BIRZEIT UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Advanced Digital Systems
ENCS3310

Course Project Report

(Generic Multiplier Using VHDL)

Prepared by: Alaa Zuhd - 1180865

Instructor: Dr. Abdallatif Abuissa

Section: 2

BIRZEIT

May 8, 2021

1 ABSTRACT

The aim of this project is to design a generic multiplier for two numbers (K-bit*J-Bit) structurally using two different algorithms with VHDL coding, the first algorithm is the parallel binary multiplication, and the second is the partial product addition and shifting, and finally we need to have a complete and automatic verification for the code using an analyzer in the test bench.

Contents

2 BRIEF INTRODUCTION

VHDL is a hardware description language that can describe the structure and the behavior of any digital circuit, and provide an important aspect in the design process, which is the testing process, so here the test can be made automatically to detect the errors and any possible glitch that can exist because of using different gates with considerable amount of delay. One of the important combinational circuits is the the multiplier circuit, so in this project the implementation of this logic will be done using two different algorithms, as will be explained later.

3 BRIEF THEORETICAL OVERVIEW

The multiplication process of two binary numbers is quiet the same as the one for the decimal numbers, so for each bit in the multiplier, we multiply it with the multiplicand which is done using an AND gate, so after multiplying each bit, it will results in partial product that's finally added together to get the final result with maximum number of bits equal to the sum of the bits values of the two numbers. The following demonstrate the two most common approaches in multiplying two unsigned binary numbers.

3.1 Parallel Binary Multiplication Method

In general, the binary multiplication of a number with another number of 1-bit, can be achieved through the and of the first number with the bit in the second number, so in-order to generalize this method to multiply a number of K-bits by another number of J-bits the following is done, X is the first number of J-bits, and Y is the second number of K-Bits, and i is initialized to 0:

1. Multiply X by Y[0], by ANDING the bits of X with Y[0], then the first bit in the result will be equal to X[0] AND Y[0], and the result of this stage will be considered as the first input with a logical shift to the right by 1.
2. If k is 1, then finish with result equal to the first input generated in step 1, else continue.
3. Multiply X by the next bit of Y (Y[i]), by ANDING the bits of X with Y[i], and this will be considered the second Input.

4. Then add the first input and the second input (using J-bit adder), and the first bit in the result will be the next bit in the final result.
5. If the number of iteration is equal to K-1, then the multiplication process is done, and the rest bits in the result will be obtained from the result of the last addition res[j+k-1 down to k-1] = cout&res-add[j-1 down to 0]. else res[i]=res-add[0], and updated the first input of the next addition to be first_input= cout&res-add[j-1 down to 1] and repeat from step 3.

3.2 Partial Product Addition Shifting

This method of multiplication, depends on using k-bit adder, and Four registers (A and B of k-bits, Q of j-bits and C of 1-bit), in addition to a control logic circuit to choose between only shift and Add then shift (2X1MUX). The following illustrate the way this algorithm work:

1. First step is the initialization, set Q=X, B=Y, A=0, C=0 and i=0 (counter).
2. Next, get the current bit of Q (Q[i]), and add it to the control Unit.
3. If the selection to the control unit is 0 then a shift only is needed, meaning the sequence of C&A&Q will be shifted to the right. else we need to do addition for A=A+B, and the carry will be in C register, then the shift right for the sequence C&A&Q will take place.
4. Increment i and repeat steps 2 and 3 until i=j-1.
5. Finally the result will be stored in A and Q respectively (res = A&Q).

4 Design Philosophy

The design process of the structural implementation of the two algorithms was done through many stages in-order to simplify and and to organize the design process, the first step was to prepare and design some simple and basic gates that will be used by any of the two algorithms, the second stage was the implementation of each algorithm, and the final stage is the testing stage. The design stages are explained as the following.

4.1 Design of the Basic Gates

Simple Gates: (2-bit AND, 2-bit OR and 2-bit XOR)

A model was implemented for each one of these gates in-order to use them to build the model of the generic multiplier structurally.

1-Bit Adder

The 1-bit Adder was implemented structurally from AND, OR and XOR gates, and this model is an important model in the design of the generic multiplier, which can help in implementing a generic n-bit adder that's used directly in the implementation of both algorithms.

N-Bit Adder

As mentioned above, the generic model for the n-bit adder was implemented through the usage of n models of the 1-bit adder structurally, and this model was used a lot in the implementation of the two algorithms since the multiplication process depends strongly on the addition process.

Special Right Sift Model

This model was used many times in the implementation of the two algorithms, it works like a logical right shift with an additional feature that the bit to be added to the left of the number to be shifted is equal to the value specified in the new_most_left_bit input signal. The following instruction explain the work of the model : output <= new_left_most_bit & input(k-1 downto 1).

2X1MUX

This model is just a simple model for the 2X1MUX. that used in the second algorithm as it will be explained later.

4.2 Design of the Two Algorithms

This process is the main aim of the project, so it include an explanation of the structure that was implemented for both algorithms.

Algorithm NO.1: Structural_Generic_Multiplier_Algorithm_NO_1

The implementation of the generic multiplier using the first algorithm was done based on the algorithm described in the theoretical part of this algorithm. the following will explain the implementation from the VHDL view and the design features.

