Assignment 2: Quarter Precision Floating Point Multiplier

Aaron John Sabu: 170070050

April 20, 2019

1 Overview of the assignment

- The aim of this assignment is to design a **VHDL** code to simulate the functionality of a **quarter-precision floating-point multiplier**.
- Furthermore, this experiment involves the testing of the code using some predefined test cases generated using a C++ script.

2 Components

The construction of the quarter-precision multiplier involves the direct use of an XOR Gate (for the sign bit), a five-bit adder (for the exponent terms) and a ten-bit multiplier (for the mantissa) which is implied in the code of the whole multiplier.
As part of the multiplier, the code consists of a twelve-bit adder and a multiplexer through which the second input is passed as the MUX control whereas the first input is passed as the MUX input. Hence the whole structure is put together to give the overall multiplier:

3 Design

The input A is split into three part: the sign bit of 1 bit, the exponent of 3 bits and the mantissa of 5 bits. A similar pattern is followed for the input B. The sign bits are passed through an XOR Gate to determine the sign bit for

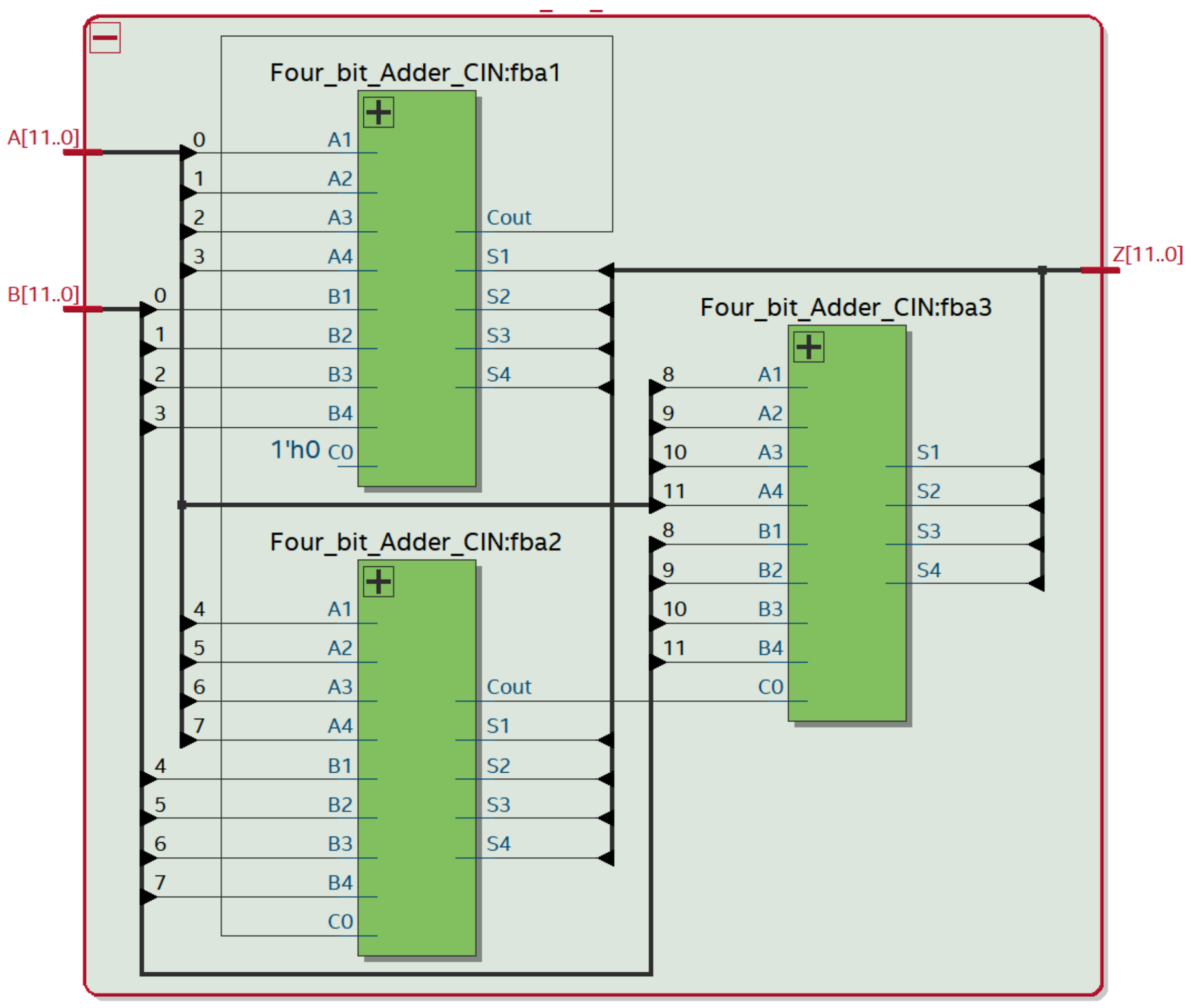


Figure 1: Twelve-bit Adder

the output; i.e. the sign of the output is positive (0) if both inputs are positive (0,0) or both are negative (1,1) and the sign is negative (1) otherwise. The exponent bits cause a shift in the number from the biased exponent. Hence the exponent bits are added together, then the input biases are removed and the output bias is added:

$$e_{OUT} = e_1 + e_2 - 2bias_{IN} + bias_{OUT}$$

$$i.e. e_{OUT} = e_1 + e_2 - 2 \times (4-1) + (16-1) = e_1 + e_2 + 9$$

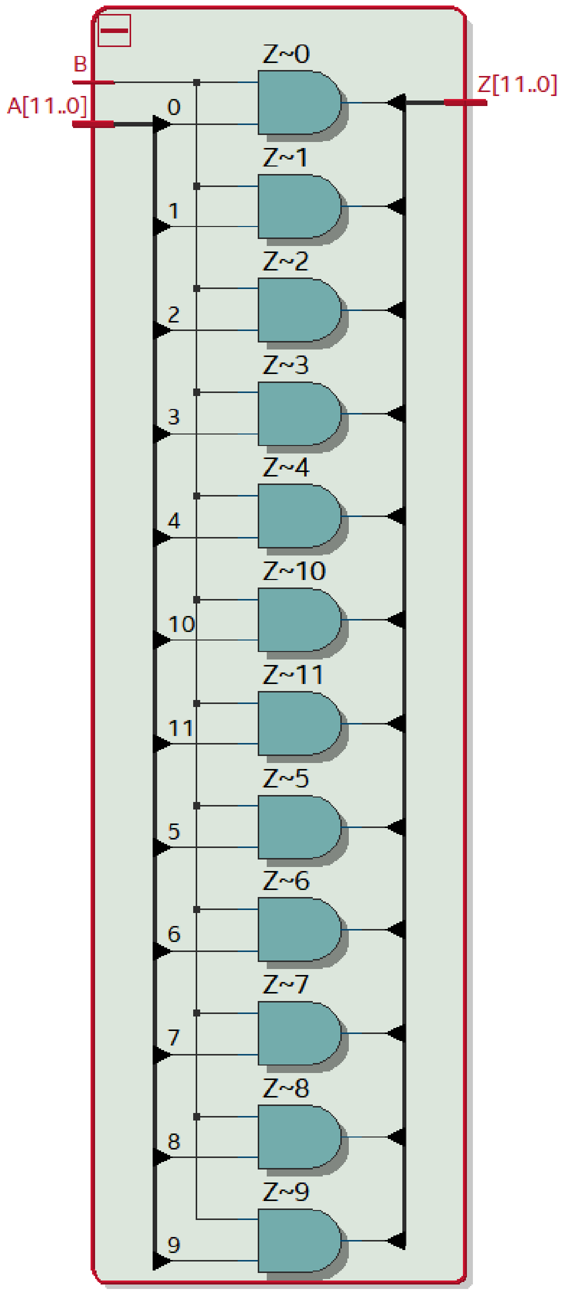


Figure 2: 11-Channel-1-bit-Multiplexer

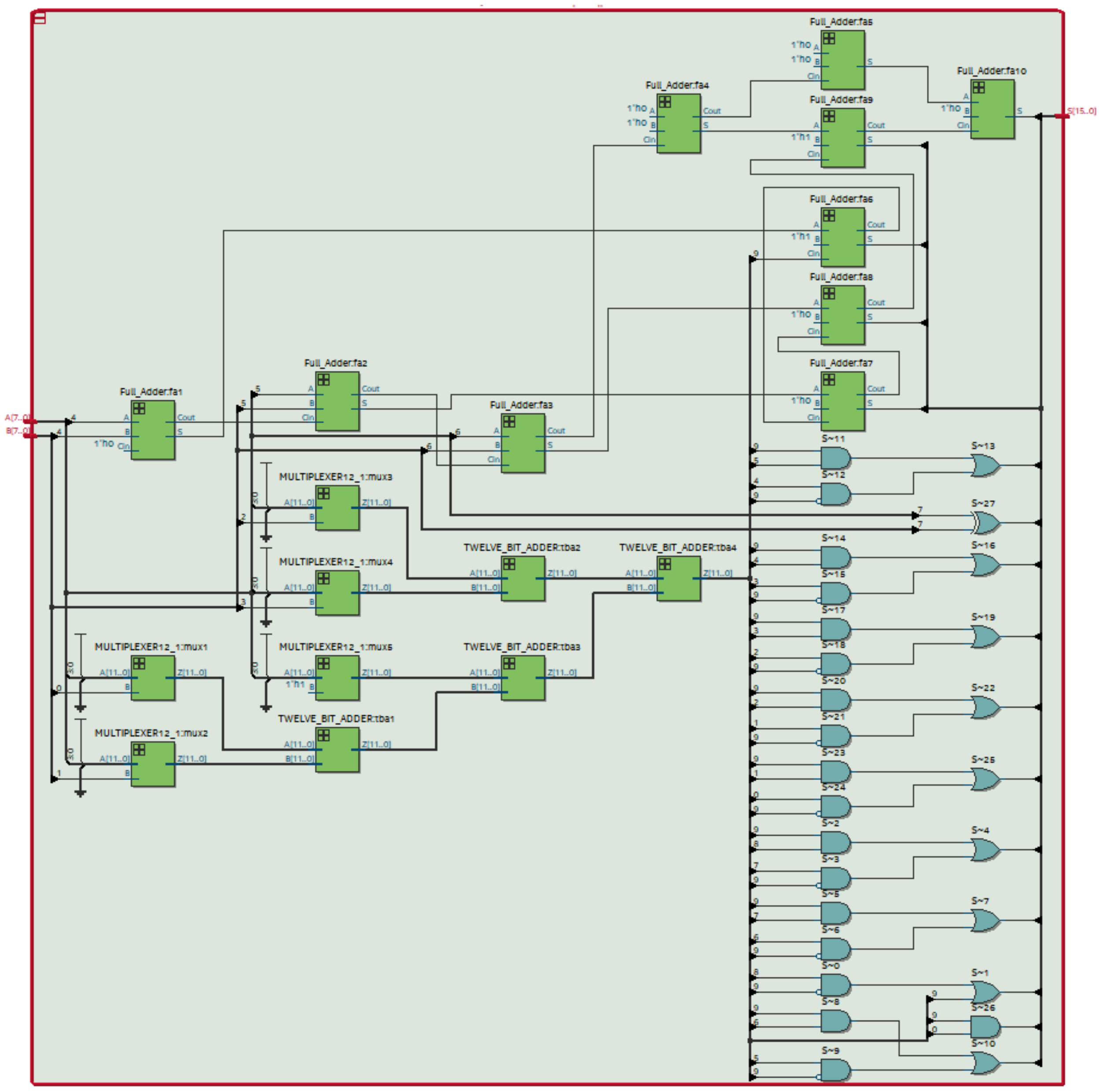


Figure 3: Multiplier

Alongside, the mantissa, representing the decimal part of a number, if rep-

resented as $a_1a_2a_3a_4a_5$, is considered to represent the number $1.a_1a_2a_3a_4a_5$. The hence obtained numbers from inputs A and B are multiplied to give an output which is removed of the first non-zero digit and considered as the mantissa of the final output.
The hence obtained parts, sign, exponent and mantissa are put together to provide the final 16-bit output. The overall circuit behaves as shown below:

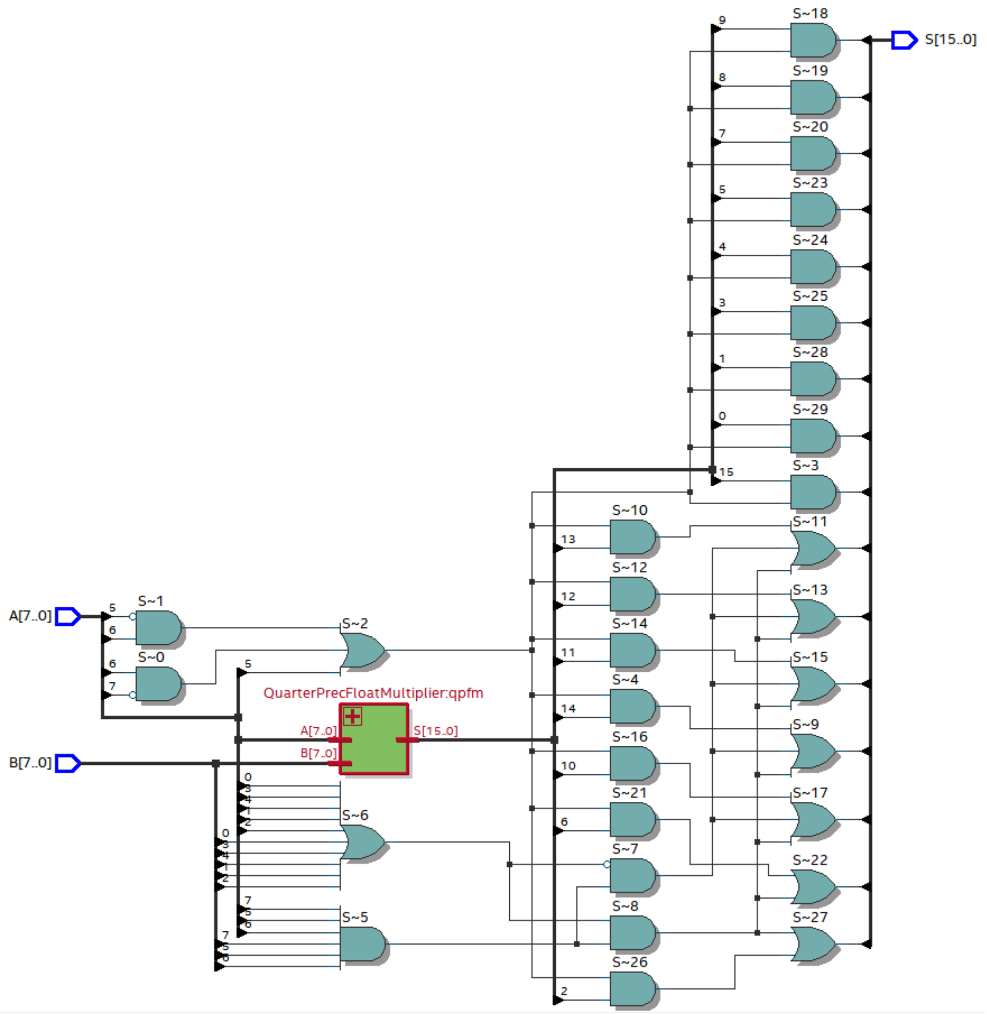


Figure 4: Overall design

In order to take care of cases such as NotANumber (NaN) and infinity (inf), the main code consists of an elaborately described multiplexer func-

tionality which selects the output from the multiplier output, infinity or NaN. (An occurrence of zero or a subnormal number is considered in the multiplier code by itself).
In the Main.vhdl code (which is considered to be the top-level entity), the output value for a NaN possibility is hard-coded as 0-11111-0101010101, whereas the output value for infinity remains as the standard IEEE 754 recommended value: 0-11111-0000000000.

