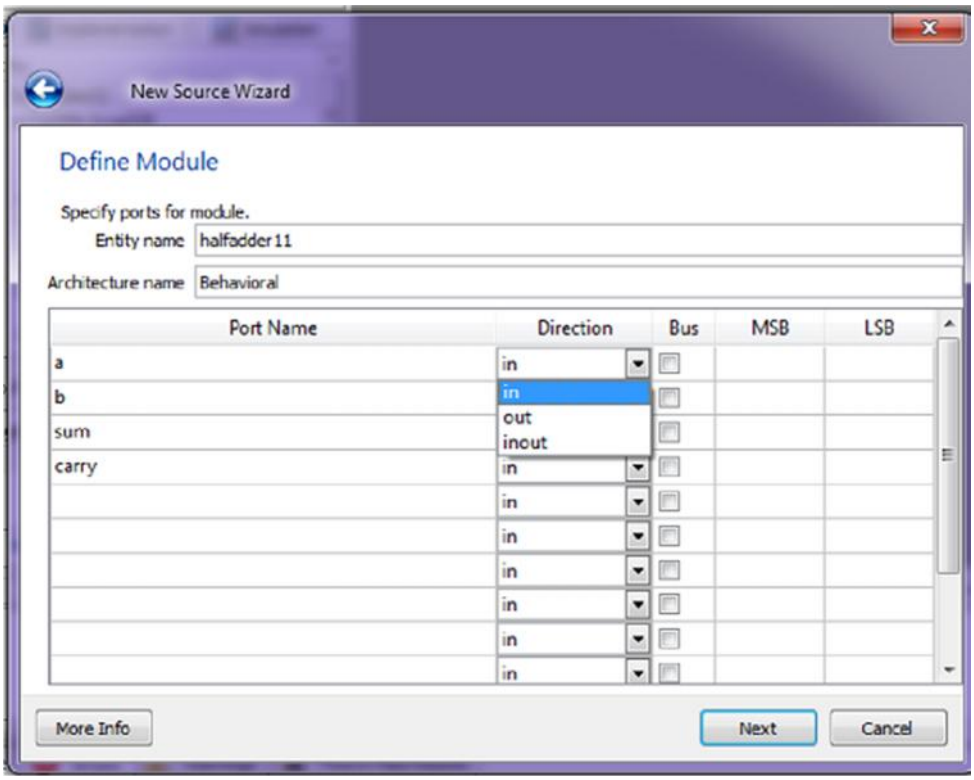


This brings up the “New Source Wizard” with options for various source types.

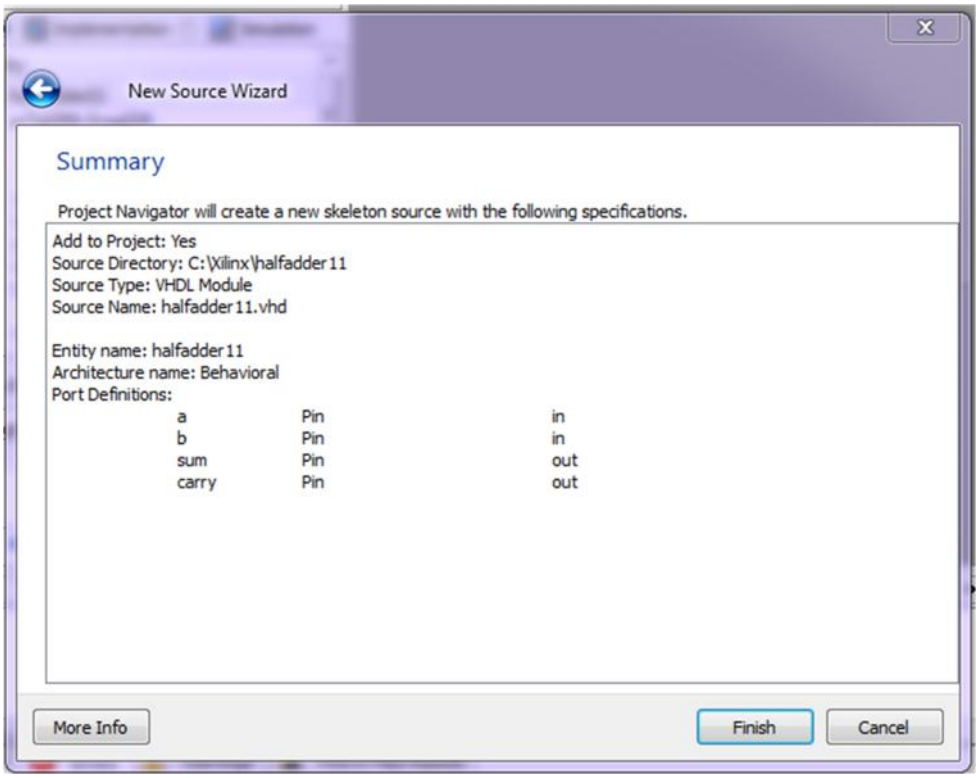


In the "New Source Wizard" interface choose VHDL Module from ‘Select Source Type’ and enter a name, “halfadder11”, for the new source file in the **File Name** field.