1. First, this design take into consideration the number of bits of each number, such that both input numbers can't have a zero or negative number of bits, since the generic parameters were defined to be positive, and both numbers don't need to be the same value.
2. As mentioned previously, This design succeed in dealing with any number for J and especially K bits, such that it was illustrated that when K is 1, the circuit used to build the model have a different sequence than with k>1. so when K is 1 all what we need to do is to AND the bits of X with the first and only bit in Y, and do a logical right shift, then store the result in res signal, and this thing was possible by implementing this operation outside the for generate that run from 1 to K-1, so whenever K is one then it will never enter the loop.
3. Finally, if K is more than one then the first step, which is the ANDING between X bits' and Y(0), will be ready from the previous step, and then by entering the for generate we will be able to add the first input (the one generated from the previous step or from the previous iteration) with the second input which represents the AND between X bits and Y(i), so here for each iteration (which they are K-1 iteration) an n-bit adder (n=J) will be used

   ```
   AdderI: entity work.N_Bit_Adder(structural_N_Bit_Adder) generic map (j)
       port map (temp_in_1(i-1), temp_in_2(i-1), '0', temp_out(i-1),
       Cout(i-1));
   ```

   After that, the first bit in temp_out(i-1) will be added to res(i), and the next input (input_in_1) will take the temp_out value with a right shift for it, and concatenating it with the carry out (cout(i-1)), by using the shifter model as the following.

   ```
   ShifterI1: entity work.special_right_shift(special_right_shift) generic
       map (j) port map (Cout(i-1), temp_out(i-1), temp_in_1(i));
   ```

   And the last step will continue for K-1 times.

Algorithm NO.2: Structural_Generic_Multiplier_Algorithm_NO_2

The structural implementation of the generic multiplier using the second algorithm was done based on the algorithm described in the theoretical part of this algorithm. the following will explain the implementation from the VHDL view and the design features.

1. First, the initialization of the registers that described in the theoretical overview was done such that Q(0)=X, and the other register were set to 0.
2. Since the multiplication process need J iterations, a generate for was used to loop through the whole stages of this process.
3. for each iteration, two main steps were be considered, first the action to be applied is considered as a shift only so a right shift for the Sequence C&A&Q was made and the result stored in C_shift(i)A_shift(i)&Q_shift(i), on the other side the action is considered to be an addition for A and Y following with a shift action for C_Add(i)&A_Add(i)&Q(i), (where C_Add(i) and A_Add(i) here, are the ones generated from the addition) and store them in C_Addition_Shift(i)&A_Addition_Shift(i)& Q_Addition_Shift(i) registers. and finally the second main step is to take one of these two actions which was done through a 2X1MUX, with Q(i)(0) as it's selection (0 means shift only, and 1 means add then shift).
4. After finishing the generate for iterations, the final output will be res <= A(J) & Q(J).

4.3 Design of the Testing System

For the testing methodology, an automatic test bench was used, consisting of a test generator, a unit of the model to be tested, and a test analyzer to automatically test the system.

Test Generator

This model, is used to generate all the possible combinations of the two test inputs, which was done by using two loops, the outer loop loops through 0 to pow(2,J)-1 and the inner loop loops through 0 to pow(2,K)-1, and converting these two integer number into an std_logic_vector of size J and K bits respectively, and finally calculating the expected output by using the (*) operator.

An Instance of the Model to be Tested

An instance of the model to be tested (algorithm 1 or 2) was placed in-order to calculated the expected value for the two inputs generated from the test generator.

Test Analyzer

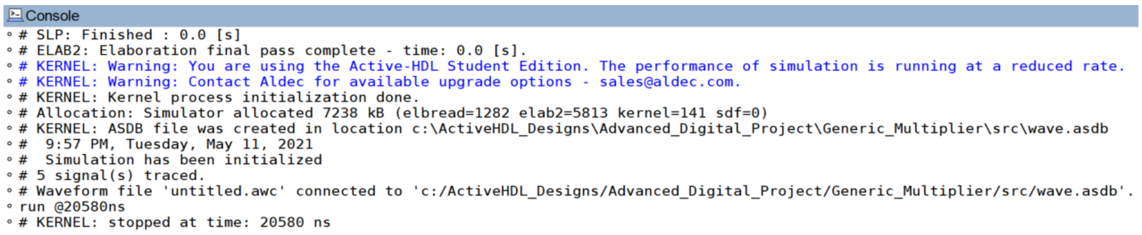
This model will check if the actual value is the same as the expected one, by using the assert statement, and in case of an inconsistency then a warning message will be printed on the console by using the report statement.

5 Simulation Results

This section is for tracing and testing the behavior of the two built models, so the following will demonstrate some wave-forms and the console output for different scenarios for J=7bits and K=4bits, also it will consider the simulation for two cases, once without any intentional error and once with an intentional error.

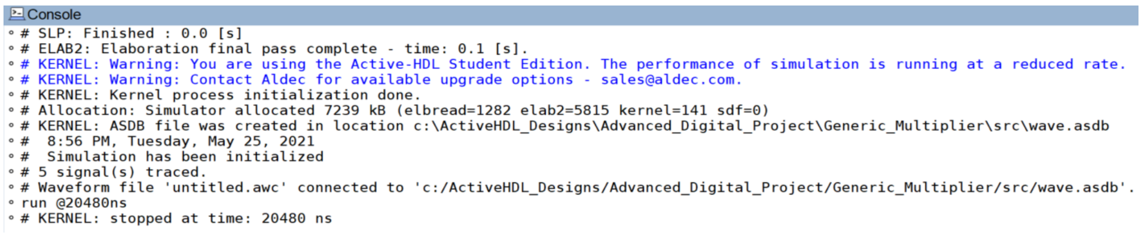
5.1 Simulation without an Introduced Error

The following will illustrate the console output when testing the two models on the whole possible combinations of the inputs with a zoom in for the waveform output of each model.



```
Console
# SLP: Finished : 0.0 [s]
# ELAB2: Elaboration final pass complete - time: 0.0 [s].
# KERNEL: Warning: You are using the Active-HDL Student Edition. The performance of simulation is running at a reduced rate.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 7238 kB (elbread=1282 elab2=5813 kernel=141 sdf=0)
# KERNEL: ASDB file was created in location c:\ActiveHDL_Designs\Advanced_Digital_Project\Generic_Multiplier\src\wave.asdb
#  9:57 PM, Tuesday, May 11, 2021
#  Simulation has been initialized
# 5 signal(s) traced.
# Waveform file 'untitled.awc' connected to 'c:/ActiveHDL_Designs/Advanced_Digital_Project/Generic_Multiplier/src/wave.asdb'.
run @20580ns
# KERNEL: stopped at time: 20580 ns
```

