Concordia University
Engineering and Computer Science

Fall 2022

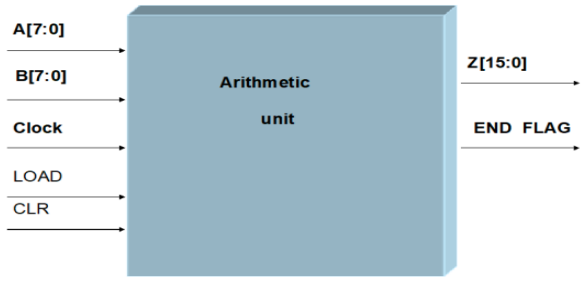
COEN 6501: Digital Design and Synthesis

Dr. Sébastien Le Beux

Project Report

Arithmetic Unit Implementation, which can calculate

Z = ¼ [A*B] + 1



Team Members

Akash Rajmohan (**40218469**)

Prithvik Adithya Ravindran **(40195464)**

Krupal Jayantibhai Dudhat **(40206078**)

Table of Contents

Table of Figures

1. Abstract

The task is to design an arithmetic unit capable of performing the Z=1/4 (A*B) operation +1. An array multiplier, a ripple carry adder using half adders and full adders, shifting right, and incremental operations are used to design this project. A and B are two unsigned 8-bit inputs, and Z is a 16- bit output. We have implemented this project using a basic array multiplier to implement this project.

First, the two 8-bit inputs are multiplied and then fed for the shifting operation. Once the shifting operation is done, then the increment by '1' operation is done. We have used positive edge-triggered D-Flipflops to create two 8-bit input registers and one 16-bit output register.
This project has been implemented using structural modelling in VHDL.

The program is simulated using ModelSim. RTL schematics are obtained from RTL synthesis.

2. Introduction

The project's main objective is to design an arithmetic unit to calculate the **Z=1/4 (A*B) +1**. The arithmetic unit consists of an array multiplier, which takes two input 8-bit unsigned numbers and generates a 16-bit output which is later right-shifted and incremented. When the LOAD signal changes from HIGH to LOW, the input values A and B are stacked into the registers Register_8bit_A and Register_8bit_B, respectively. The final output is shown in Register_16bit. The registers used in the design are asynchronous reset positive edge triggered registers.

Multiplication is an important aspect of arithmetic operations. The multiplier's performance decides the computational capability of a system. With the wide variety of multipliers available in the market, it's always confusing when it comes to choosing a multiplier. It always depends on the task and the requirements to be met. Here in our case, we have used an array multiplier for simplicity and scalability purposes for the design.

3. Project Description

This project aims to design an arithmetic unit capable of calculating Z=1/4 [A×B] +1. The unit will receive two unsigned 8-bit operands, A and B. Two clock synchronous internal registers (RA and RB) latch the data when LOAD = 1. The unit outputs the results in a 16-bit register RZ output port, and END_FLAG rises. An asynchronous CLEAR signal will clear all the registers to '0'. The design shall be structural, and the choice of type of multiplier is left to the student.

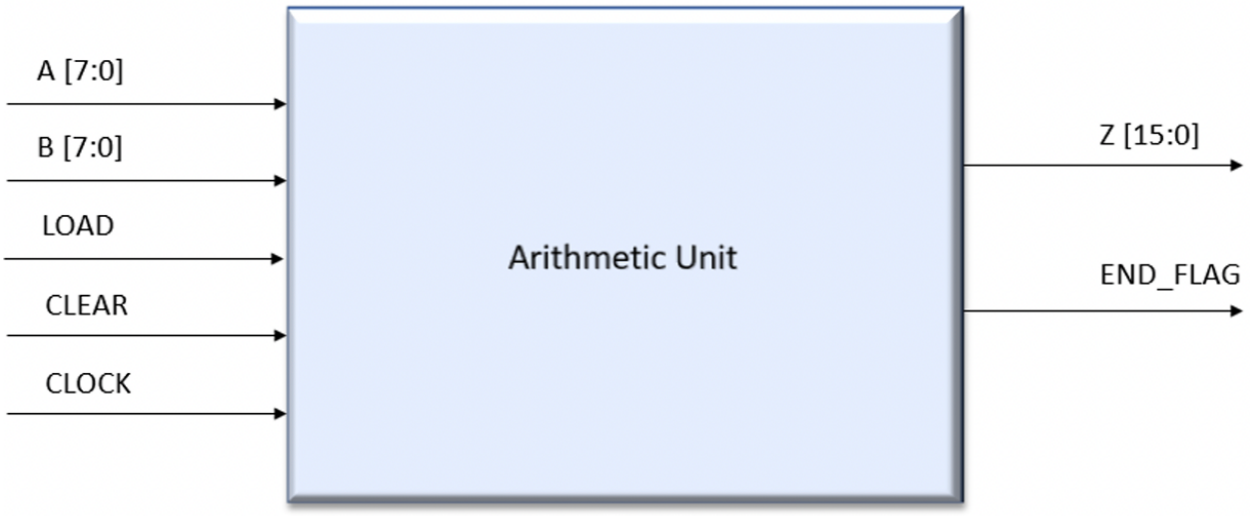


Figure 1 Black Box of the Design.

Signal Specifications

A: Unsigned 8-bit Operand A

B: Unsigned 8-bit Operand B

Z: Signed Output

CLEAR: Clears selected registers

LOAD: Loads Operand into internal registers

CLOCK: Input Clock

END_FLAG: Indicates end of operation

4. Design of Functional Blocks

To get started with the design of this project. We were required to design basic functional blocks to go further. We started with basic logic gates and using them; we made further progress by building essential components for the design.

We designed the following components to complete the design of the arithmetic unit capable of calculating the expression Z=1/4 (A*B) +1.

a. Half Adder
b. Full Adder
c. 8-Bit Array Multiplier
d. Array Multiplier
e. Partial Products Creation
f. ¼ Operation (Using Shifting Right)
g. Incrementation Operation
h. 8-Bit and 16-Bit Registers
i. END Flag Design

By following this above order, we were able to design the unit and come to a place where we get to understand each unit's functionality independently and as a collective unit.

4.1 Half Adder

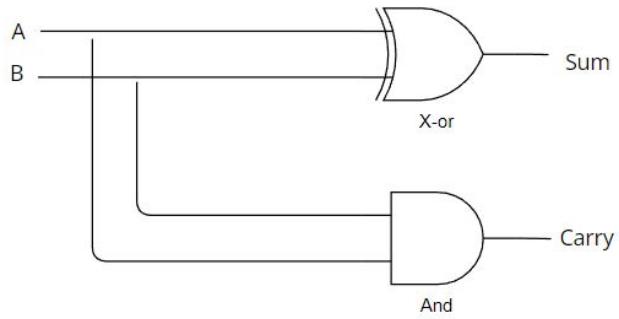


Figure 2 Half Adder Conceptual diagram

The half adder is a combinational circuit which takes two inputs and gives sum and carry as the output.

```
13  architecture half_adder_arch of half_adder is
14
15    component AND_gate is
16      Port (
17      a : in STD_LOGIC;
18      b : in STD_LOGIC;
19      Y : out STD_LOGIC);
20    end component;
21
22    component xor_gate is
23      Port (
24      a : in STD_LOGIC;
25      b : in STD_LOGIC;
26      Y : out STD_LOGIC);
27    end component;
28
29    begin
30      HA_xor: xor_gate
31      port map (a=>A,b=>B,Y=>S);
32
33      HA_and: AND_gate
34      port map (a=>A,b=>B,Y=>Cout);
35
36    end half_adder_arch;
37
38
39
40
```

Figure 3 Code for the Half Adder.

We have written the code structurally in VHDL, as shown above.

```
constant data_array: test_vector_data(3 downto 0) :=
(('0','0','0','0'),
('0','1','1','0'),
('1','0','1','0'),
('1','1','0','1'));

component half_adder is
  port(
    A, B: in std_logic;
    S, Cout: out std_logic);
end component;

signal test_a, test_b, test_s, test_cout: std_logic;
signal test_ok: boolean;

begin
  uut:half_adder
  port map(A=>test_a, B=>test_b, S=>test_s, Cout=>test_cout);

    process begin
      for i in data_array'range loop
        test_a <= data_array(i).test_a;
        test_b <= data_array(i).test_b;
        wait for 10 ns;
        if ((test_s /= data_array(i).test_s) and (test_cout /= data_array(i).test_cout)) then
          test_ok <= FALSE;
        else
          test_ok <= TRUE;
        end if;
```

Figure 4 Testbench for the Half Adder.