4 VHDL Codes

Gates.vhdl, **Four_bit_Adder.vhdl** and **DUT.vhdl** have been used as basic codes to the project. Moreover, **tracefile_generator.cpp** has been used to generate the TRACEFILE.txt file. The codes for the main code, the sixteen-bit-adder, the custom multiplexer and the C++ code to generate the tracefile are given below:

4.1 QuarterPrecFloatMultiplier.vhdl

```
use ieee.std_logic_1164.all;
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.Gates.all;
entity QuarterPrecFloatMultiplier is
  port(A, B: in std_logic_vector(7 downto 0);
    S: out std_logic_vector(15 downto 0));
end entity;

architecture Struct of QuarterPrecFloatMultiplier is
  component TWELVE_BIT_ADDER is
    port (A, B: in std_logic_vector(11 downto 0);
      Z: out std_logic_vector(11 downto 0));
  end component TWELVE_BIT_ADDER;
  component MULTIPLEXER12_1 is
    port (B: in std_logic; A: in std_logic_vector(11 downto 0);
      Z: out std_logic_vector(11 downto 0));
  end component MULTIPLEXER12_1;
  type matrix12x5 is array(0 to 4) of std_logic_vector(11 downto 0);
  type matrix12x4 is array(0 to 3) of std_logic_vector(11 downto 0);
  signal MA : std_logic_vector(4 downto 0);
  signal A12, B12 : matrix12x5;
  signal C12 : matrix12x4;
  signal S1, S2, S3, S4, S5, C1, C2, C3, C4, C5,
    D1, D2, D3, D4, D5 : std_logic;
begin
  MA <= '1' & A(3 downto 0);
  A12(0) <= ("0000000" & MA);
  A12(1) <= ("000000" & MA) & '0';
  A12(2) <= ("00000" & MA) & "00";
  A12(3) <= ("0000" & MA) & "000";
  A12(4) <= ("000" & MA) & "0000";
  mux1: MULTIPLEXER12_1 port map(A => A12(0), B => B(0), Z => B12(0));
  mux2: MULTIPLEXER12_1 port map(A => A12(1), B => B(1), Z => B12(1));
  mux3: MULTIPLEXER12_1 port map(A => A12(2), B => B(2), Z => B12(2));
  mux4: MULTIPLEXER12_1 port map(A => A12(3), B => B(3), Z => B12(3));
  mux5: MULTIPLEXER12_1 port map(A => A12(4), B => '1',  Z => B12(4));
  tba1: TWELVE_BIT_ADDER port map(A => B12(0), B => B12(1), Z => C12(0));
  tba2: TWELVE_BIT_ADDER port map(A => B12(2), B => B12(3), Z => C12(1));
  tba3: TWELVE_BIT_ADDER port map(A => B12(4), B => C12(0), Z => C12(2));
  tba4: TWELVE_BIT_ADDER port map(A => C12(1), B => C12(2), Z => C12(3));

  S(9)  <= (C12(3)(9)  and C12(3)(9)) or (C12(3)(8)  and not(C12(3)(9)));
  S(8)  <= (C12(3)(8)  and C12(3)(9)) or (C12(3)(7)  and not(C12(3)(9)));
  S(7)  <= (C12(3)(7)  and C12(3)(9)) or (C12(3)(6)  and not(C12(3)(9)));
  S(6)  <= (C12(3)(6)  and C12(3)(9)) or (C12(3)(5)  and not(C12(3)(9)));
  S(5)  <= (C12(3)(5)  and C12(3)(9)) or (C12(3)(4)  and not(C12(3)(9)));
  S(4)  <= (C12(3)(4)  and C12(3)(9)) or (C12(3)(3)  and not(C12(3)(9)));
  S(3)  <= (C12(3)(3)  and C12(3)(9)) or (C12(3)(2)  and not(C12(3)(9)));
  S(2)  <= (C12(3)(2)  and C12(3)(9)) or (C12(3)(1)  and not(C12(3)(9)));
  S(1)  <= (C12(3)(1)  and C12(3)(9)) or (C12(3)(0)  and not(C12(3)(9)));
  S(0)  <= (C12(3)(0)  and C12(3)(9)) or ('0'        and not(C12(3)(9)));

  fa1: FULL_ADDER port map(A => A(4), B => B(4), Cin => '0',
    S => S1, Cout => C1);
  fa2: FULL_ADDER port map(A => A(5), B => B(5), Cin => C1,
    S => S2, Cout => C2);
  fa3: FULL_ADDER port map(A => A(6), B => B(6), Cin => C2,
    S => S3, Cout => C3);
  fa4: FULL_ADDER port map(A => '0',  B => '0',  Cin => C3,
    S => S4, Cout => C4);
  fa5: FULL_ADDER port map(A => '0',  B => '0',  Cin => C4,
    S => S5, Cout => C5);
  fa6: FULL_ADDER port map(A => S1, B => '1',  Cin => C12(3)(9),
    S => S(10), Cout => D1);
  fa7: FULL_ADDER port map(A => S2, B => '0',Cin => D1,
    S => S(11), Cout => D2);
  fa8: FULL_ADDER port map(A => S3, B => '0',Cin => D2,
    S => S(12), Cout => D3);
  fa9: FULL_ADDER port map(A => S4, B => '1',  Cin => D3,
    S => S(13), Cout => D4);
  fa10: FULL_ADDER port map(A => S5, B => '0',  Cin => D4,
    S => S(14), Cout => D5);

  S(15) <= A(7) xor B(7);
end Struct;
```