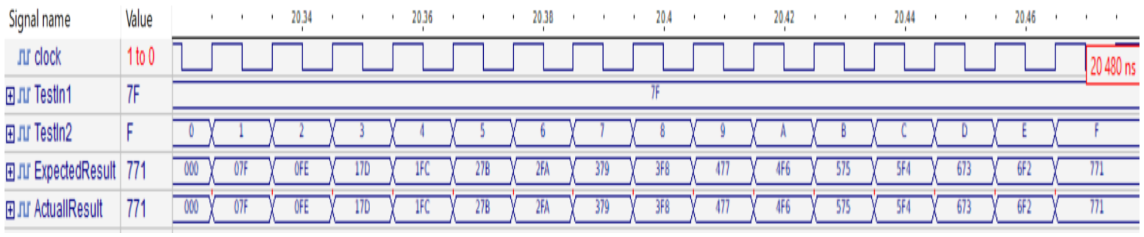
Figure 1: Console output for Algorithm NO.1 model without having any error



```
Console
# SLP: Finished : 0.0 [s]
# ELAB2: Elaboration final pass complete - time: 0.1 [s].
# KERNEL: Warning: You are using the Active-HDL Student Edition. The performance of simulation is running at a reduced rate.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 7239 kB (elbread=1282 elab2=5815 kernel=141 sdf=0)
# KERNEL: ASDB file was created in location c:\ActiveHDL_Designs\Advanced_Digital_Project\Generic_Multiplier\src\wave.asdb
#  8:56 PM, Tuesday, May 25, 2021
#  Simulation has been initialized
# 5 signal(s) traced.
# Waveform file 'untitled.awc' connected to 'c:/ActiveHDL_Designs/Advanced_Digital_Project/Generic_Multiplier/src/wave.asdb'.
run @20480ns
# KERNEL: stopped at time: 20480 ns
```

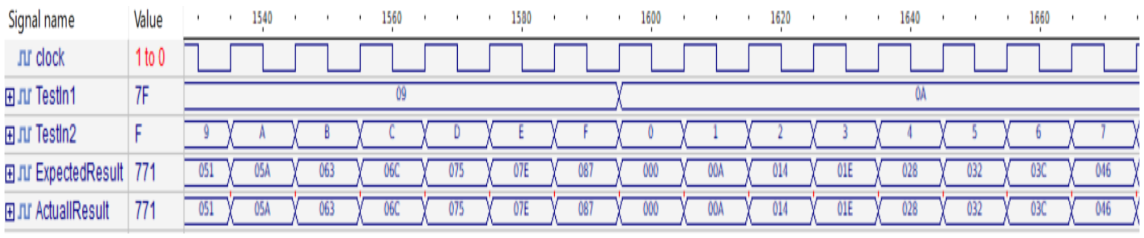
Figure 2: Console output for Algorithm NO.2 model without having any error

Discussion: From the above two images, we can notice that each simulation was run for "20480 ns", meaning it covers all the possible combinations of X and Y (pow(2,7) * pow(2,4) *10ns 'clock period'), also we noticed there isn't any error in any of the two models, since the statement "Multiplier output is incorrect" is not exist, which provide a proof that the implementation for the two models is true.



Signal name
Value
clock 1 to 0
TestIn1 7F
TestIn2 F
ExpectedResult 771
ActualResult 771
20.34
20.36
20.38
20.4
20.42
20.44
20.46
20 480 ns
7F
0 1 2 3 4 5 6 7 8 9 A B C D E F
000 07F 0FE 17D 1FC 27B 2FA 379 3F8 477 4F6 575 5F4 673 6F2 771
000 07F 0FE 17D 1FC 27B 2FA 379 3F8 477 4F6 575 5F4 673 6F2 771

Figure 3: A zoom in for the simulation of Algorithm NO.1 model



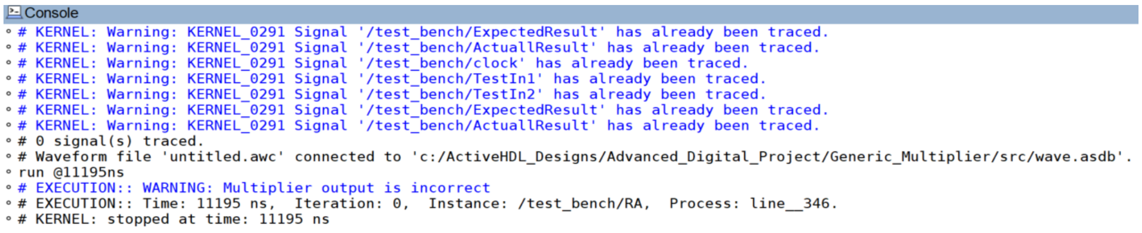
Signal name
Value
clock 1 to 0
TestIn1 7F
TestIn2 F
ExpectedResult 771
ActualResult 771
1540
1560
1580
1600
1620
1640
1660
09
0A
9 A B C D E F 0 1 2 3 4 5 6 7
051 05A 063 06C 075 07E 087 000 00A 014 01E 028 032 03C 046
051 05A 063 06C 075 07E 087 000 00A 014 01E 028 032 03C 046

Figure 4: A zoom in for the simulation of Algorithm NO.2 model

Discussion: From the above two images we noticed that the simulation waveform of the first model includes multiplying X "TestIn1"with value 7F, with some other values of Y"TestIn2", while the simulation waveform of the second model illustrate multiplying X with values 9 and A by some values of Y, and from that we noticed that in both simulations the expected result is equal to the actual result, meaning there is no errors, which is as expected.