The half adder was verified with the above testbench, and the below timing diagrams were obtained.

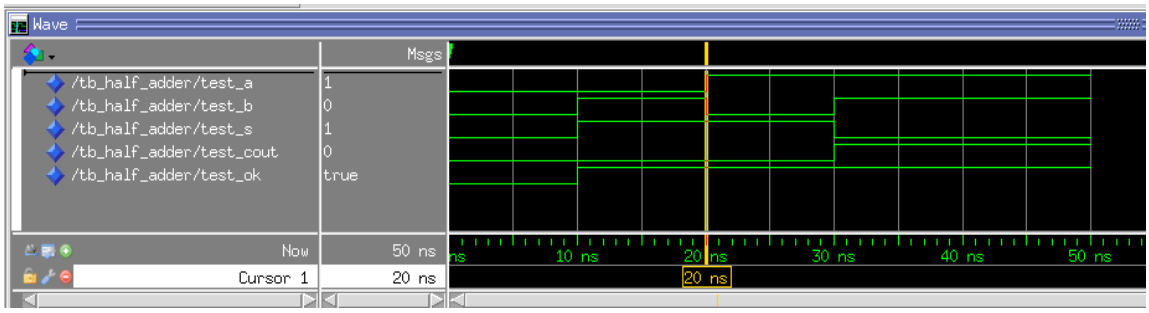


Figure 5 Timing Diagram for Half Adder.

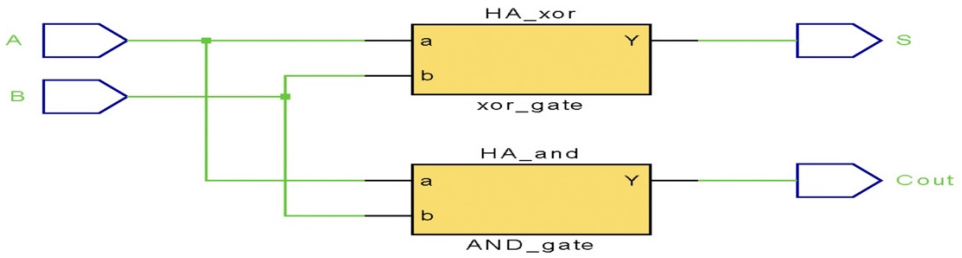


Figure 6 Half Adder synthesis diagram.

4.2 Full Adder

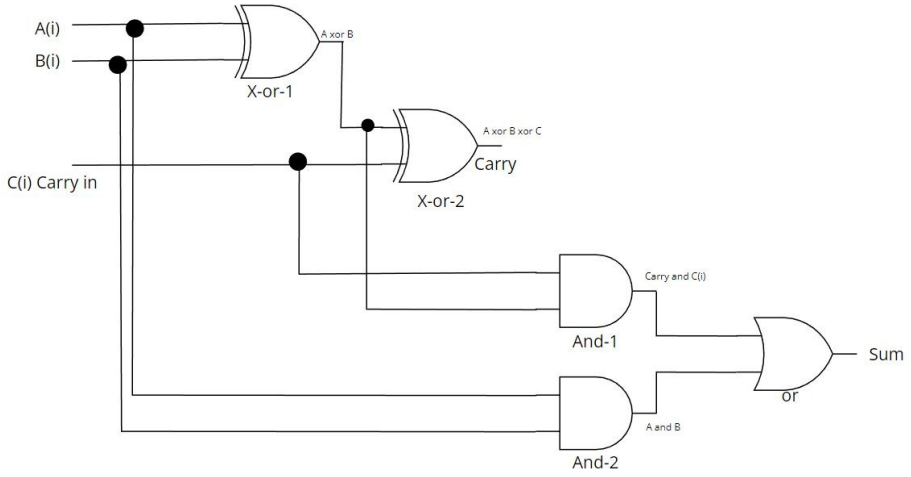


Figure 7 Conceptual Diagram for Full Adder.

The full adder circuit is one of the most important circuits in digital electronics. It can perform both addition and subtraction. It's called a full adder because it adds two binary digits and a carry-in digit to produce a sum and a carry-out digit as output. So technically, it has three inputs and two outputs in total.

```
architecture full_adder_1bit_arch of full_adder_1bit is

    component AND_gate is
      Port ( a : in STD_LOGIC;
      b : in STD_LOGIC;
      Y : out STD_LOGIC);
    end component;

    component OR_gate is
      Port ( a : in STD_LOGIC;
      b : in STD_LOGIC;
      Y : out STD_LOGIC);
    end component;

    component xor_gate is
      Port (
      a : in STD_LOGIC;
      b : in STD_LOGIC;
      Y : out STD_LOGIC);
    end component;

  signal xor_1_out, and_1_out, and_2_out: std_logic;

  begin
    xor_1: xor_gate
    port map(a=>A, b=>B, Y=>xor_1_out);

    xor_2: xor_gate
    port map(a=>xor_1_out, b=>Cin, Y=>S);

    and_1: AND_gate
    port map(a=>xor_1_out, b=>Cin, Y=>and_1_out);

    and_2: AND_gate
    port map(a=>A, b=>B, Y=>and_2_out);

    or_1: OR_gate
    port map(a=>and_1_out, b=>and_2_out, Y=>Cout);

  end full_adder_1bit_arch;
```

Figure 8 Code for the Full Adder.

The code written VHDL for the full adder is above. It's written structurally in VHDL.

```
constant data_array: test_vector_data(7 downto 0) :=
(('0','0','0','0','0'),
('0','0','1','1','0'),
('0','1','0','1','0'),
('0','1','1','0','1'),
('1','0','0','1','0'),
('1','0','1','0','1'),
('1','1','0','0','1'),
('1','1','1','1','1'));

component full_adder_1bit is
  port(
    A, B, Cin: in std_logic;
    S, Cout: out std_logic);
end component;

signal test_a, test_b, test_cin, test_s, test_cout: std_logic;
signal test_ok: boolean;

begin
  uut:full_adder_1bit
  port map(A=>test_a, B=>test_b, Cin=>test_cin, S=>test_s, Cout=>test_cout);

    process begin
      for i in data_array'range loop
        test_a <= data_array(i).test_a;
        test_b <= data_array(i).test_b;
        test_cin <= data_array(i).test_cin;
        wait for 10 ns;
        if ((test_s /= data_array(i).test_s) and (test_cout /= data_array(i).test_cout)) then
          test_ok <= FALSE;
        else
          test_ok <= TRUE;
        end if;
```

Figure 9 Testbench for the Full Adder.

The full adder was verified with the above testbench, and the timing diagrams were obtained.

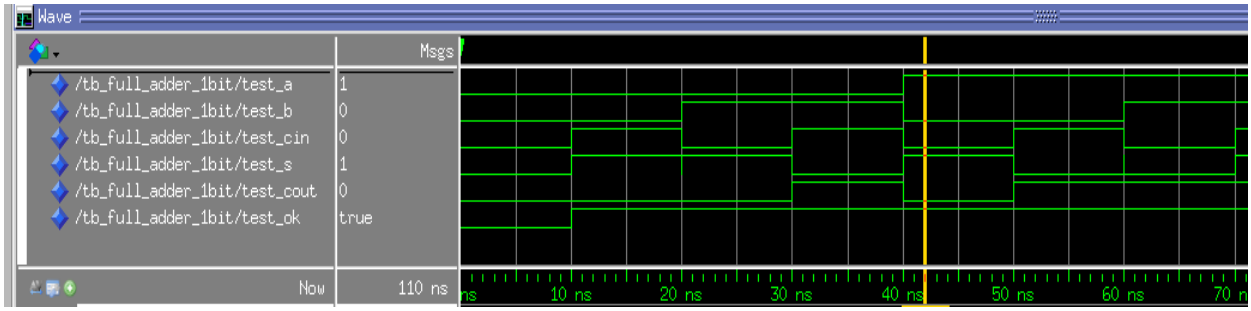


Figure 10 Testbench for the Full Adder.