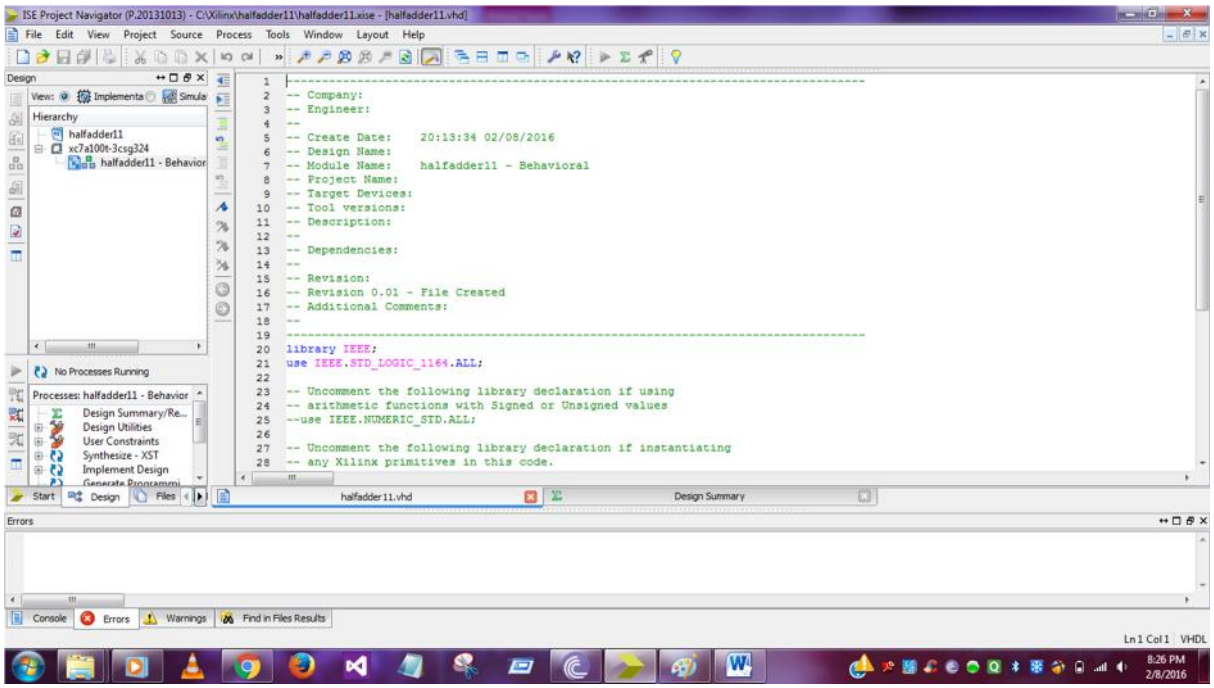
Make sure that “Add to project” option is selected. Click Next to continue. You will be asked to input the ports information, you can use the wizard or you just click Next. **Note:-**This information will be used by the tool, Xilinx ISE, to create a skeleton code to help you. You can edit the VHDL code generated if needed, for e.g., You can delete, add or modify port signals that might have specified.



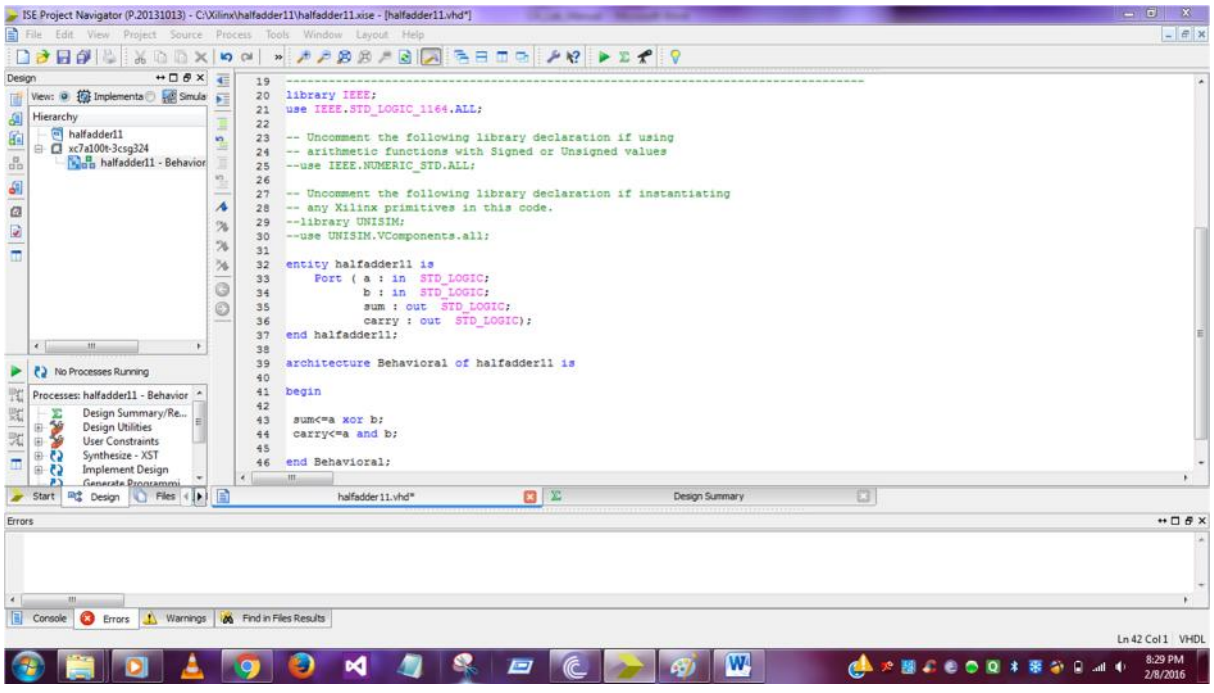
In this, we just click Next and then get the following summarized information for this source file.



Click the "Finish" button to continue. You will return to Xilinx ISE interface and see that the new source file, halfadder11.vhd, has been added to the project.