5.2 Simulation with an Introduced Error

The following will illustrate the console output when testing the two models with an introduced error by applying the simulation on some combinations of the two inputs, also the corresponding waveform results for each model will be displayed.



```
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/ExpectedResult' has already been traced.
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/ActuallResult' has already been traced.
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/clock' has already been traced.
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/TestIn1' has already been traced.
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/TestIn2' has already been traced.
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/ExpectedResult' has already been traced.
# KERNEL: Warning: KERNEL_0291 Signal '/test_bench/ActuallResult' has already been traced.
# 0 signal(s) traced.
# Waveform file 'untitled.awc' connected to 'c:/ActiveHDL_Designs/Advanced_Digital_Project/Generic_Multiplier/src/wave.asdb'.
run @11195ns
# EXECUTION:: WARNING: Multiplier output is incorrect
# EXECUTION:: Time: 11195 ns,  Iteration: 0,  Instance: /test_bench/RA,  Process: line__346.
# KERNEL: stopped at time: 11195 ns
```

Figure 5: Console output for Algorithm NO.1 model simulation, with the introduced error

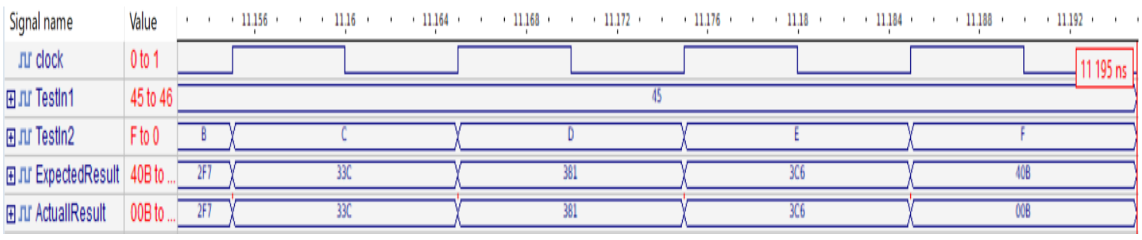
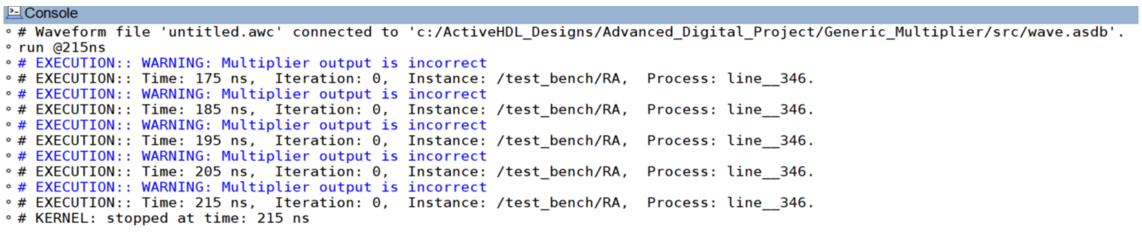


Figure 6: A zoom in for the simulation of Algorithm NO.1 model, with the introducing error

Discussion: from the console output of algorithm NO.1 model The simulation was run for 11195 ns, and there was an error in the output of the Generic multiplier model 1 output, which it was detected at the rising edge of the clock at 11195 ns, and this error can be also noticed from the waveform of the simulation, so the last value of the output was inconsistent with the expected result, and that is expected since an error was introduced in the design which is :

```
res(j+k-1 downto k-1) <= '0' & temp_out(k-2); -- the introduced error.
```

so here the correct implementation require replacing '0' with Cout(k-2).



```
# Waveform file 'untitled.awc' connected to 'c:/ActiveHDL_Designs/Advanced_Digital_Project/Generic_Multiplier/src/wave.asdb'.
run @215ns
# EXECUTION:: WARNING: Multiplier output is incorrect
# EXECUTION:: Time: 175 ns,  Iteration: 0,  Instance: /test_bench/RA,  Process: line__346.
# EXECUTION:: WARNING: Multiplier output is incorrect
# EXECUTION:: Time: 185 ns,  Iteration: 0,  Instance: /test_bench/RA,  Process: line__346.
# EXECUTION:: WARNING: Multiplier output is incorrect
# EXECUTION:: Time: 195 ns,  Iteration: 0,  Instance: /test_bench/RA,  Process: line__346.
# EXECUTION:: WARNING: Multiplier output is incorrect
# EXECUTION:: Time: 205 ns,  Iteration: 0,  Instance: /test_bench/RA,  Process: line__346.
# EXECUTION:: WARNING: Multiplier output is incorrect
# EXECUTION:: Time: 215 ns,  Iteration: 0,  Instance: /test_bench/RA,  Process: line__346.
# KERNEL: stopped at time: 215 ns
```

Figure 7: Console output for Algorithm NO.2 model simulation, with the introduced error

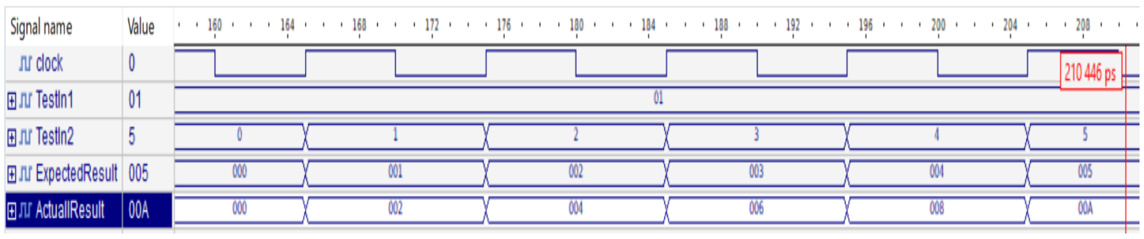


Figure 8: A zoom in for the simulation of Algorithm NO.2 model, with the introducing error

Discussion: from the console output of algorithm NO.2 model the simulation was run for 215 ns, and there were errors in the output of the Generic multiplier model 2 output at each rising edge of the clock starting from 175 ns to the end of the simulation, and these errors, can be also noticed from the waveform of the simulation, so the last 5 values of the output were inconsistent with the expected result, and that is expected since an error was introduced in the design which is :

```
res <= A(j-1) & Q(j-1); -- the introduced error
```

so here the correct implementation require replacing (j-1) with (j).