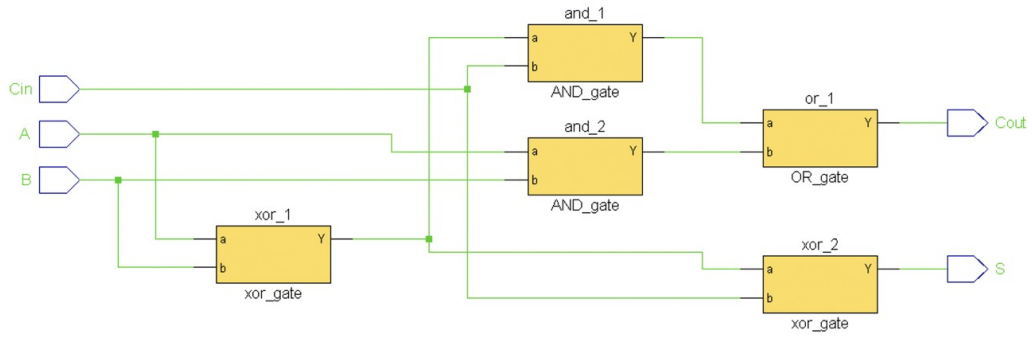


Figure 11 Full adder synthesis diagram

4.3 8-Bit Array Multiplier

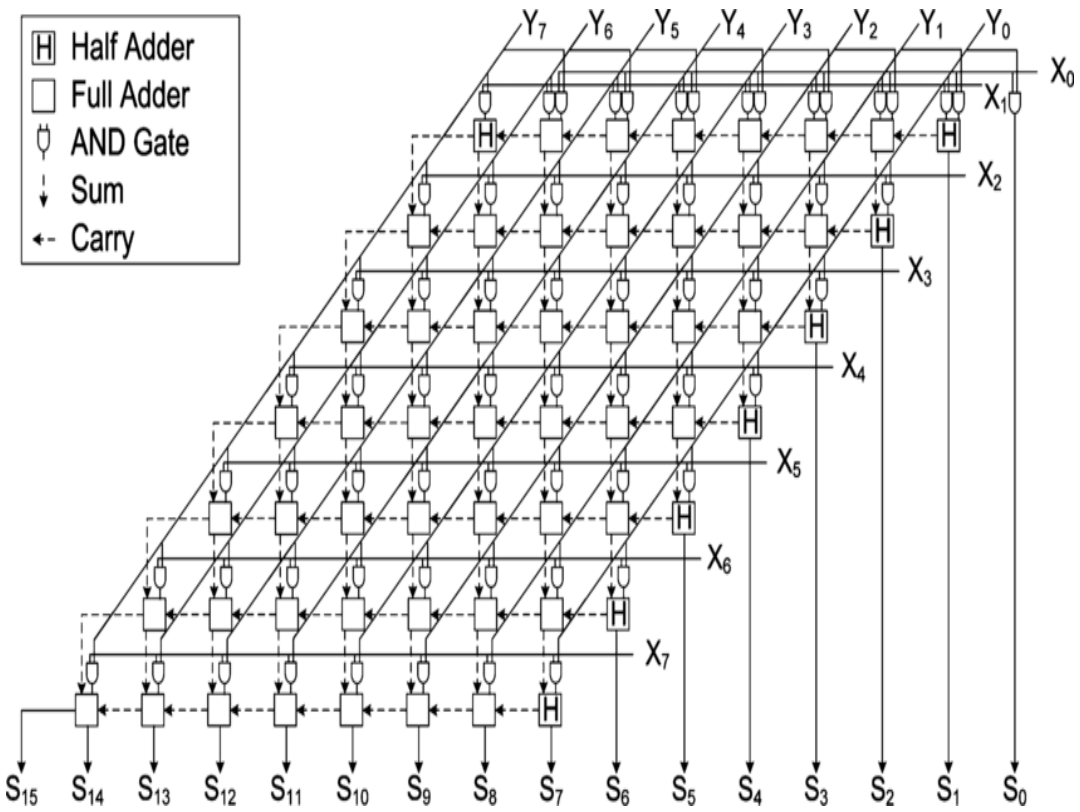


Figure 12 8-bit array multiplication block diagram.[1]

We have designed an 8-bit array multiplier following the above array multiplier block diagram. The diagram shows that we have used 64 logic-and gates, 48 full adders and eight half-adders. The array multiplier works on the concept of carry and shift. As the inputs are 8 bits unsigned, the resulting output from the circuit will be 16 bits unsigned. As seen in the diagram, the initial bits of the inputs are directly given to, the logic-and gate for the bit-wise multiplication, and the resulting output from it will be fed to the half adders (initial bits). The remaining bits will be fed to the full adders. The diagram shows that the sum of the half adders(initial) will be directly available as the output bits (S0 to S7). The carry will be shifted to the next full adder, and the sum generated by the full adders will be shifted to the full adder below it. Bits (S8 to S15) will be available at the sum of the last full adders of the circuit, as shown in the diagram.

array_multiplier_8bit.vhd

```
35    component partial product creation is
41
42    component full adder 1bit is
47
48    component half adder is
53
54      signal pp_op :std_logic_vector (63 downto 0);
55      signal carry :std_logic_vector (55 downto 0);
56      signal sum :std_logic_vector (55 downto 0);
57
58
59      begin
60
61      pp: partial_product_creation port map(A,B,pp_op);
62
63      -- LSB of Result--
64      Z(0) <= pp_op(0);
65
66      ha1: half_adder
67      port map(pp_op(1),pp_op(8),Z(1),carry(1));
68      fa1: full_adder_1bit
69      port map(pp_op(2),pp_op(9),carry(1),sum(2),carry(2));
70      ha2: half_adder
71      port map(sum(2),pp_op(16),Z(2),carry(3));
72      fa2: full_adder_1bit
73      port map(pp_op(3),carry(2),pp_op(10),sum(4),carry(4));
74      fa3: full_adder_1bit
75      port map(sum(4),carry(3),pp_op(17),sum(5),carry(5));
76      ha3: half_adder
77      port map(sum(5),pp_op(24),Z(3),carry(6));
78      fa4: full_adder_1bit
79      port map(pp_op(4),carry(4),pp_op(11),sum(7),carry(7));
80      fa5: full_adder_1bit
81      port map(sum(7),carry(5),pp_op(18),sum(8),carry(8));
82      fa6: full_adder_1bit
83      port map(sum(8),carry(6),pp_op(25),sum(9),carry(9));
84      ha4: half_adder
85      port map(sum(9),pp_op(32),Z(4),carry(10));
86      fa7: full_adder_1bit
87      port map(pp_op(5),carry(7),pp_op(12),sum(11),carry(11));
88      fa8: full_adder_1bit
89      port map(sum(11),carry(8),pp_op(19),sum(12),carry(12));
90      fa9: full_adder_1bit
91      port map(sum(12),carry(9),pp_op(26),sum(13),carry(13));
92      fa10: full_adder_1bit
93      port map(sum(13),carry(10),pp_op(33),sum(14),carry(14));
94      ha5: half_adder
95      port map(sum(14),pp_op(40),Z(5),carry(15));
96      fa11: full_adder_1bit
97      port map(pp_op(6),carry(11),pp_op(13),sum(16),carry(16));
98      fa12: full_adder_1bit
99      port map(sum(16),carry(12),pp_op(20),sum(17),carry(17));
100     fa13: full_adder_1bit
101     port map(sum(17),carry(13),pp_op(27),sum(18),carry(18));
102     fa14: full_adder_1bit
103     port map(sum(18),carry(14),pp_op(34),sum(19),carry(19));
104     fa15: full_adder_1bit
```

Figure 13 VHDL code for Array Multiplier (8 bit).

/nfs/home/a/a_rajmoh/digital_design_syn/vhdl_array_multiplier/tb_array_8x8_mult.vhd

```
Ln#
13    type test_vector_8x8_mult is record
14
15      a: std_logic_vector(7 downto 0);
16      b: std_logic_vector(7 downto 0);
17      res: std_logic_vector(15 downto 0);
18    end record;
19
20    type test_vector_data is array (natural range <>) of test_vector_8x8_mult;
21
22    constant data_array: test_vector_data(14 downto 0) :=
23    (("00000010","00000001","0000000000000010"),
24    ("00000010","00000010","0000000000000100"),
25    ("00000010","00000011","0000000000000110"),
26    ("00000010","00000100","0000000000001000"),
27    ("00000010","00000101","0000000000001010"),
28    ("00000010","00000110","0000000000001100"),
29    ("00000010","00000111","0000000000001110"),
30    ("00000010","00001000","0000000000010000"),
31    ("00000010","00001001","0000000000010010"),
32    ("00000010","00001010","0000000000010100"),
33    ("00000010","00001011","0000000000010110"),
34    ("00000010","00001100","0000000000011000"),
35    ("00000010","00001101","0000000000011010"),
36    ("00000010","00001110","0000000000011100"),
37    ("00000010","00001111","0000000000011110"));
38
39    component array_multiplier_8bit is
40      Port (
41      A : in std_logic_vector (7 downto 0);
42      B : in std_logic_vector (7 downto 0);
43      Z : out std_logic_vector (15 downto 0));
44    end component;
45
46    signal a_test, b_test: std_logic_vector(7 downto 0);
47    signal res_test: std_logic_vector(15 downto 0);
48    signal test_ok: boolean;
49
50    begin
51      uut:array_multiplier_8bit
52      port map(A=>a_test, B=>b_test, Z=>res_test);
53
54        process begin
55          for i in data_array'range loop
56            a_test <= data_array(i).a;
57            b_test <= data_array(i).b;
58            wait for 10 ns;
59            if res_test /= data_array(i).res then
60              test_ok <= FALSE;
61            else
62              test_ok <= TRUE;
63            end if;
64          end loop;
```

Figure 14 Testbench for 8-bit array multiplier.