4.2 Twelve_bit_Adder.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.Gates.all;
entity TWELVE_BIT_ADDER is
  port (A, B: in std_logic_vector(11 downto 0);
    Z: out std_logic_vector(11 downto 0));
end entity TWELVE_BIT_ADDER;
architecture Struct of TWELVE_BIT_ADDER is
  signal C1, C2, C3 : std_logic;
  component Four_bit_Adder_CIN  is
    port (A1, A2, A3, A4, B1, B2, B3, B4, C0: in std_logic;
      S1, S2, S3, S4, Cout: out std_logic);
  end component Four_bit_Adder_CIN;
begin
```

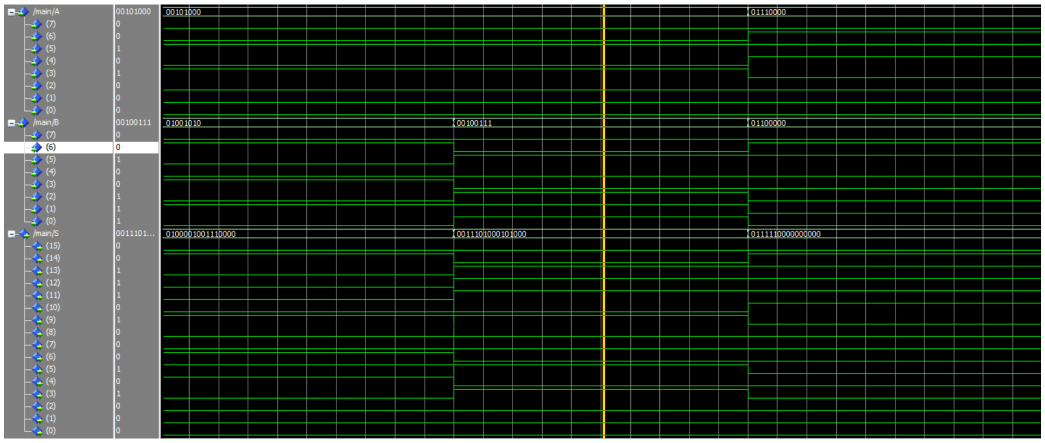
```
  fba1: FOUR_BIT_ADDER_CIN port map (A1 => A(0), A2 => A(1), A3  => A(2),
    A4 => A(3),  C0 => '0', B1 => B(0), B2 => B(1), B3 => B(2),
    B4 => B(3),  S1 => Z(0), S2 => Z(1), S3 => Z(2),  S4 => Z(3),  Cout => C1);
  fba2: FOUR_BIT_ADDER_CIN port map (A1 => A(4), A2 => A(5), A3  => A(6),
    A4 => A(7),  C0 => C1,  B1 => B(4),B2 => B(5), B3 => B(6),  B4 => B(7),
    S1 => Z(4), S2 => Z(5), S3 => Z(6),  S4 => Z(7),  Cout => C2);
  fba3: FOUR_BIT_ADDER_CIN port map (A1 => A(8), A2 => A(9), A3  => A(10),
    A4 => A(11), C0 => C2,B1 => B(8),B2 => B(9), B3 => B(10), B4 => B(11),
    S1 => Z(8), S2 => Z(9), S3 => Z(10), S4 => Z(11), Cout => C3);
end Struct;
```

4.3 Multiplexer.vhdl

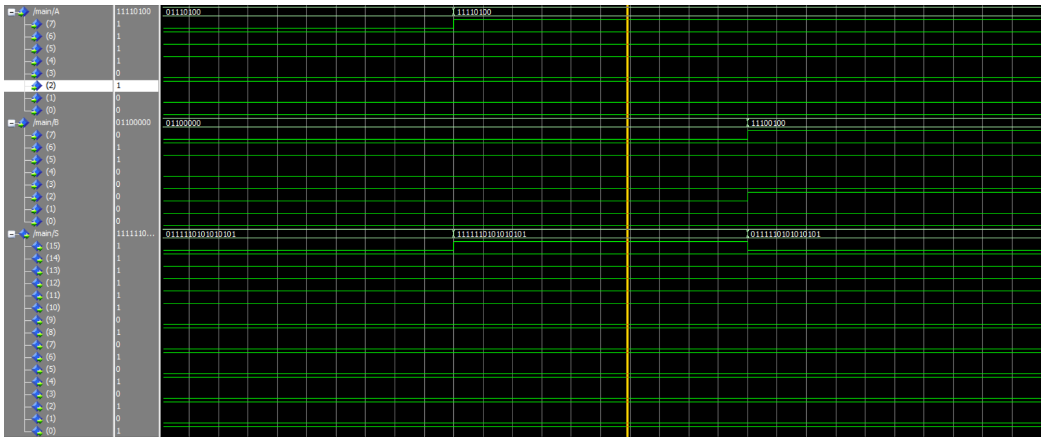
```
library ieee;
use ieee.std_logic_1164.all;
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.Gates.all;
entity MULTIPLEXER12_1 is
  port (B: in std_logic; A: in std_logic_vector(11 downto 0);
    Z: out std_logic_vector(11 downto 0));
end entity MULTIPLEXER12_1;
architecture Struct of MULTIPLEXER12_1 is
begin
  Z(0)  <= A(0)  and B;
  Z(1)  <= A(1)  and B;
  Z(2)  <= A(2)  and B;
  Z(3)  <= A(3)  and B;
  Z(4)  <= A(4)  and B;
  Z(5)  <= A(5)  and B;
  Z(6)  <= A(6)  and B;
  Z(7)  <= A(7)  and B;
  Z(8)  <= A(8)  and B;
  Z(9)  <= A(9)  and B;
  Z(10) <= A(10) and B;
  Z(11) <= A(11) and B;
end Struct;
```

5 Output

Given below are snapshots of some portions of the digital waveform obtained from the multiplier simulation in ModelSim-Altera:



(a) Wave 1



(b) Wave 2

Figure 5: Digital Waveform

Also given below is a portion of the TRACEFILE generated from the code given above:

```
0000000000000000 0000000000000000 1111111111111111
```

```
0000000000000001 0000000000000000 1111111111111111
0000000000000010 0000000000000000 1111111111111111
0000000000000011 0000000000000000 1111111111111111
0000000000000100 0000000000000000 1111111111111111
0000000000000101 0000000000000000 1111111111111111
0000000000000110 0000000000000000 1111111111111111
...
0000010110010001 1010100110010100 1111111111111111
0000010110010010 1010100111101000 1111111111111111
0000010110010011 1010101000111100 1111111111111111
0000010110010100 1010101010010000 1111111111111111
0000010110010101 1010101011100100 1111111111111111
0000010110010110 1010101100111000 1111111111111111
0000010110010111 1010101110001100 1111111111111111
...
1000101100111001 1011010101000110 1111111111111111
1000101100111010 1011010101111100 1111111111111111
1000101100111011 1011010110110010 1111111111111111
1000101100111100 1011010111101000 1111111111111111
1000101100111101 1011011000011110 1111111111111111
1000101100111110 1011011001010100 1111111111111111
1000101100111111 1011011010001010 1111111111111111
...
1111111111111001 0111110000000001 1111111111111111
1111111111111010 0111110000000001 1111111111111111
1111111111111011 0111110000000001 1111111111111111
1111111111111100 0111110000000001 1111111111111111
1111111111111101 0111110000000001 1111111111111111
1111111111111110 0111110000000001 1111111111111111
1111111111111111 0111110000000001 1111111111111111
```