6 CONCLUSION AND FUTURE WORK

In this project, I succeed in implementing the two previously mentioned algorithms structurally, to build a model of the generic multiplier, Also I have solved all the problems that might face my model, like if K = 1 especially in the first algorithm, and I have written a simple and a brief VHDL code, so anyone can understand the steps in the implementation. Finally, this project encouraged me to dive more into VHDL coding, and Digital design and testing.

7 APPENDICES

7.1 APPENDIX A -BASIC GATES-

```
-----------------------------------------------------------
-- implemnation of the And gate with two inputs.

-- Defining needed Libraries
library ieee;
use ieee.std_logic_1164.all;

-- defining the entity of the 2-inputs And
entity And2 is
  port(In0, In1: in std_logic;
    res: out std_logic);
end entity And2;

-- defining and building the architecture of the 2-inputs And
architecture And2 of And2 is
begin
  res <= In0 and In1;
end architecture And2;

-----------------------------------------------------------
-- implemnation of the OR gate with two inputs.

-- Defining needed Libraries
library ieee;
use ieee.std_logic_1164.all;

-- defining the entity of the 2-inputs OR
```

```
entity Or2 is
  port(In0, In1: in std_logic;
    res: out std_logic);
end entity Or2;

-- defining and building the architecture of the 2-inputs OR
architecture Or2 of Or2 is
begin
   res <= In0 or In1;
end architecture Or2;



-----------------------------------------------------------
-- implemnation of the XOR gate with two inputs.

-- Defining needed Libraries
library ieee;
use ieee.std_logic_1164.all;

-- defining the entity of the 2-inputs XOR
entity Xor2 is
  port(In0, In1: in std_logic;
    res: out std_logic);
end entity Xor2;

-- defining and building the architecture of the 2-inputs XOR
architecture Xor2 of Xor2 is
begin
   res <= In0 xor In1;
end architecture Xor2;
```

```vhdl
-----------------------------------------------------------
-- Structural implementation of the one bit adder.

-- Defining needed Libraries
library ieee;
use ieee.std_logic_1164.all;

-- defining the entity of the 1-bit adder
entity One_Bit_Adder is
  port(In0, In1, Cin: in std_logic;
    Sum, Cout: out std_logic);
  --  in0 and in1 are the inputs to be added, and Cin is the carry in.
  -- sum is the container of the addition result, and cout is for the carry out
      of the addition.
end entity One_Bit_Adder;


-- defining and building the architecture of the 1-bit adder
architecture structural_One_Bit_Adder of One_Bit_Adder is
signal s0, s1, s2: std_logic;
begin

  Xor_In0_In1: entity work.Xor2(Xor2) port map (In0, In1, s0);
  Xor_S0_Cin: entity work.Xor2(Xor2) port map (s0, Cin, Sum);
  And_In0_In1: entity work.And2(And2) port map (In0, In1, s1);
  And_S0_Cin: entity work.And2(And2) port map (s0, Cin, s2);
  Or_S1_S2: entity work.Or2(Or2) port map (S1, S2, Cout);
  -- this is eqivelant to => cout & Sum <= in0 + in1 + Cin.

end architecture structural_One_Bit_Adder;
```

```
-----------------------------------------------------------
-- structural implementation of n-bit adder using generic parameter ’n’ and
    One_Bit_Adder entity
library ieee;
use ieee.std_logic_1164.all;



-- defining the entity of the n-bit adder
ENTITY n_bit_adder IS
  GENERIC ( n: positive ); -- n -> generic parameter
  PORT (X, Y: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
        Cin:  IN STD_LOGIC;
      Sum: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
      Cout: OUT STD_LOGIC); -- finish declaring the ports of the entity
    -- X and Y are the inputs to be added together.
    -- cin is the carry in.
    -- Sum is the register for the resulted sum.
    -- Cout is the carry out of the addition.
END ENTITY n_bit_adder;

-- internal functional of the n-bit adder
architecture structural_N_Bit_Adder of N_Bit_Adder is
signal Carry: std_logic_vector(n downto 0); -- carry signal to use in a the
    carry (in or out) for each one-bit-adder
begin

  Carry(0) <= Cin; -- the first element in carry is the cin
  -- the last element in carry represents the carry out (final carry)resulted
     from the addition of the last bit
  Cout <= Carry(n);

  -- add every two bits of the n-bit numbers X(i) and Y(i)
  -- and using the previous geenrated carry as the carry in for each current
      level of addition.
  generateLoop1: FOR i IN 0 TO n-1 GENERATE
       One_Bit_Adder_GateI: ENTITY work.One_Bit_Adder(structural_One_Bit_Adder)
           PORT MAP (X(i),Y(i),Carry(i),Sum(i),Carry(i+1));
     END GENERATE generateLoop1;

end architecture structural_N_Bit_Adder;




-----------------------------------------------------------
-- implemnation of the 2X1MUX circuit.

-- Defining needed Libraries
library ieee;
use ieee.std_logic_1164.all;

-- defining the entity of the 2X1MUX
entity MUX2X1 is
  generic (n,m: positive);
  port(selection: std_logic;
      Input0, Input1: in std_logic_vector(n+m downto 0);
    output: out std_logic_vector(n+m downto 0));
  -- the first signal "selection" is the selection line of the mux,
  -- input0 and input1 is two signals that the mux will choose one of them
      based on "selection" value, each of them are m+n bit
  -- output is the result of the max which is either input0 or input1, and it's
      also m+n bits number.
end entity MUX2X1;

-- defining and building the architecture of the 2X1MUX
architecture MUX2X1 of MUX2X1 is
signal out1, out2: std_logic_vector(n+m downto 0);
begin
  output <= input0 when selection ='0' else input1;
  -- if selection = 0, then the output will take the first signal "input0",
      else it will take the second signal "input1".
end architecture MUX2X1;




-----------------------------------------------------------
-- implemnation of a special model for the right shift, such that it's
    euivelent to the usuall right-shift with an additional
-- features, such that the new left-most bit will be equal to new_left_most_bit
    signal.

-- Defining needed Libraries
library ieee;
use ieee.std_logic_1164.all;

-- defining the entity of the special_right_shift.
entity special_right_shift is
  generic (K: positive);
  port(new_left_most_bit: std_logic;
      Input: in std_logic_vector(K-1 downto 0);
    output: out std_logic_vector(K-1 downto 0));
```

```
end entity special_right_shift;

-- defining and building the architecture of the special_right_shift.
architecture special_right_shift of special_right_shift is
begin
  --output(K-1) <= new_left_most_bit; -- the left most bit take the value of
      new_left_most_bit signal
--
-- -- the work of this loop is identical to output(K-2 downto 0) <= input(k-1
    downto 1)
-- generateForLoop: for i in 0 to K-2 generate
--    begin
--      output(i) <= Input(i+1);
--    end generate generateForLoop;
  output <= new_left_most_bit & input(k-1 downto 1);
  -- the final result will be identical to the following instruction in VHDL :
      output <= new_left_most_bit & input(k-1 downto 1)
end architecture special_right_shift;
```