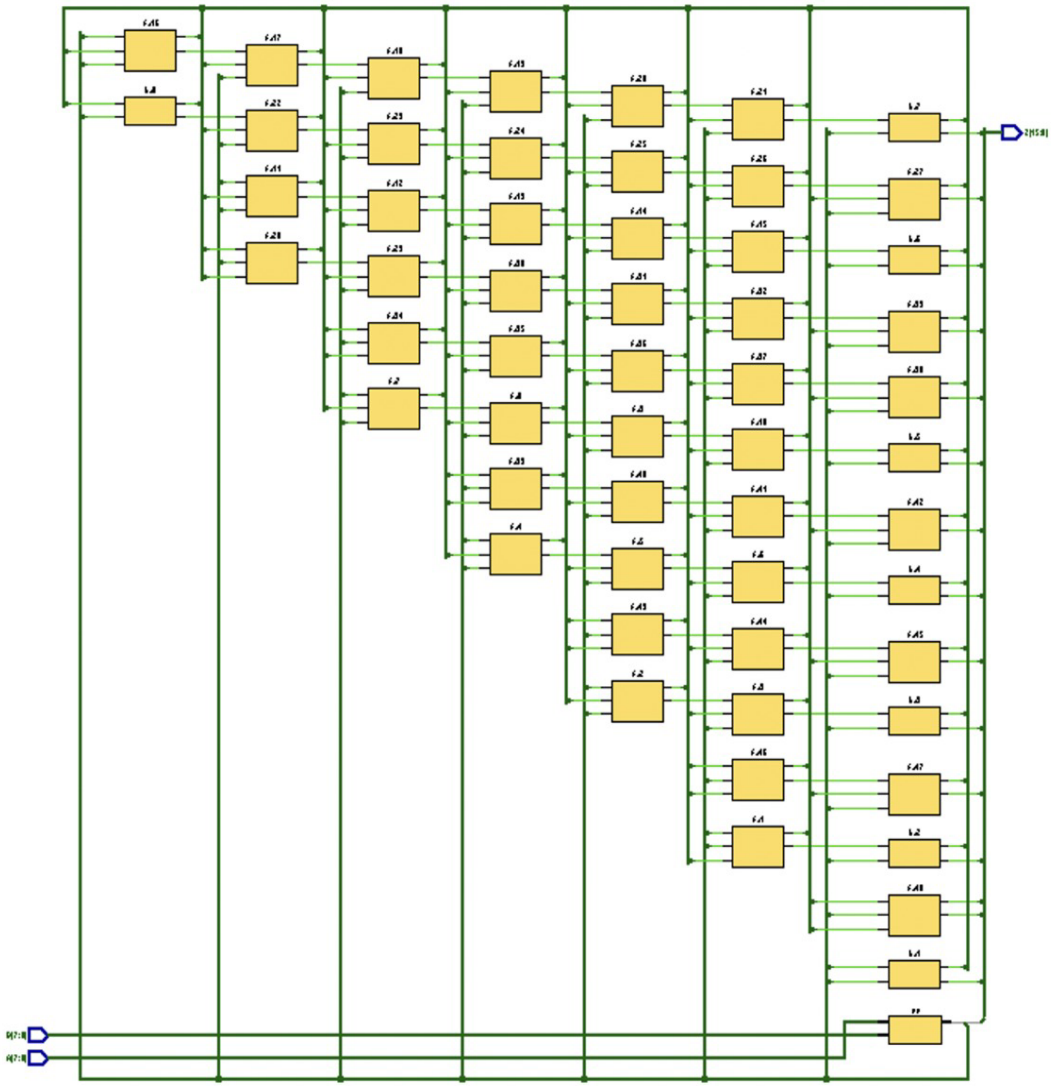


Figure 15 8-bit array multiplier synthesis diagram.

4.3.1 Array Multiplier

It is a combinational circuit used to multiply two binary numbers. The circuit is composed of half and full adders. The array multiplier uses simultaneous operations of the addition of different partial products to generate the final output.

The sequential operation requires two micro-operations: add and shift. The multiplication of binary numbers will take place through these micro-operations. This method is fast if there are fewer input bits because the propagation delay will be very small as there are only gates in the path of signals. But as the number of multipliers increases, the overall delay of the circuit will greatly increase because the number of gates associated with the operation increases.

For instance, take a 2-bit multiplier.

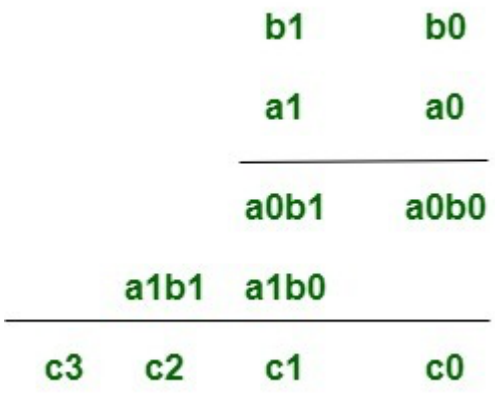


Figure 16 2-bits multiplication.[2]

B - b1, b0: Multiplicand's bits
A - a1, a0: Multiplier's bits
C - c3 c2 c1 c0: Output bits

For the above multiplication, we need 4 AND gates to create product terms like a0b0 etc. The given terms will then be propagated further to create partial products.

4.3.2 Partial Products Creation

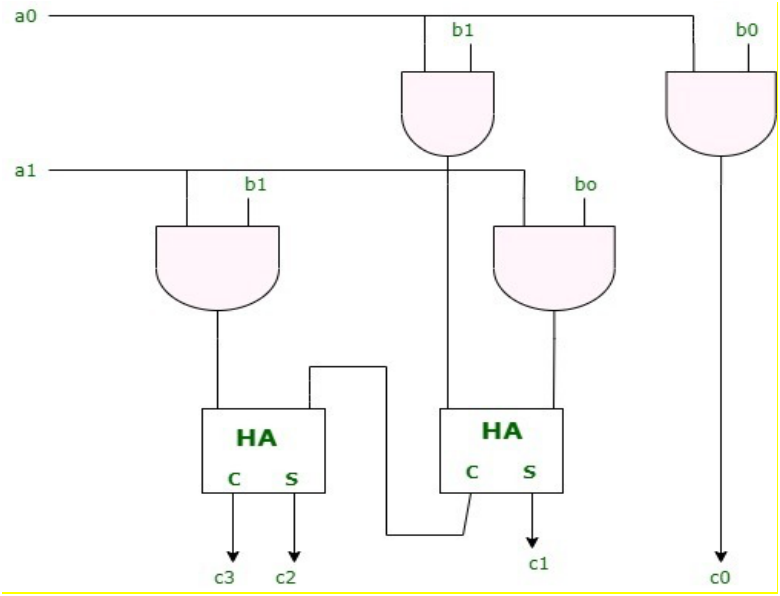


Figure 17 Array multiplier partial products.[2]

As we can observe in figure 4.2, the partial products resulting from the bit-wise multiplication will be P1, P2, P3 and P4 where the values are (a0b0), (a0b1+ a1b0), (a1b1+c1(carry after P1) and c3(carry after p2) respectively.
The first partial product generated is the multiplication of bit a0 with b1 and b0. It is the mean of two AND gates. The second partial product is generated by multiplying a1 bit with b0 and b1 and shifting one position to the left. Both the partial products are added using two half-adder circuits. When there are more partial products, it is necessary to use full adders to generate a sum. Now for the output generated, the least significant bit doesn't have to go through any operations since the first gate forms it.

In this operation, the output of each level of AND gate is added parallel with the partial product of the previous level to generate the new partial product. The levels at the end of the circuit generate the final product.