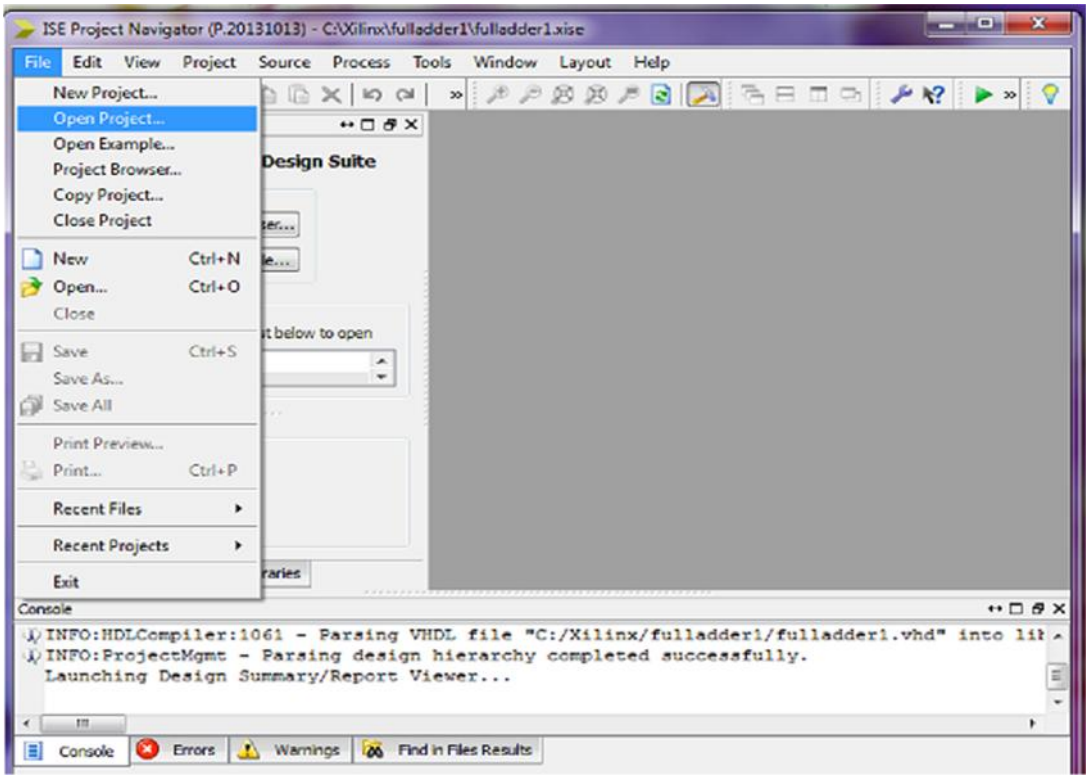
You can see in the right-hand window that a template of the VHDL source file, based on information you provided while creating the new source, is already generated for you by Xilinx ISE. You need to complete the source code based on your own design project.



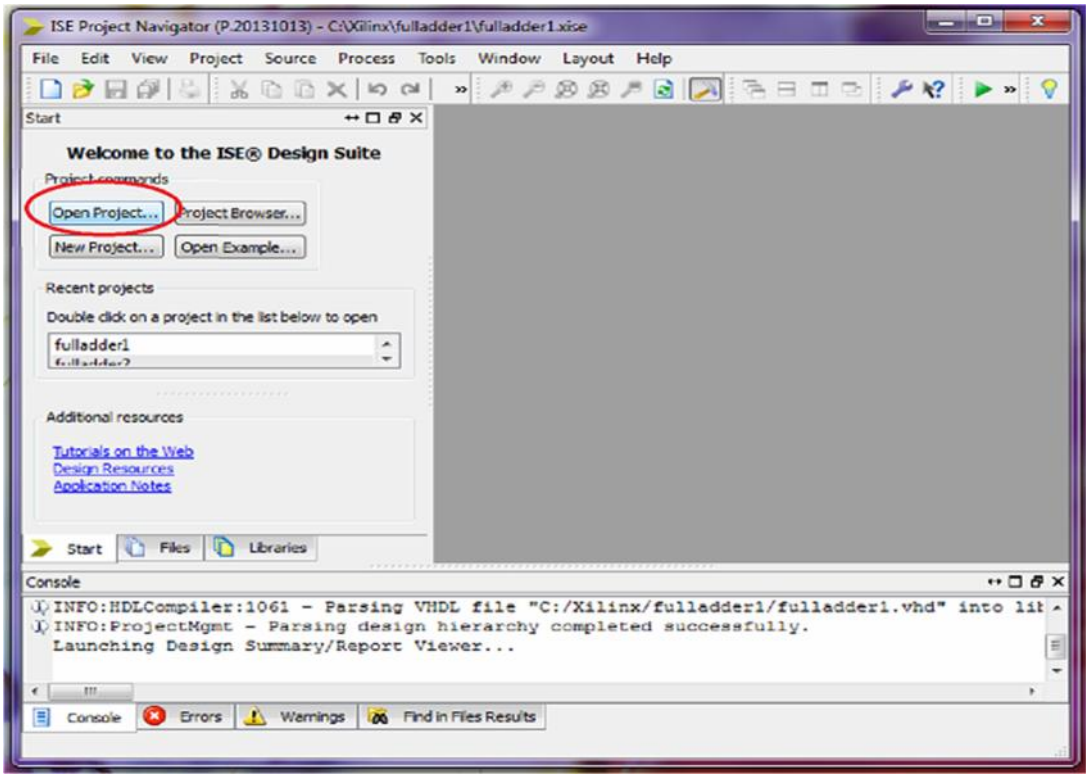
Note:-If you have an existing source file that you would like to add to the project you can either use the "Add Source" or "Add Copy of Source" option. "Add Source" can be used when the source file is either in the project directory or in a remote directory whereas "Add Copy of Source" is used when the source file is in a remote directory. "Add Copy of Source" copies the source file to the project directory.

Note: You should clearly understand what statements are automatically generated and what are needed to be coded by yourself.