7.2 APPENDIX B -GENERIC MULTIPLIER IMPLEMENTATION-

```
-----------------------------------------------------------
-- structural implementation of the generic multiplier (K-bit*J-bit)
-- using generic parameters 'k&J' with two differnet algorithms.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- defining the entity of the generic-multiplier.
entity Generic_Multiplier is

  Generic (J, k: positive); -- geenric parameters where j and k represent the
      number of bits in X and Y respectively.
  port (X: in std_logic_vector(j-1 downto 0);
     Y: in std_logic_vector(k-1 downto 0);
     res: out std_logic_vector((k+j-1) downto 0)); -- defining the port of the
        generic multiplier entity.
    -- X is the first multiplicand, and Y is the multiplier.
    -- res is the reuslt of multiplying X by Y.

end entity Generic_Multiplier;

-- implemenation of the generic multiplier using the first algorithm (Parallel
    Binary multiplication method),
-- which is depend mainly on the usage of the n-bit adder for k-1 stages.
-- the inputs of each n-bit adder will be the and of Y(i) with X and the
    resulted cout&sum(n-1 downto1) from the result of n-bit-adder from the
    previous stage
```

```
-- expect for the first stage the first and the second operand will be the and
    of Y(0) with X & the and of Y(1) with X.

architecture Structural_Generic_Multiplier_Algorithm_NO_1 of Generic_Multiplier
    is

Type array_1 is array (0 to k) of std_logic_vector(j-1 downto 0); -- array_1 is
    specified for registers temp_in_1, temp_in_2 and temp_out that used in the
    addition operation
Type array_2 is array (0 to k-1) of std_logic; -- -- array_1 is specified for
    registers of kind Cout (the carry out of each addition operation)

signal temp_out, temp_in_1, temp_in_2: array_1 :=(others=> (others=>'0'));
signal Cout: array_2;
signal temp1, temp2: std_logic_vector(j-1 downto 0);
signal zeroBit: std_logic:='0';

begin
  -- generate the first input for the addition process (which is the and
      between bits of X and Y(0))
  gen10: for i in 0 to j-1 Generate
  begin
    andJ101: ENTITY work.And2(And2) PORT MAP (X(i), Y(0), temp1(i));
    end generate ;
  res(0) <= temp1(0); -- the right most bit in the result of the and between X
      and Y(0) is given to the firs bit in the final result.
  ShifterI1: entity work.special_right_shift(special_right_shift) generic map
      (j) port map (zeroBit, temp1, temp_in_1(0));-- the first input to the
      adder is the right shift of the result of the and.
  -- loop k-1 times
  gen1: FOR i IN 1 TO k-1 GENERATE
    BEGIN
    -- preparing the second input for the n-bit adder which is the and between
        bits of X and Y(i+1).
    gen12: for L in 0 to j-1 Generate
    begin
      andJ121: entity work.And2(And2) port map (X(L), Y(i), temp_in_2(i-1)(L));
    end generate gen12;
    -- after the inputs to the n-bit adder being ready, the addition process
        come to take a place, and here we use n-bit addeer such that n=j.
    AdderI: entity work.N_Bit_Adder(structural_N_Bit_Adder) generic map (j)
                                port map (temp_in_1(i-1), temp_in_2(i-1), '0',
                                  temp_out(i-1), Cout(i-1));
    res(i) <= temp_out(i-1)(0); -- added a new bit (first bit in the output of
        the adder) to the final result.
    temp_in_1(i) <= Cout(i-1) & temp_out(i-1)(j-1 downto 1);
    ShifterI1: entity work.special_right_shift(special_right_shift) generic map
        (j) port map (Cout(i-1), temp_out(i-1), temp_in_1(i));-- update the
        value of the the next first input to the n-bit adder through the
        special right shift model
    END GENERATE;
  if_K_Is_1:
  if(k=1) generate
  begin
    res <= '0' & temp1;
  end generate if_K_Is_1;
  if_K_Is_Not_1:
  if(k/=1) generate
  begin
    res(k+j-1 downto k-1) <= Cout(k-2) & temp_out(k-2); -- after the last
        addition operation, update the value of the last K-bits.
    --res(k+j-1 downto k-1) <= '0' & temp_out(k-2); -- the introduced error.
  end generate if_K_Is_Not_1;
end architecture Structural_Generic_Multiplier_Algorithm_NO_1;







-- implemenation of the generic multiplier using thr second algorithm (Add and
    shift method),
-- which happens on many levels (j levels), and the output for each level is
    either the shift for C&A$Q to the right
-- or the addition of Y and A, and then shift C&A&Q to the right.

-- depending on the value of Q[i].
-- where C is a one-bit register, A is a j-bit register and Q is a j-bit
    register.
-- After finishing all the iteration the final result will be held in A & Q
    respectively.

architecture Structural_Generic_Multiplier_Algorithm_NO_2 of Generic_Multiplier
    is
-- declaring 4 types of Arrays, each is specified for certain registers.
Type array_1 is array (0 to j) of std_logic_vector(j-1 downto 0); -- array_1 is
    specified for registers of Kind Q