Generally, for 'n' multiplier bits and 'm' multiplicand bits, we need a total of (n*m) AND gates and (n-1)*m bit adders to generate a product of (n + m) bits.

```
/nfs/home/a/a_rajmoh/digital_design_syn/vhdl_array_multiplier/partial_product_creation.vhd
Ln#
  6  entity partial_product_creation is
  7    Port (
  8    A : in std_logic_vector (7 downto 0);
  9    B : in std_logic_vector (7 downto 0);
 10    Z : out std_logic_vector (63 downto 0));
 11  end partial_product_creation;
 12  architecture partial_product_creation_arch of partial_product_creation is
 13
 14  component AND_gate is
 15    Port ( a : in STD_LOGIC;
 16    b : in STD_LOGIC;
 17    Y : out STD_LOGIC);
 18  end component;
 19
 20  begin
 21  unit_a0: AND_gate
 22  port map(a=>A(0),b=>B(0),Y=>Z(0));
 23  unit_a1: AND_gate
 24  port map(a=>A(1),b=>B(0),Y=>Z(1));
 25  unit_a2: AND_gate
 26  port map(a=>A(2),b=>B(0),Y=>Z(2));
 27  unit_a3: AND_gate
 28  port map(a=>A(3),b=>B(0),Y=>Z(3));
 29  unit_a4: AND_gate
 30  port map(a=>A(4),b=>B(0),Y=>Z(4));
 31  unit_a5: AND_gate
 32  port map(a=>A(5),b=>B(0),Y=>Z(5));
 33  unit_a6: AND_gate
 34  port map(a=>A(6),b=>B(0),Y=>Z(6));
 35  unit_a7: AND_gate
 36  port map(a=>A(7),b=>B(0),Y=>Z(7));
 37  unit_a8: AND_gate
 38  port map(a=>A(0),b=>B(1),Y=>Z(8));
 39  unit_a9: AND_gate
 40  port map(a=>A(1),b=>B(1),Y=>Z(9));
 41  unit_a10: AND_gate
 42  port map(a=>A(2),b=>B(1),Y=>Z(10));
 43  unit_a11: AND_gate
 44  port map(a=>A(3),b=>B(1),Y=>Z(11));
 45  unit_a12: AND_gate
 46  port map(a=>A(4),b=>B(1),Y=>Z(12));
 47  unit_a13: AND_gate
 48  port map(a=>A(5),b=>B(1),Y=>Z(13));
 49  unit_a14: AND_gate
 50  port map(a=>A(6),b=>B(1),Y=>Z(14));
 51  unit_a15: AND_gate
 52  port map(a=>A(7),b=>B(1),Y=>Z(15));
 53  unit_a16: AND_gate
 54  port map(a=>A(0),b=>B(2),Y=>Z(16));
 55  unit_a17: AND_gate
 56  port map(a=>A(1),b=>B(2),Y=>Z(17));
 57  unit_a18: AND_gate
 58  port map(a=>A(2),b=>B(2),Y=>Z(18));
 59  unit_a19: AND_gate
 60  port map(a=>A(3),b=>B(2),Y=>Z(19));
```

Figure 18 VHDL code for partial product generation.

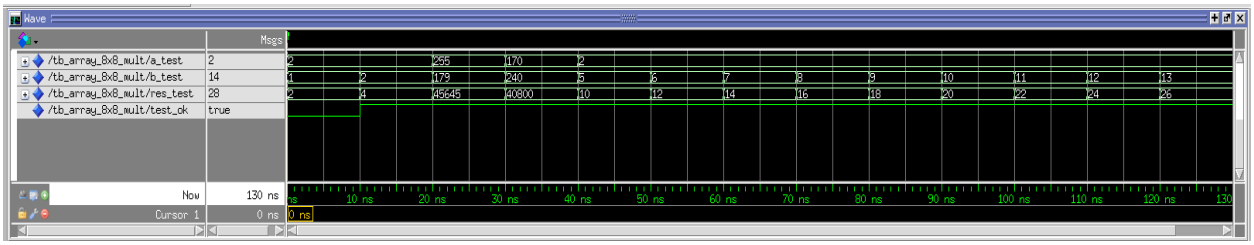


Figure 19 16-bit output generation.

4.4 ¼ Operation (Using Shifting Right)

After the multiplication of 8-bit digital numbers is done and the 16-bit output is available on the output side, the next operation we must perform on this output is to divide it by 4. We perform this operation by right shifting as explained below

- ***Right shifting the 16-bit output:***
  The approach is to right-shift the 16-bit output by 2 bits. Using the concatenation operator, we will right-shift the output, replace the two most significant bits with '00', and remove the two least significant bits. The resulting output will be divided by '4'.

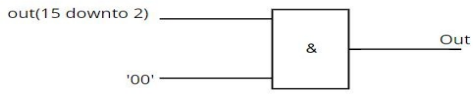


Figure 20 Conceptual Diagram of our Shift right operation.

```
/nfs/home/a/a_rajmoh/digital_design_syn/vhdl_array_multiplier/divide_by_4_unit.vhd
Ln#
 1  --*********************************************************
 2  --
 3  -- Author : Team G4
 4
 5  -- Design Unit: Divide by 4 unit
 6  --
 7  --            appends 2 zeros in the front of the given 16-bit number
 8
 9  -- inputs : a and b
10  -- Output : Y
11  --Achitecture : Behavioral architecure
12
13  -- Revision History
14  -- Version 1.0
15  -- Date: 01-12-2022
16
17  --*********************************************************
18
19  library IEEE;
20  use IEEE.STD_LOGIC_1164.ALL;
21  use IEEE.STD_LOGIC_ARITH.ALL;
22  use IEEE.STD_LOGIC_UNSIGNED.ALL;
23
24
25  entity divide_by_4_unit is
26    port(
27    data_in: in std_logic_vector(15 downto 0);
28    data_out: out std_logic_vector(15 downto 0));
29  end divide_by_4_unit;
30
31  architecture concat_arch_divide_by_4_unit of divide_by_4_unit is
32    begin
33
34    data_out <= "00" & data_in(15 downto 2);
35
36  end concat_arch_divide_by_4_unit;
37
```

Figure 21 VHDL code for the shift operation.

```
type test_vector_divide_by_4_unit is record
  data_in, data_out: std_logic_vector(15 downto 0);
end record;
type test_vector_data is array (natural range <>) of test_vector_divide_by_4_unit;

CONSTANT data_test: test_vector_data(3 downto 0):=
(("0000000000000100", "0000000000000001"),
("0000000000001000", "0000000000000010"),
("1010101010101010", "0010101010101010"),
("1100101011001011", "0011001010110010"));

component divide_by_4_unit is
port(
data_in: in std_logic_vector(15 downto 0);
data_out: out std_logic_vector(15 downto 0));
end component;

signal test_data_in,test_data_out: std_logic_vector(15 downto 0);
signal test_ok: boolean;

begin
  uut: divide_by_4_unit
  port map(data_in=>test_data_in,data_out=>test_data_out);

  process begin
    for i in data_test'range loop
      test_data_in <= data_test(i).data_in;
      wait for 10 ns;
      if test_data_out /= data_test(i).data_out then
        test_ok <= FALSE;
      else
        test_ok <= TRUE;
      end if;
    end loop;
    wait;
  end process;
end tb_arch_divide_by_4_unit;
```

Figure 22 Testbench for divide by 4 operations.

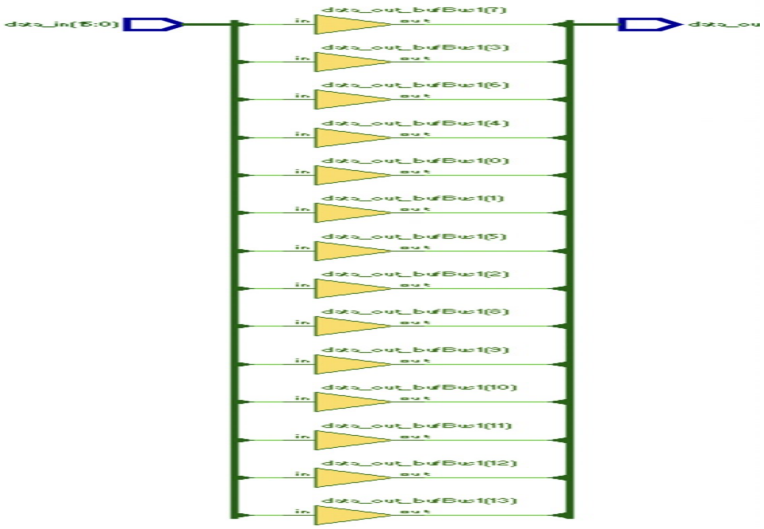


Figure 23 Divide by 4 unit.