To open an existing project the following steps should be followed:-

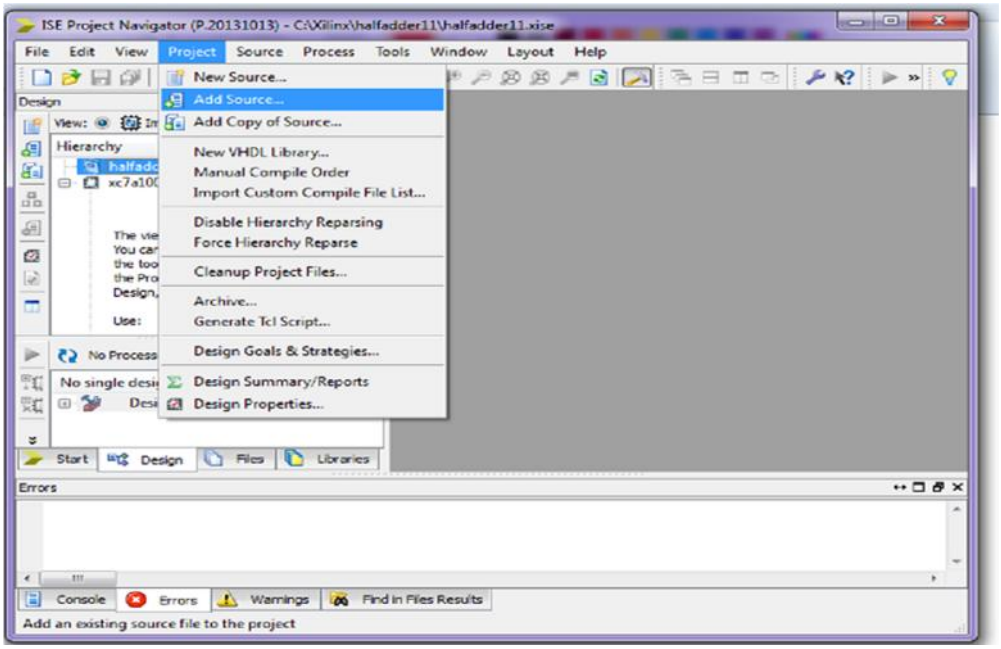


or

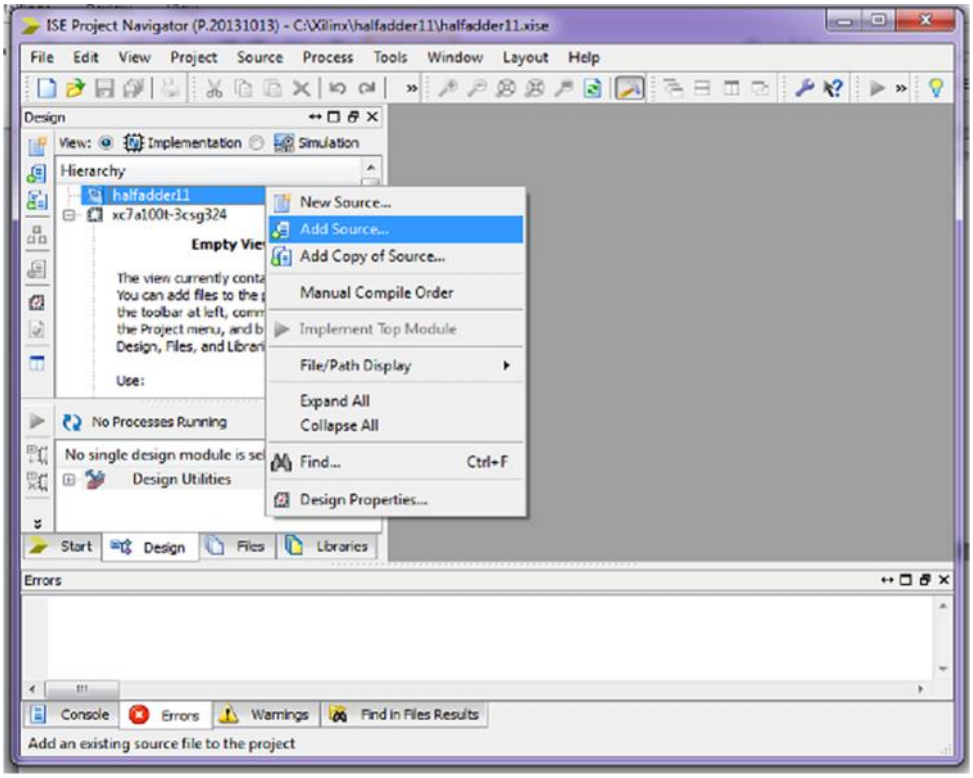


ADDING EXISTING SOURCE

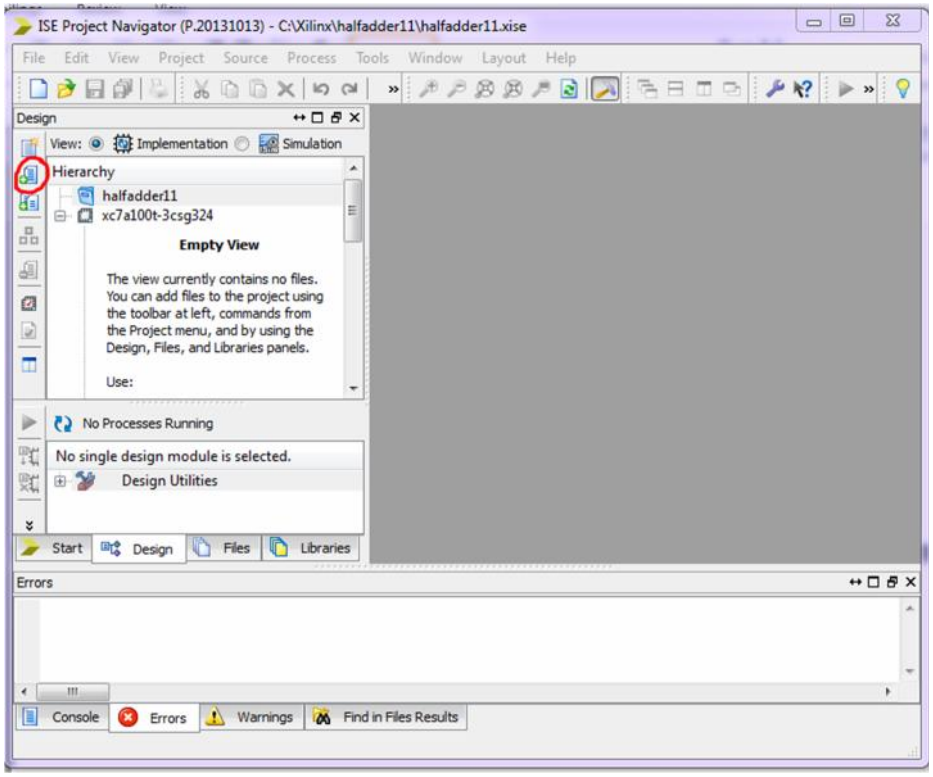
Option 1



Option 2

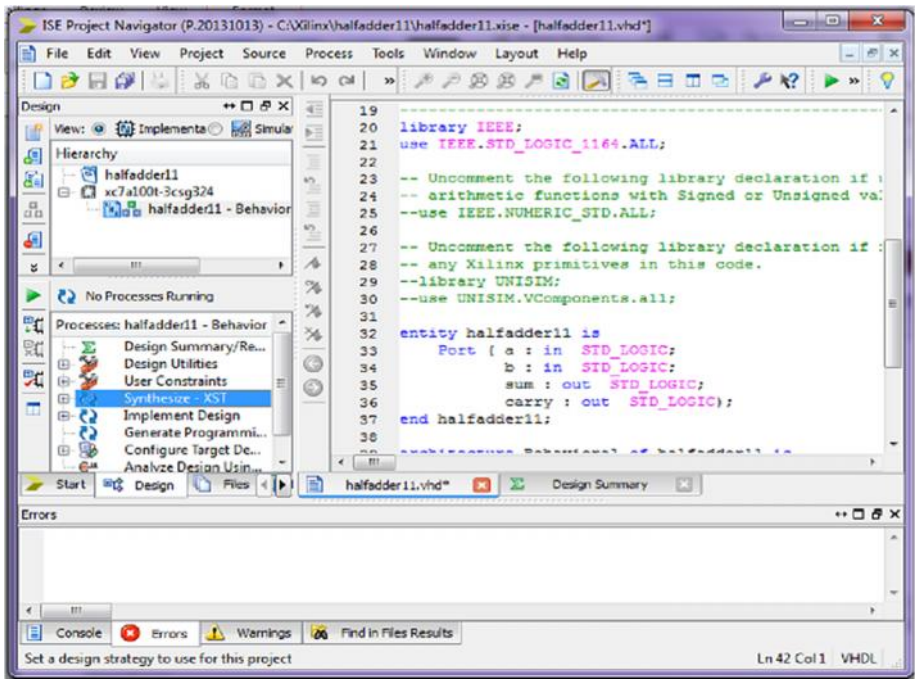


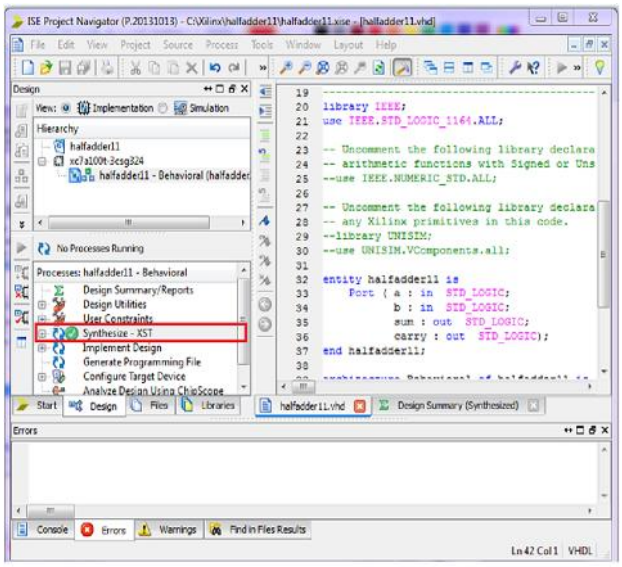
Option 3