Type array_2 is array (0 to j) of std_logic_vector(k-1 downto 0); -- array_2 is
    specified for registers of kind A
Type array_3 is array (0 to j) of std_logic;      -- array_3 is specified for
    registers of kind C
Type array_4 is array (0 to j) of std_logic_vector(k+j downto 0); -- array_4 is
    specified for temp_in_1, temp_in_2 and temp_out that's specified for the
    mux operation.
```

```
signal Q, Q_Shift, Q_Addition_Shift: array_1 :=(others=> (others=>'0'));
-- Q contains the final reuslt of Q register after each iteration (which is
    eaither Q_Shift or Q_Addition_Shift).
-- Q_shift contains the result of Q after each iteration, by considering the
    current action is shidt only
-- Q_Addition_shift contains the reuslt of Q after each iteration, by
    considering the current action is Addition then shift.

signal A, A_shift, A_Add, A_Addition_Shift: array_2 :=(others=> (others=>'0'));
-- A contains the final reuslt of A register after each iteration (which is
    eaither A_Shift or A_Addition_Shift).
-- A_shift contains the result of A after each iteration, by considering the
    current action is shidt only
-- A_Add contsins the reuslt of A after eahc iteration, by considering the
    Addition process for Addition and shift action.
-- A_Addition_shift contains the reuslt of A after each iteration, by
    considering the current action is Addition then shift.

signal C, C_Shift, C_Add, C_Addition_Shift: array_3:= (others=> '0');
-- C contains the final reuslt of C register after each iteration (which is
    eaither C_Shift or C_Addition_Shift).
-- C_shift contains the result of C after each iteration, by considering the
    current action is shidt only
-- C_Add contsins the reuslt of C after eahc iteration, by considering the
    Addition process for Addition and shift action.
-- C_Addition_shift contains the reuslt of C after each iteration, by
    considering the current action is Addition then shift.

signal temp_in_1, temp_in_2, temp_out: array_4 := (others=> (others=> '0'));
-- temp_in_1 is a temporarily register for the first input of the 2X1mux
```

```
    (C_Shift(i) & A_Shift(i) & Q_Shift(i)).
-- temp_in_2 is a temporarily register for the second input of the 2X1mux
    (C_Addition_Shift(i) & A_Addition_Shift(i) & Q_Addition_Shift(i)).
-- temp_out is a temporarily register for the output of the 2X1mux
    (C(i+1)&A(i+1)&Q(i+1)).
begin
  Q(0) <= X; -- initalize Q(0) to be X, while the initalization of the rest
      registers done above.
  gen1: for i in 0 to j-1 generate -- to loop through the bits of Q register (j
      times).
    begin
      -- stage1_1: First consider the action for the of Q(i)(0) is shift only
      C_shift(i) <= '0'; -- do the right shift for C register and store the
          reuslt is C_shift.
      ShifterI1: entity work.special_right_shift(special_right_shift) generic
          map (k) port map (C(i), A(i), A_Shift(i));-- do the right shift for A
          register and store the reuslt is A_shift, also take of C(i) as the one
          that will be added to the left of A_shift.
      ShifterI2: entity work.special_right_shift(special_right_shift) generic
          map (j) port map (A(i)(0), Q(i), Q_Shift(i)); -- do the right shift
          for Q register and store the reuslt is AQ_shift, also take of A(i)(0)
          as the one that will be added to the left of Q_shift.

      -- stage1_2: Second consider the action for the bit of Q(i)(0) is Add
          then shift.
      AdderI: entity work.N_Bit_Adder(Structural_N_Bit_Adder) generic map(k)
          port map(A(i), Y, C(i), A_Add(i), C_Add(i)); -- do the addition
          between A(i) and Y.
      -- after the addition do the same as in stage1_1, which is the shift of
          C&A_Add&Q_Add.
      C_Addition_shift(i) <= '0';
      ShifterI3: entity work.special_right_shift(special_right_shift) generic
          map (k) port map (C_Add(i), A_Add(i), A_Addition_Shift(i));
      ShifterI4: entity work.special_right_shift(special_right_shift) generic
          map (j) port map (A_Add(i)(0), Q(i), Q_Addition_Shift(i));

      -- stage2 : Now the turn come to select one of the results of the above
          two sub_stages, by using a 2X1MUX with a selction = Q(i)(0).
      temp_in_1(i) <= C_Shift(i) & A_Shift(i) & Q_Shift(i);
      temp_in_2(i) <= C_Addition_Shift(i) & A_Addition_Shift(i) &
          Q_Addition_Shift(i);
      Mux2X1: entity work.MUX2X1(MUX2X1) generic map(j,k) port map(Q(i)(0),
          temp_in_1(i), temp_in_2(i), temp_out(i));
      -- finally update the values of C(i+1), A(i+1) and Q(i+1) based on
          temp_out
      C(i+1) <= temp_out(i)(k+j);
      A(i+1) <= temp_out(i)(k+j-1 downto j);
      Q(i+1) <= temp_out(i)(j-1 downto 0);
    end generate gen1;
  res <= A(j) & Q(j); -- the final result is held in A(j) & Q(j).
  --res <= A(j-1) & Q(j-1); -- the introduced error
end architecture Structural_Generic_Multiplier_Algorithm_NO_2;
```