4.5 Incrementation Operation

The last operation is incrementing the resulting output from the shifting operation by 1. This operation can be done using the '+' operator. The EDA tools can synthesize the hardware with the least area and power consumption.

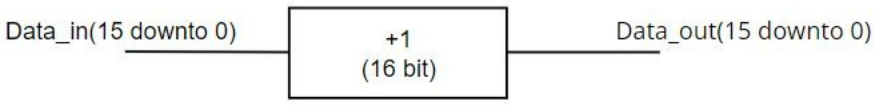


Figure 24 Conceptual diagram for increment by 1.

```
/nfs/home/a/a_rajmoh/digital_design_syn/vhdl_array_multiplier/increment_by_1_unit.vhd
Ln#
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3  use IEEE.STD_LOGIC_ARITH.ALL;
 4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
 5
 6  entity increment_by_1_unit is
 7    port(
 8      data_in: in std_logic_vector(15 downto 0);
 9      data_out: out std_logic_vector(15 downto 0) );
10
11    end increment_by_1_unit;
12
13    architecture increment_by_1_unit_arch of increment_by_1_unit is
14      CONSTANT ONE : unsigned(15 downto 0) := (0 => '1', others => '0');
15      signal data_out_temp: unsigned(15 downto 0);
16
17      begin
18        data_out_temp <= unsigned(data_in) + ONE;
19
20        data_out <= std_logic_vector(data_out_temp);
21
22      end increment_by_1_unit_arch;
23
```

Figure 25 VHDL code for increment by 1.

```
13  type test_vector_increment_by_1_unit is record
14
15    t_in: std_logic_vector(15 downto 0);
16    t_out: std_logic_vector(15 downto 0);
17
18  end record;
19
20  type test_vector_data is array (natural range <>) of test_vector_increment_by_1_unit;
21
22  constant data_array: test_vector_data(1 downto 0) :=
23  (("0000000000000001", "0000000000000010"),
24  ("0000000000000101", "0000000000000110"));
25
26  component increment_by_1_unit is
27    port(
28      data_in: in std_logic_vector(15 downto 0);
29      data_out: out std_logic_vector(15 downto 0) );
30    end component;
31
32  signal t_in, t_out: std_logic_vector(15 downto 0);
33  signal test_ok: boolean;
34
35  begin
36    uut:increment_by_1_unit
37    port map(data_in=>t_in, data_out=>t_out);
38
39      process begin
40        for i in data_array'range loop
41          t_in <= data_array(i).t_in;
42          wait for 10 ns;
43          if t_out /= data_array(i).t_out then
44            test_ok <= FALSE;
45          else
46            test_ok <= TRUE;
47          end if;
48        end loop;
49        wait;
50      end process;
51  end tb_increment_by_1_unit_arch;
52
```

Figure 26 Testbench for increment by 1 operation.

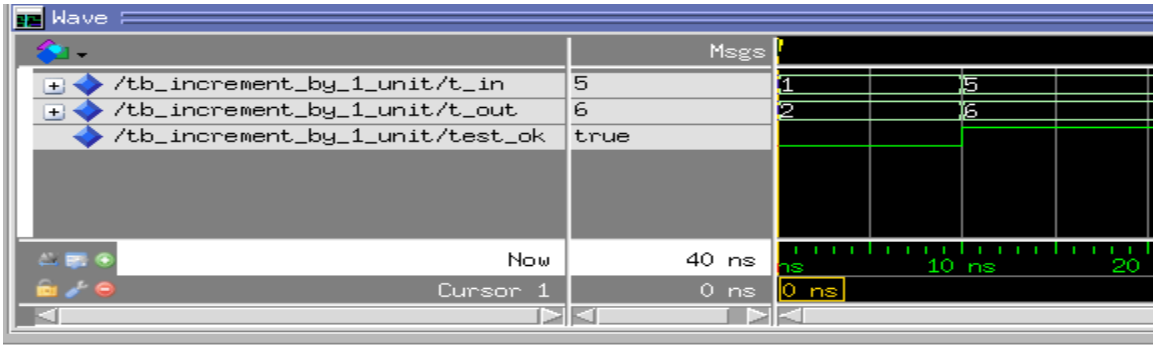


Figure 27 Increment by 1 timing diagram.

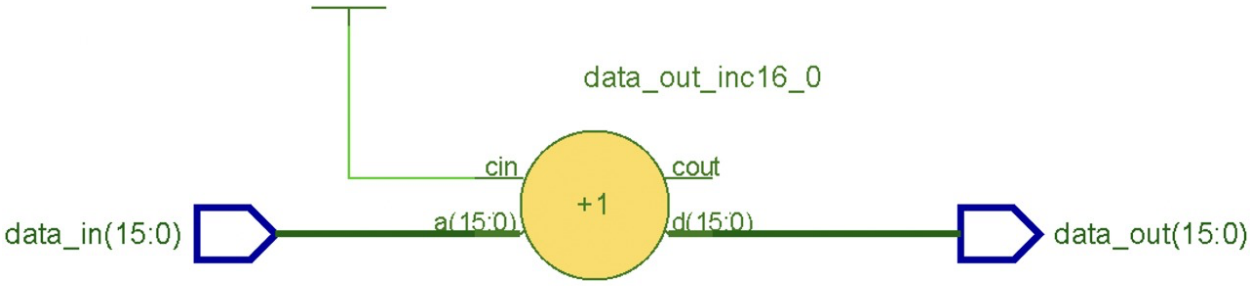


Figure 28 Increment by 1 synthesis diagram.

4.6 8-Bit and 16-Bit Registers

Initially, when the 8-bit inputs are received, they must be stored in two different 8-bit registers. We have used two 8-bit registers, A_reg and B_reg.

We have two 8-bit input registers and one 16-bit output register. These registers are designed using '+ve' edge-triggered d-flipflops with an asynchronous reset and synchronous enable (load_in and load_out).

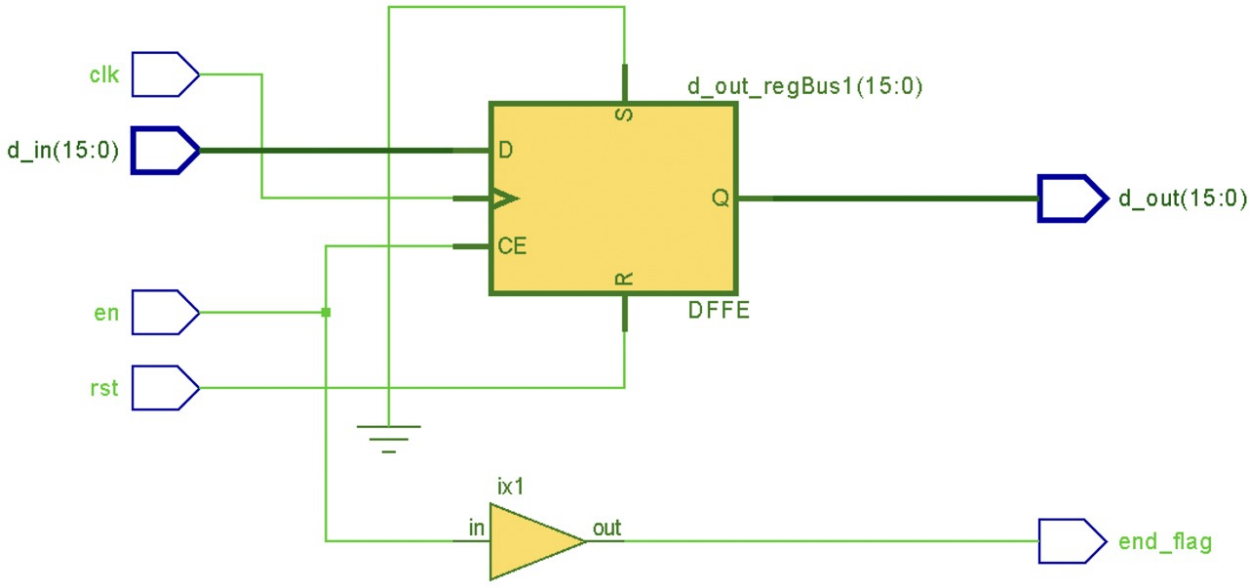


Figure 29 16-Bit register synthesis diagram.

4.7 END Flag Design

As per the design requirement, an end flag must be raised after completing the arithmetic operation. To design the end flag, we have assumed the following,

- The output register enable signal ***(load_out)*** is asserted only when the output is ready. This is done to avoid glitches from the multiplier, shifter and incrementor combinational circuits to pass on to the output side.