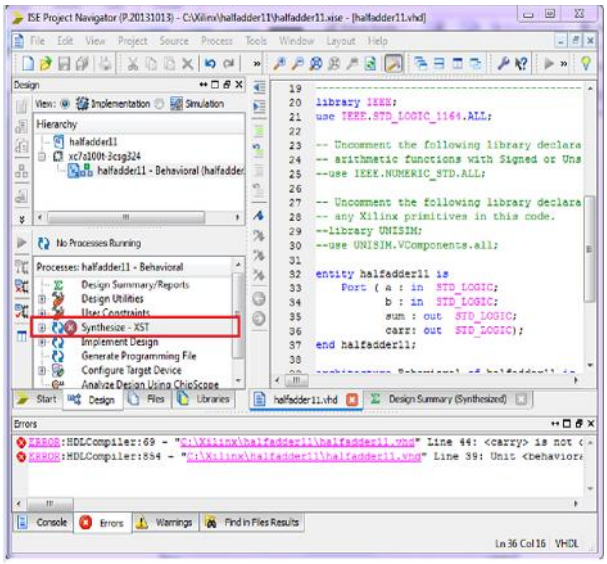
SYNTHESIZE DESIGN

1. Save the file by click Save All under file menu.
2. Select top-level entity. Here, halfadder11 is already on the top-level.
3. Double click Synthesize - XST to start the synthesizing process. When this process is finished, "**Process "Synthesize - XST" completed successfully**" is displayed in the console window