7.3 APPENDIX C -TESTING SYSTEM-

```
-- result analyszer
library ieee;
use ieee.std_logic_1164.all;

-- declaring the entity of the result analyzer.
entity result_analyzer is
  generic (k,j : positive); -- using two generic parameters k and j
  port(clock: in std_logic;
    ExpectedResult, ActuallResult: in std_logic_vector (k+j-1 downto 0)
    );
end entity result_analyzer;

architecture result_analyzer OF result_analyzer is
begin
    process(clock)
    begin
      if rising_edge(clock) then
        -- Check whether Generic_multiplier output matches expectation which wa
            getten from using * oerator (X*Y).
        assert  ExpectedResult = ActuallResult -- if ExpectedResult !=
            ActuallResult then assert a report of warning severity
        report  "Multiplier output is incorrect" -- in case of an error this
            meesage will be displayed into the consle
        severity WARNING;
      end if;
    end process;
end architecture result_analyzer;
```

```
-- test generater
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- entity declartion of test generator
entity test_generator is
  generic (k,j: positive); -- two generic parameters K and J.
  port(clock: in std_logic;
    TestIn1: out std_logic_vector(k-1 downto 0);
    TestIn2: out std_logic_vector(j-1 downto 0);
    ExpectedResult: out std_logic_vector(j+k-1 downto 0)
  );
  -- TestIn1 and TestIn2 are the two inputs for the generic multiplier.
  -- ExpectedResult is the result of the multiplication of the two numbers.
end entity test_generator;

-- description of the test generator
-- this test generator loops through all possible cases of the two inputs for
    the multiplier (based on their number of bits)
-- and generat the test inputs for each possible case, and the expected result
    of the multiplication.
architecture test_generator of test_generator is
begin
  process
    begin
      for I in 0 to 2**k-1 loop -- this loop to generate all possible value of
          the first number (which have a range from 0 to (2^k)-1).
          for L in 0 to 2**j-1 loop -- this loop to generate all possible value
              of the second number (which have a range from 0 to (2^j)-1).
              -- set the inputs to the multiplier, by converting I and L into a
                 bonary std_logic vector of K and J bits respectively.
              TestIn1 <= conv_std_logic_vector(i,k);
              TestIn2 <= CONV_STD_LOGIC_VECTOR(L,j);
              -- Calculate what the output of the multiplier should be
              ExpectedResult <= CONV_STD_LOGIC_VECTOR(I*L,k+j);
              -- Wait until multiplier output has settled
              wait until rising_edge(clock);
            end loop;
          end loop;
      wait;
    end process;

end architecture test_generator;



-- test bench for each algorithm
library ieee;
use ieee.std_logic_1164.all;
-- entity of test bench
entity test_bench is
end entity test_bench;

-- test bench architecture of the first algorithm.
architecture test_bench_1 of test_bench is
signal clock: std_logic:='0'; -- clock signal used here for running the
    test_generator at each rising edge.
signal TestIn1: std_logic_vector(6 downto 0); -- the first input of the
    multiplier
signal TestIn2: std_logic_vector(3 downto 0); -- the seocnd input of the
    multiplier
```

```
signal ExpectedResult, ActuallResult: std_logic_vector(10 downto 0); -- the
    escpected and actuall result of the multiplier
begin

 -- Place one instance of test generator (from this model we can generate the
     test inputs and the expected result)
  TG: entity work.test_generator(test_generator) generic map (7,4)
                                                port map(clock, TestIn1, TestIn2,
                                                    ExpectedResult);
 -- Place one instance of the Unit Under Test which is the generic multiplier
     fot he first algorithm
 ---UUT: entity work.m(mm) generic map (7,4) port map(TestIn1, TestIn2,
     ActuallResult);
   -- Place one instance of the Unit Under Test which is the generic multiplier
       fot he second algorithm
    UUT: entity
        work.Generic_Multiplier(Structural_Generic_Multiplier_Algorithm_NO_1)
        generic map (7,4)
                                                                                    port
                                                                                        map(TestIn1,
                                                                                        TestIn2,
                                                                                        ActuallResult)


 -- Place one instance of result analyzer (the one that will check if the
     result obtained from the built model is as the expected result or not)
    RA: entity work.result_analyzer(result_analyzer) generic map (7,4)
                            port map(clock, ExpectedResult, ActuallResult);
  clock <= not clock after 5 ns; -- flip the clock after each 5 ns.

end architecture test_bench_1;
```

```
-- test bench architecture of the first algorithm.
architecture test_bench_2 of test_bench is
signal clock: std_logic:='0'; -- clock signal used here for running the
    test_generator at each rising edge.
signal TestIn1: std_logic_vector(6 downto 0); -- the first input of the
    multiplier
signal TestIn2: std_logic_vector(3 downto 0); -- the second input of the
    multiplier
signal ExpectedResult, ActuallResult: std_logic_vector(10 downto 0); -- the
    escpected and actuall result of the multiplier
begin

 -- Place one instance of test generator (from this model we can generate the
     test inputs and the expected result)
  TG: entity work.test_generator(test_generator) generic map (7,4)
                                              port map(clock, TestIn1, TestIn2,
                                                  ExpectedResult);
 -- Place one instance of the Unit Under Test which is the generic multiplier
     fot he second algorithm
    UUT: entity
        work.Generic_Multiplier(Structural_Generic_Multiplier_Algorithm_NO_2)
        generic map (7,4)
                                                                                               port
                                                                                                   map(TestIn1,
                                                                                                   TestIn2,
                                                                                                   ActuallResult)

 -- Place one instance of result analyzer (the one that will check if the
     result obtained from the built model is as the expected result or not)
    RA: entity work.result_analyzer(result_analyzer) generic map (7,4)
                        port map(clock, ExpectedResult, ActuallResult);
  clock <= not clock after 5 ns; -- flip the clock after each 5 ns.


end architecture test_bench_2;






-----------------------------------------------------------
```