So, whenever load_out is asserted, the end flag is also asserted.

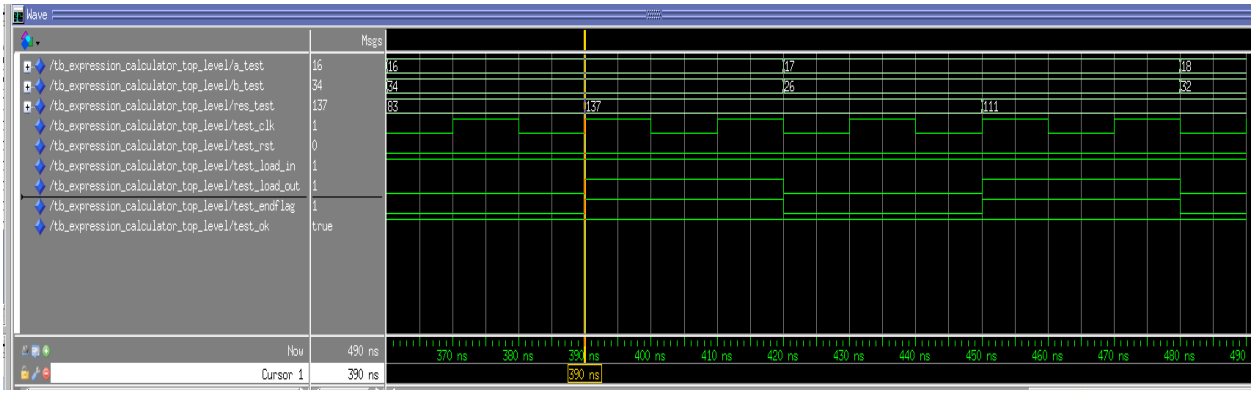


Figure 30 END flag output.

4.8 Testbench and Verifier Design

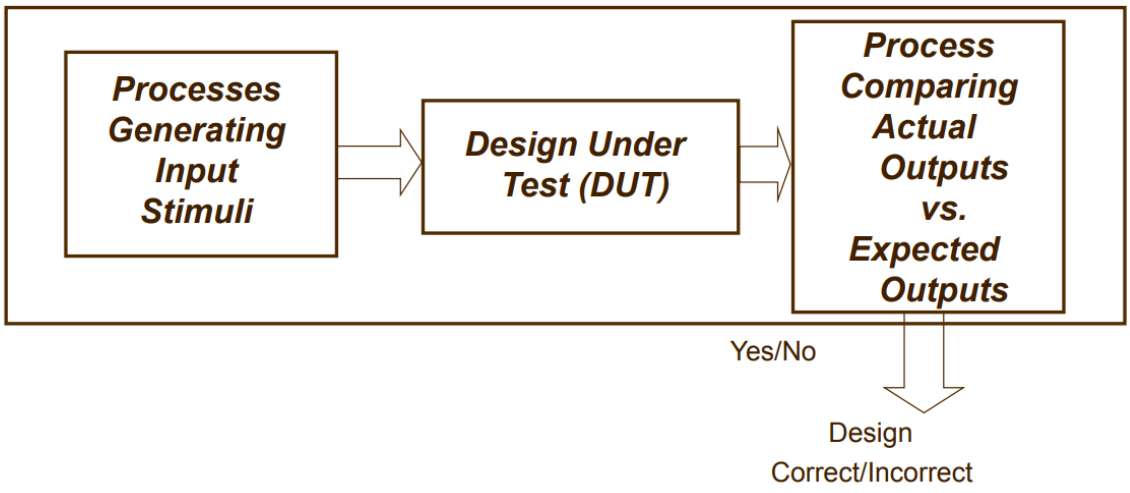


Figure 31 Testbench and verifier design.

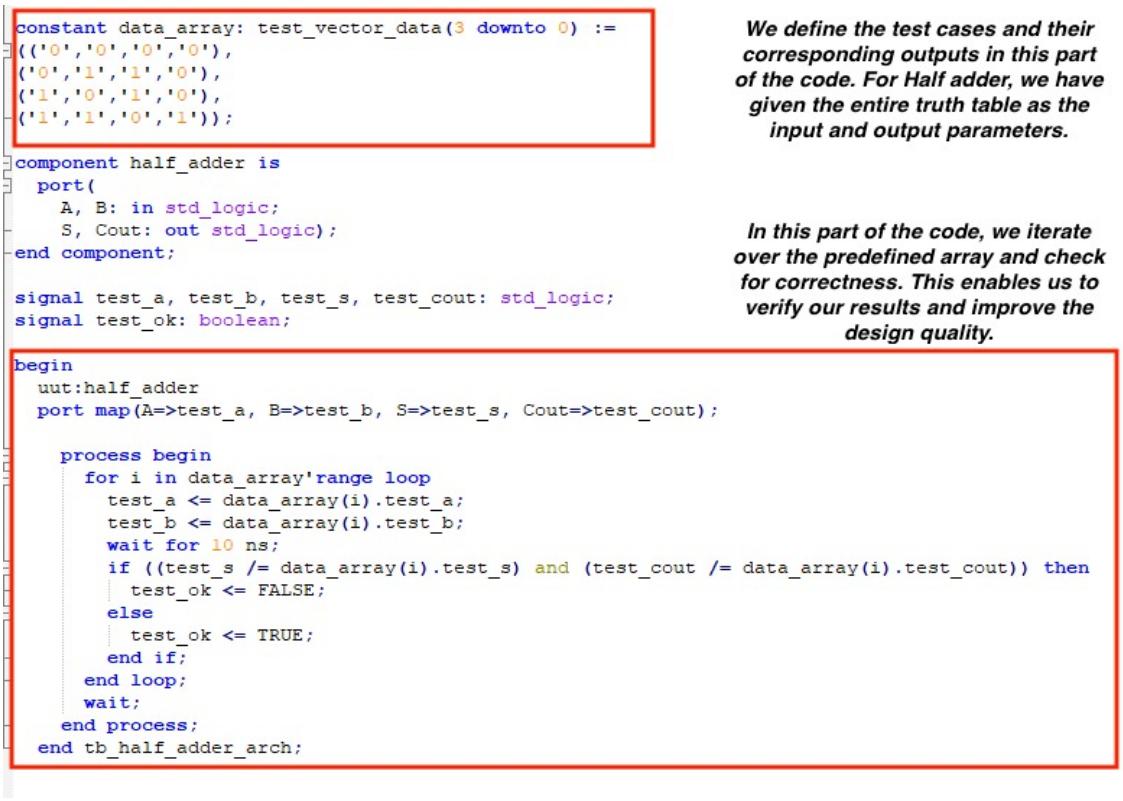


Figure 32 Half adder testbench design

```
("10100010","11111010","1001111000110100"),
("10100110","11011100","1000111010101000"),
("10101011","10110001","0111011000111011"),
("10101110","10110100","0111101001011000"),
("10100100","10100000","0110011010000000"),
("10110010","11111110","1011000010011100"),
("10101111","10100011","0110111101101101"),
("01111100","10111110","0101110000001000"),
("01111111","11000100","0110000100111100"),
("01111011","11111011","0111100010011001"),
("01111101","11100111","0111000011001011"),
("10000001","11101101","0111011101101101"),
("11111111","11111111","1111111000000001"));

component array_multiplier_8bit is
  Port (
  A : in std_logic_vector (7 downto 0);
  B : in std_logic_vector (7 downto 0);
  Z : out std_logic_vector (15 downto 0));
end component;

signal a_test, b_test: std_logic_vector(7 downto 0);
signal res_test: std_logic_vector(15 downto 0);
signal test_ok: boolean;

begin
  uut:array_multiplier_8bit
  port map(A=>a_test, B=>b_test, Z=>res_test);

    process begin
      for i in data_array'range loop
        a_test <= data_array(i).a;
        b_test <= data_array(i).b;
        wait for 10 ns;
        if res_test /= data_array(i).res then
          test_ok <= FALSE;
        else
          test_ok <= TRUE;
        end if;
      end loop;
      wait;
    end process;
  end tb_array_8x8_mult_arch;
```

Here we define the test cases for a more complex design, an 8-bit array multiplier. We apply a similar input-output combination. Verifying an array multiplier is more complex as the number of bits involved increases. So, designing a verifier eases the ASIC designer's life. We can ensure good design quality and achieve full test coverage using this test bench and verifier design.