If synthesize report is OK



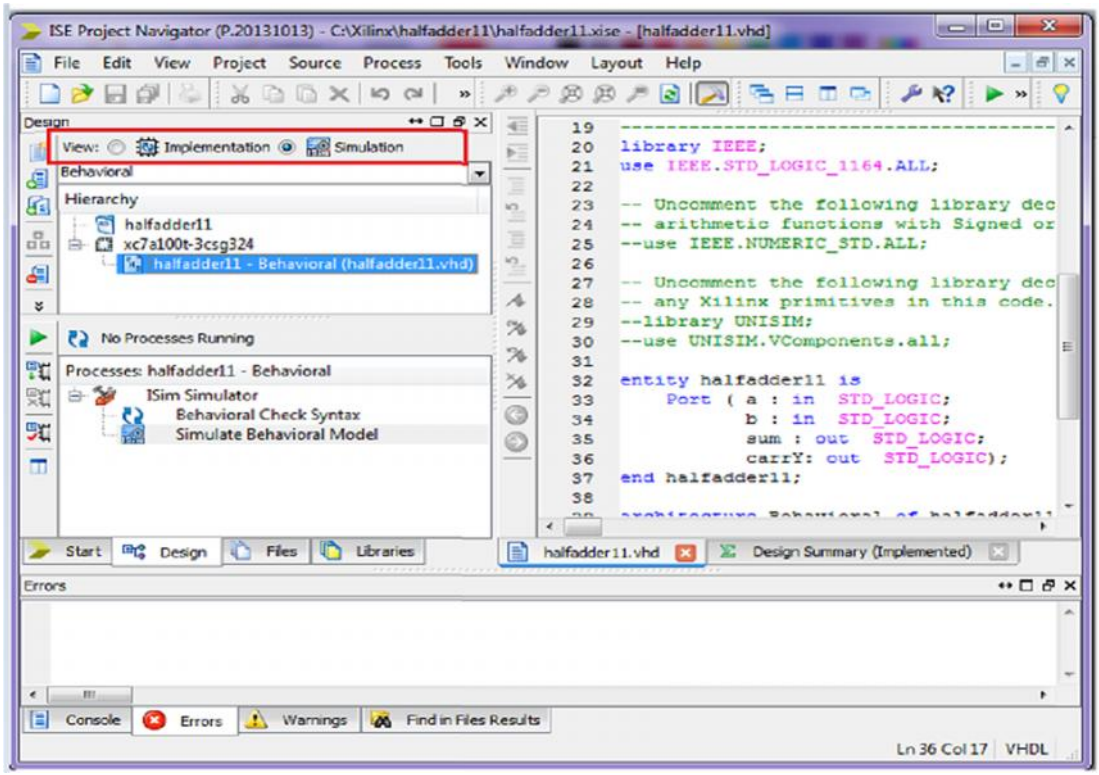
If synthesize fails

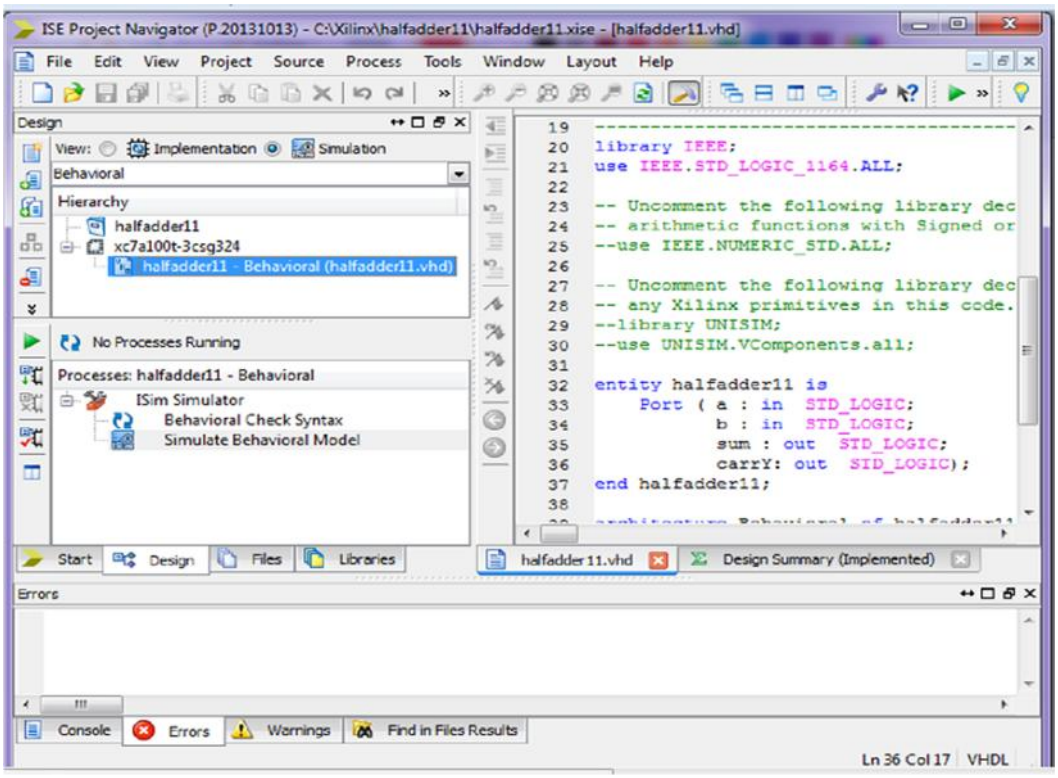
Note:

1. Synthesis is the process of converting the abstract circuit behavior, described by a VHDL (or any HDL) code, into a hardware implementation in terms of logic gates.
2. To observe the results of the synthesis, expand Synthesize - XST in "Process: halfadder11 (click on the + sign).

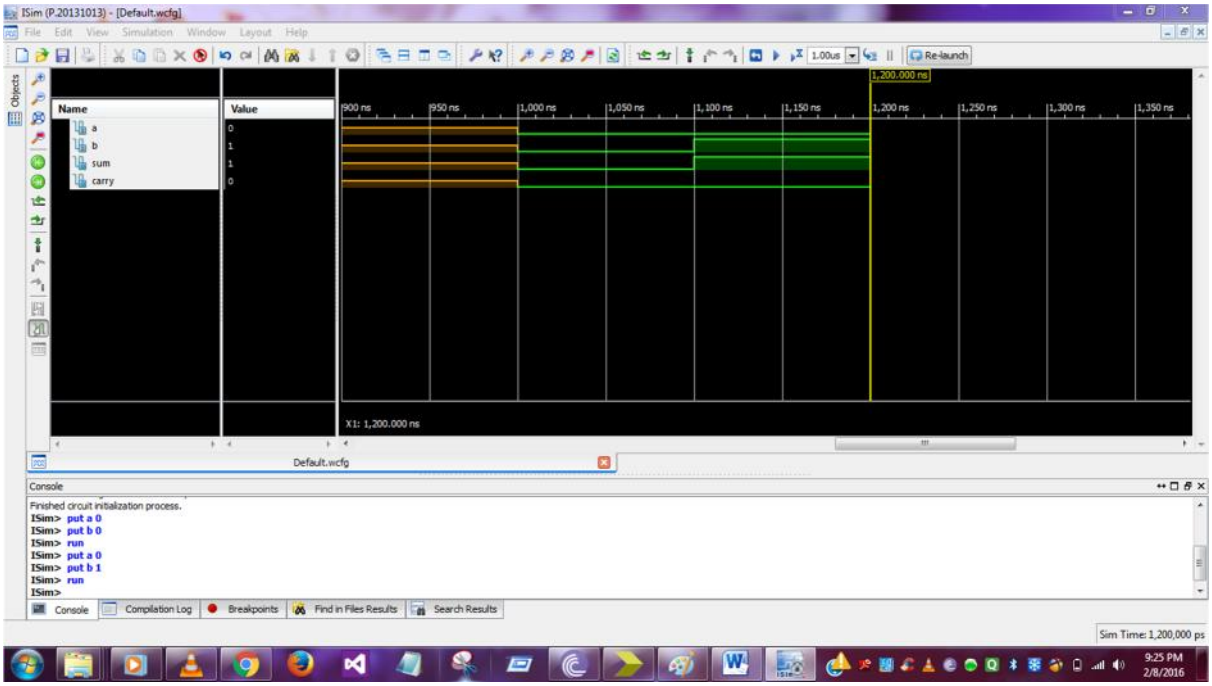
Source Code Simulation

<u>For Source Code simulation the following steps should be followed:</u>





ISIM Behavioral Mode



EXAMPLE CODES

VHDL introduction

Some Examples of Basic Logic gates

AND Gate

```
entity and1 is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          y : out  STD_LOGIC);
end and1;

architecture Behavioral of and1 is
begin
y<=a and b;
end Behavioral;
```

OR Gate

```
entity or1 is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          y : out  STD_LOGIC);
end or1;

architecture Behavioral of or1 is
begin
y<= a or b;
end Behavioral;
```

XOR Gate

```
entity xor1 is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          y : out  STD_LOGIC);
end xor1;

architecture Behavioral of xor1 is
begin
y<=a xor b;
end Behavioral;
```

Some Examples of Digital Logic Circuits

Half Adder using XOR and AND Gate as Component

```
entity hanew is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          sum : out  STD_LOGIC;
          carry : out  STD_LOGIC);
end hanew;

architecture Behavioral of hanew is

component xornew is
   Port ( x : in  STD_LOGIC;
          y : in  STD_LOGIC;
          z : out  STD_LOGIC);
end component;

component andnew is
   Port ( l : in  STD_LOGIC;
          m : in  STD_LOGIC;
          n : out  STD_LOGIC);
end component;

begin
X1: entity xornew port map(a,b,sum);
A1: entity andnew port map(a,b,carry);
end Behavioral;

entity xornew is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          c : out  STD_LOGIC);
end xornew;

architecture Behavioral of xornew is
begin
c<= a xor b;
end Behavioral;

entity andnew is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          c : out  STD_LOGIC);
end andnew;

architecture Behavioral of andnew is
begin
c<= a and b;
end Behavioral
```

Design of 2 to 4 Decoder

```
entity decoder2x4 is
   Port ( a : in  STD_LOGIC;
```

```
        b : in  STD_LOGIC;
        enable : in  STD_LOGIC;
        z : out  STD_LOGIC_VECTOR (3 downto 0));
end decoder2x4;

architecture Behavioral of decoder2x4 is

component inv is
   Port ( pin : in  STD_LOGIC;
        pout : out  STD_LOGIC);
end component;

component nand3 is
   Port ( d0 : in  STD_LOGIC;
        d1 : in  STD_LOGIC;
        d2 : in  STD_LOGIC;
        dz : out  STD_LOGIC);
end component;

signal abar,bbar:STD_LOGIC;

begin
v0: entity inv port map(a,abar);
v1: entity inv port map(b,bbar);
n0:entity nand3 port map(enable,abar,bbar,z(0));
n1:entity nand3 port map(enable,abar,b,z(1));
n2:entity nand3 port map(enable,a,bbar,z(2));
n3:entity nand3 port map(enable,a,b,z(3));
end Behavioral;
```