Figure 33 Array multiplier testbench design

5. Top Level Module

The top-level module is designed by port-mapping all the structurally designed units explained above to form the overall design, which will be able to perform ***Z=(A*B)/4 + 1.*** The below code snippet shows how this is achieved.

expression_calculator_top_level.vhd

```
entity expression_calculator_top_level is
  port(
    A, B: in std_logic_vector(7 downto 0);
    clk, rst, load_in, load_out: in std_logic;
    end_flag: out std_logic;
    Z: out signed(15 downto 0));

end expression_calculator_top_level;

architecture expression_calculator_top_level_arch of expression_calculator_top_level is

  component register_8bit is

  component array_multiplier_8bit is

  component divide_by_4_unit is

  component increment_by_1_unit is

  component register_16bit is

  signal multiplier_in_A, multiplier_in_B: std_logic_vector(7 downto 0);
  signal divide_4_input, increment_unit_input, output_reg_input: std_logic_vector(15 downto 0);

  begin
    ----------------------------Input stage----------------------------------
    reg_A: register_8bit
    port map (d_in=>A, clk=>clk, rst=>rst, en=>load_in, d_out=>multiplier_in_A );

    reg_B: register_8bit
    port map (d_in=>B, clk=>clk, rst=>rst, en=>load_in, d_out=>multiplier_in_B);
    ----------------------------Stage - 1-----------------------------------
    stage_1_multiplication: array_multiplier_8bit
    port map (A=>multiplier_in_A, B=>multiplier_in_B, Z=>divide_4_input );

    ----------------------------Stage - 2-----------------------------------
    stage_2_divide_by_4: divide_by_4_unit
    port map(data_in=>divide_4_input, data_out=> increment_unit_input);

    ----------------------------Stage - 3-----------------------------------
    stage_3_increment_by_1: increment_by_1_unit
    port map(data_in=>increment_unit_input, data_out=> output_reg_input);

    ----------------------------Output stage--------------------------------
    output_stage: register_16bit
    port map(d_in=>output_reg_input, clk=>clk, rst=>rst, en=>load_out,end_flag=>end_flag, d_out=> Z);

  end expression_calculator_top_level_arch;
```

Figure 34 Top-level VHDL code.

As discussed in the section above, we use an advanced test bench to test our design using record statements and test-vector arrays. These test vector arrays and record statements felicitate the test bench with verifier capability. This allows the designer to test many test cases and produce an error-free design.

```
architecture tb_expression_calculator_top_level_arch of tb_expression_calculator_top_level is

type test_vector_toplevel is record

   a: std_logic_vector(7 downto 0);
   b: std_logic_vector(7 downto 0);
   res: signed(15 downto 0);
   load_in : std_logic;
   load_out :std_logic;
end record;

type test_vector_data is array (natural range <>) of test_vector_toplevel;

constant data_array: test_vector_data(24 downto 0) :=
(("00001000","00000101","0000000000001011",'1','1'),
("00000010","00001010","0000000000000110",'1','1'),
("10101010","11001100","0010000111011111",'1','1'),
("00011101","01011101","0010000111011111",'0','0'),     -- when input Load is 0, inputs are changed but output remains the same as previous
("00001111","00010110","0000000001010011",'1','1'),
("00010000","00100010","0000000010001001",'1','1'),
("00010001","00011010","0000000001101111",'1','1'),
("00010010","00100000","0000000010010001",'1','1'),
("00010011","00111000","0000000100001011",'1','1'),
("00010100","00100001","0000000010100110",'1','1'),
("01000001","00101100","0000001011001100",'1','1'),
("01000100","00110001","0000001101000010",'1','1'),
("01000101","00110100","0000001110000010",'1','1'),
("00111110","00110010","0000001100001000",'1','1'),
("01000010","00010100","0000000101001011",'1','1'),
("01000111","01001101","0000010101010111",'1','1'),
("01001010","01010000","0000010111001001",'1','1'),
("01000000","00111100","0000001111000001",'1','1'),
("01001110","00110110","0000010000011110",'1','1'),
("01001011","00111111","0000010010011110",'1','1'),
("00011000","01011010","0000001000011101",'1','1'),
("00011011","01100000","0000001010001001",'1','1'),
("00010111","01100100","0000001001000000",'1','1'),
("00011001","01100011","0000001001101011",'1','1'),
("00011101","01011101","0000001010100011",'1','1'));

component expression_calculator_top_level is
  port(

  end component;

signal a_test, b_test: std_logic_vector(7 downto 0) := (others=>'0');
signal res_test: signed(15 downto 0) := (others=>'0');
signal test_clk, test_rst, test_load_in,test_load_out, test_endflag: std_logic := '0';
```

Figure 35 Top-level Testbench and verifier VHDL code.

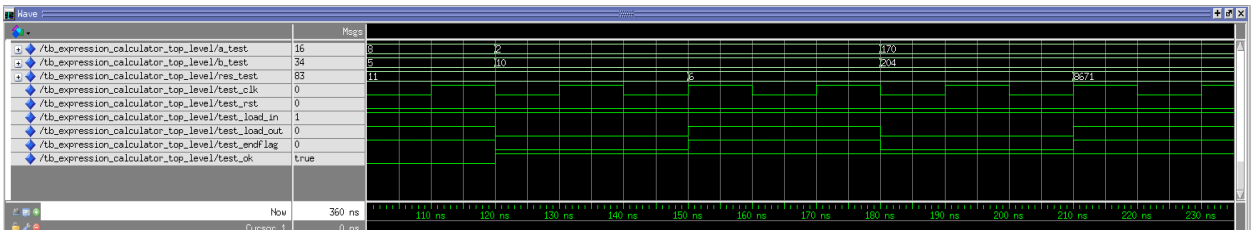


Figure 36 Top-level timing diagram

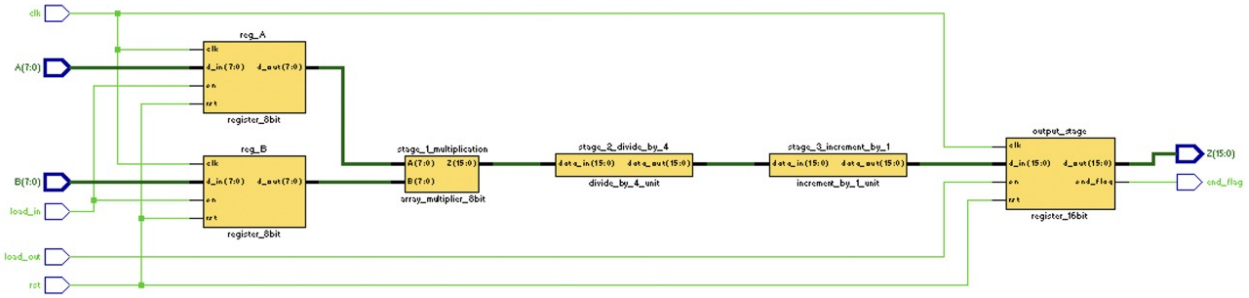


Figure 37 Top-level synthesis diagram.

6. Future Scope:

The idea behind using an array multiplier is the ease of design and its repetitive structure. With the advantages, there come some disadvantages too. To address those disadvantages, we plan to implement this array multiplier into a two's complement multiplier. The major advantage of two's complement multiplier is that it is composed of different Full adders. The last line of the array is a serial adder responsible for the multiplier's speed.

Pipelining can be achieved by implementing the two's complement multiplier. We can reduce the delay in the array multiplier by achieving the pipelining. By making certain modifications in the two's complement array multiplier, we can achieve improved results compared to a simple array multiplier**. [3]**

We have already used an advanced test bench in this project. To add more features, we can use python scripts to generate the test vectors and make the testing more automated.

7. References

[1] **Själander, Magnus & Larsson-Edefors, Per. (2009). Multiplication Acceleration Through Twin Precision. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on. 17. 1233 - 1246. 10.1109/TVLSI.2008.2002107.**

[2] **https://www.geeksforgeeks.org/array-multiplier-in-digital-logic/#:~:text=An%20array%20multiplier%20is%20a,the%20various%20product%20terms%20involved.**

[3] **Z. -y. Yang and J. -q. Xiao, "The design and simulation of array multiplier improved with pipeline techniques," Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology, 2011, pp. 4326-4329, DOI: 10.1109/EMEIT.2011.6023995**

8. Contributions:

Name	Contributions
Akash Rajmohan	Top Level Design, Multiplier Design and Testbench
Prithvik Adithya Ravindran	Half and Full Adder, Shifter Design and Testbench
Krupal Jayantibhai Dudhat	8 &16 bit-Registers, End flag design Incrementor Design and Testbench