Design of Full Adder

```
entity fa is
   Port ( a : in  STD_LOGIC;
        b : in  STD_LOGIC;
        c : in  STD_LOGIC;
        sum : out  STD_LOGIC;
        carry : out  STD_LOGIC);
end fa;

architecture Behavioral of fa is
begin
sum<= a xor b xor c;
carry<= (a and b) or ( b and c) or ( c and a);
end Behavioral;
```

Design of J-K Flip-Flop

```
entity jkflipflop is
   Port ( J : in  STD_LOGIC;
          K : in  STD_LOGIC;
          CLK: in  STD_LOGIC;
          Q : inout  STD_LOGIC;
          QN : inout  STD_LOGIC);
end jkflipflop;

architecture Behavioral of jkflipflop is
begin
          process(CLK,J,K)
          begin
                    if (CLK='1' and CLK'event) then
                    if(J='0' and K='0') then
                              Q <=Q;
                              QN <=QN;
                    elsif(J='0' and K='1') then
                              Q <= '0';
                              QN <= '1';
                    elsif(J='1' and K='0') then
                              Q <= '1';
                              QN <= '0';
                    elsif(J='1' and K='1') then
                              Q <= NOT Q;
                              QN <= NOT QN;
                    end if;
                    end if;
          end process;
end Behavioral;
```

Implementation of Multiplier using Booth's Algorithm

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
library IEEE;
entity multiplier2 is
   Port ( mplr : in  STD_LOGIC_VECTOR (1 downto 0);
          mpcd : in  STD_LOGIC_VECTOR (1 downto 0);
          start : in  STD_LOGIC;
          result : out  STD_LOGIC_VECTOR (3 downto 0));
end multiplier2;

architecture Behavioral of multiplier2 is

begin
process(start,mpcd,mplr)
variable br,nbr:std_logic_vector(1 downto 0);
variable acqr:std_logic_vector(3 downto 0);
variable qn1:std_logic;
begin
          if( start='1') then
                    acqr(3 downto 2):=(others=>'0');
                    acqr(1 downto 0):=mplr;
                    br:=mpcd;
                    nbr:=(not mpcd)+'1';
                    qn1:='0';

          loop1:for i in 1 downto 0 loop
                    if(acqr(0)='0' and qn1='0') then
                              qn1:=acqr(0);
                              acqr(2 downto 0 ):=acqr(3 downto 1);
                    end if;
                    if(acqr(0)='0' and qn1='1') then
                                        acqr(3 downto 2 ):=acqr(3 downto 2)+br;
                                        qn1:=acqr(0);
                                        acqr(2 downto 0 ):=acqr(3 downto 1);
                    end if;
                    if(acqr(0)='1' and qn1='0') then
                                                  acqr(3 downto 2 ):=acqr(3 downto 2)+nbr;
                                                  qn1:=acqr(0);
                                                  acqr(2 downto 0 ):=acqr(3 downto 1);
                    end if;
                    if(acqr(0)='1' and qn1='1') then
                                                                qn1:=acqr(0);
                                                                acqr(2 downto 0 ):=acqr(3 downto 1);
                    end if;

          end loop loop1;
```

```
        result<=acqr;
        end if;
end process;

end Behavioral;
```

Implementation of Divisor

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

entity divisor is
   Port ( a,b: in integer range 0 to 15;
          y : out  STD_LOGIC_VECTOR (3 downto 0);
          rest : out integer range 0 to 15;
          err : out  STD_LOGIC);
end divisor;

architecture Behavioral of divisor is

begin
process(a,b)

variable temp1  : integer range 0 to 15;
variable temp2  : integer range 0 to 15;

begin
temp1   :=a;
temp2   :=b;
if(b=0)then err <='1';
else err <='0';
end if;
if(temp1>=temp2*8) then
y(3)<='1';
temp1:=temp1-temp2*8;
else y(3)<='0';
end if;

if(temp1>=temp2*4) then
y(2)<='1';
temp1:=temp1-temp2*4;
else y(2)<='0';
end if;

if(temp1>=temp2*2) then
y(1)<='1';
temp1:=temp1-temp2*2;
else y(1)<='0';
end if;
```

```
if(temp1>=temp2) then
y(0)<='1';
temp1:=temp1-temp2;
else y(0)<='0';
end if;
rest<=temp1;
end process;

end Behavioral;
```

Design of different Computer System

Implementation of Shift Register

```
entity shiftregister is

   Port ( CLK : in  STD_LOGIC;
          SI : in  STD_LOGIC;
          SO : out  STD_LOGIC);
end shiftregister;

architecture Behavioral of shiftregister is

signal tmp: std_logic_vector(7 downto 0);
begin
process (CLK)
begin
if (CLK'event and CLK='1') then
for i in 0 to 6 loop
tmp(i+1) <= tmp(i);
end loop;
tmp(0) <= SI;
end if;
end process;
SO <= tmp(7);

end Behavioral;
```

Design of RAM

```
entity rams_01 is
 port (clk : in std_logic;
 we : in std_logic;
 en : in std_logic;
 addr : in std_logic_vector(5 downto 0);
 di : in std_logic_vector(15 downto 0);
 do : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is

 type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);

 signal RAM: ram_type;

begin

 process (clk)

 begin

 if clk'event and clk = '1' then
 if en = '1' then
 if we = '1' then
 RAM(conv_integer(addr)) <= di;
 end if;
 do <= RAM(conv_integer(addr)) ;
 end if;
 end if;

 end process;

end syn;
```